Marcin Moskała

Kotlin Coroutines

DEEP DIVE

Third Edition

Kotlin Coroutines

Deep Dive

Marcin Moskała

This book is available at https://leanpub.com/coroutines

This version was published on 2025-06-20

ISBN 978-83-963958-1-8

Tweet This Book!

Please help Marcin Moskała by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Kotlin Coroutines by Marcin Moskała

The suggested hashtag for this book is #KotlinCoroutines.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#KotlinCoroutines

For my beloved wife, Maja.

Contents

Introduction

Why do you want to learn about Kotlin Coroutines? I often ask my workshop attendees this question. “Because they’re cool” and “Because everyone is talking about them” are common answers. Then, when I dig deeper, I hear responses like “because they are lighter threads”, “because they are easier than RxJava”, or “because they enable concurrency while maintaining an imperative code style.” However, coroutines are much more than that. They are the holy grail of concurrency. As a concept, they have been known in Computer Science since the 1960s, but in mainstream programming, they have only been used in a very limited form (such as async/await). This changed with Golang, which introduced much more general-purpose coroutines. Kotlin built on that, creating what I believe is currently the most powerful and practical implementation of this idea.

The importance of concurrency is growing, but traditional techniques are not enough. Current trends suggest that coroutines are the direction in which our industry is clearly heading, and Kotlin Coroutines represent a significant step forward. Let me introduce them to you, along with examples of how well they handle common use cases. I hope you will have a lot of fun reading this book.

Who is this book for?

As a developer experienced in both backend and Android, I primarily focus on these two perspectives in this book. These are currently the two major industry applications of Kotlin, and it’s evident that coroutines were largely designed to suit these use cases well[1]. So, you might say that this book is mainly designed for Android and backend developers, but it should be just as useful for other developers using Kotlin.

This book assumes that readers know Kotlin well. If you do not, I recommend starting with my other book, *Kotlin Essentials*.

The structure of this book

The book is divided into the following parts:

[1]Google’s Android team cooperated in designing and creating some features we will present in this book.

- **Part 1: Understanding Kotlin Coroutines** - dedicated to explaining what Kotlin Coroutines are and how they really work.
- **Part 2: Kotlin Coroutines library** - explaining the most important concepts from the kotlinx.coroutines library and how to use them well.
- **Part 3: Channel and Flow** - focused on Channel and Flow from the kotlinx.coroutines library.
- **Part 4: Kotlin Coroutines in practice** - reviewing common use cases and the most important best practices for using Kotlin Coroutines.

What will be covered?

This book is based on a workshop I conduct. During its iterations I have been able to observe what interested attendees and what did not. These are the elements that are most often mentioned by attendees:

- **How do coroutines really work?** (Part 1)
- **How to use coroutines in practice?** (Part 2, 3 and especially 4)
- **What are the best practices?** (Part 2, 3 and especially 4)
- **Testing Kotlin coroutines** (*Testing Kotlin Coroutines* in Part 2 and *Testing Flow* in Part 3)
- **What is Flow and how does it work?** (Part 3)

The Kotlin for Developers series

This book is a part of a series of books called *Kotlin for Developers*, which includes the following books:

- *Kotlin Essentials*, which covers all the basic Kotlin features.
- *Functional Kotlin*, which is dedicated to functional Kotlin features, including function types, lambda expressions, collection processing, DSLs, and scope functions.
- *Kotlin Coroutines: Deep Dive*, which covers all the Kotlin Coroutines features, including how to use and test them, using flow, the best practices, and the most common mistakes.
- *Advanced Kotlin*, which is dedicated to advanced Kotlin features, including generic variance modifiers, delegation, multiplatform programming, annotation processing, KSP and compiler plugins.
- *Effective Kotlin: Best Practices*, which is dedicated to the best practices of Kotlin programming.

Do not worry, you do not need to read the previous books in the series to understand this one. However, if you are interested in learning more about Kotlin, I recommend considering the other books from this series.

Conventions

When I use a concrete element from code, I will use code-font. When naming a concept, I will capitalize the word. To reference an arbitrary element of some type, I will use lowercase. For example:

- `Flow` is a type or an interface, it is printed with code-font (like in "Function needs to return `Flow`");
- Flow represents a concept, so it starts with an uppercase letter (like in "This explains the essential difference between Channel and Flow");
- a flow is an instance, like a list or a set, therefore it is in lowercase (like in "Every flow consists of a few elements").

Another example: `List` refers concretely to a list interface or type ("The type of `l` is `List`"), while List represents a concept ("This explains the essential difference between List and Set"), and a list is one of many lists ("The `list` variable holds a list").

I have used American English in the book, except for the spelling of "cancellation/cancelled", which I chose due to the spelling of the "Cancelled" coroutine state.

Code conventions

Most of the presented snippets are executable, so if you copy-paste them to a Kotlin file, you should be able to execute them. The source code of all the snippets is published in the following repository:

https://github.com/MarcinMoskala/coroutines_sources

Snippet results are presented using the `println` function. The result will often be placed at the end of these snippets in comments. If there is a delay between output lines, it will be shown in brackets. Here is an example:

```
suspend fun main(): Unit = coroutineScope {
  launch {
      delay(1000L)
      println("World!")
  }
  println("Hello,")
}
// Hello,
// (1 sec)
// World!
```

In some cases, I will show comments next to the line that prints them. I do this when the order is clear:

```
suspend fun main() {
  println("Hello,") // Hello,
  delay(1000L) // (1 sec)
  println("World!") // World!
}
```

Sometimes, some parts of code or results are shortened with . . . in a comment. In such cases, you can read it as "there should be more here, but the author decided to omit it".

```
launch(CoroutineName("Name1")) { /*...*/ }
launch(CoroutineName("Name2") + Job()) { /*...*/ }
```

In a few snippets I have added a number after the line to more easily explain the snippet's behavior. This is what it might look like:

```
suspend fun main() {
   println("Hello,") // 1
   delay(1000L) // 2
   println("World!") // 3
}
```

At 1 we print "Hello,", then we wait for a second because line 2 contains `delay`, and we print "World!" at line 3.

In some snippets, you might notice strange formatting. This is because the line length in this book is only 67 characters, so I adjusted the formatting to fit the page width.

Version

This is version 3.1 of the book. It is based on Kotlin 2.0.20 and kotlinx.coroutines 1.8.0 and kotlinx.coroutines-test 1.8.0. It was last updated in Jul 2024.

Exercises and solutions

At the end of most chapters, you will find exercises. They are designed to help you understand the material better. Starting code and unit tests for most of those

exercises can be found in the MarcinMoskala/kotlin-exercises project on GitHub. You can clone this project and solve these exercises locally. Solutions can be found at the end of the book.

Suggestions

If you have any suggestions or corrections regarding this book, send them to contact@kt.academy

Acknowledgments

This book would not be so good without the reviewers' great suggestions and comments. I would like to thank all of them. Here is the whole list of reviewers, starting from the most active ones.

Nicola Corti - a Google Developer Expert for Kotlin. He has been working with the language since before version 1.0, and he is the maintainer of several open-source libraries and tools for mobile developers (Detekt, Chucker, AppIntro). He's currently working in the React Native core team at Meta, building one of the most popular cross-platform mobile frameworks. Furthermore, he is an active member of the developer community. His involvement goes from speaking at international conferences to being a member of CFP committees and supporting developer communities across Europe. In his free time, he also loves baking, podcasting, and running.

Garima Jain - a Google Developer Expert in Android from India. She is also known around the community as @ragdroid. Garima works as a Principal Android Engineer at GoDaddy. She is also an international speaker and an active technical blogger. She enjoys interacting with other people from the community and sharing her thoughts with them. In her leisure time, she loves watching television shows, playing TT, and basketball. Due to her love for fiction and coding, she loves to mix technology with fiction and then shares her ideas with others through talks and blog posts.

Ilmir Usmanov - a software developer at JetBrains, working on coroutine support in the Kotlin compiler since 2017. Was responsible for stabilization and implementation of the coroutines design. Since then, he has moved to other features, namely inline classes. Currently, his work with coroutines is limited to bug fixing and optimization, since coroutines as a language feature is complete and stable and does not require much attention.

Sean McQuillan - a Developer Advocate at Google. With a decade of experience at Twilio and other San Francisco startups, he is an expert at building apps that scale. Sean is passionate about using great tooling to build high-quality apps quickly. When he is not working on Android, you can find him fiddling on the piano or crocheting hats.

Igor Wojda - a passionate engineer with over a decade of software development experience. He is deeply interested in Android application architecture and the Kotlin language, and he is an active member of the open-source community. Igor is a conference speaker, technical proofreader for the 'Kotlin In Action' book, and author of the 'Android Development with Kotlin' book. Igor enjoys sharing his passion for coding with other developers.

Jana Jarolimova - an Android developer at Avast. She started her career teaching Java classes at Prague City University, before moving on to mobile development, which inevitably led to Kotlin and her love thereof.

Richard Schielek - an experienced developer and an early adopter of Kotlin and coroutines, using both in production before they became stable. Worked in the European space industry for several years.

Vsevolod Tolstopyatov - a team lead of the Kotlin Libraries team. He works at JetBrains and is interested in API design, concurrency, JVM internals, performance tuning and methodologies.

Lukas Lechner, **Ibrahim Yilmaz**, **Dean Djermanović** and **Dan O'Neill**.

I would also like to thank **Michael Timberlake**, our language reviewer, for his excellent corrections to the whole book.

Part 1: Understanding Kotlin Coroutines

Before we begin our adventure with the Kotlin Coroutines library, let's start with some more basic concepts. What are coroutines? How does suspension work? What does it all look like under the hood? We will explore all this while learning some useful tools and practices.

Why Kotlin Coroutines?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Simplicity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Cancellation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Synchronization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Coroutines are multiplatform

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

The biggest problem with Kotlin Coroutines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Sequence builder

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Real-life usages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Factorial sequence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Prime numbers sequence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

How does suspension work?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

A game analogy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Suspending functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Your first suspension

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

What is stored in the continuation?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Delaying a coroutine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Resuming with a value

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Resume with an exception

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Suspending a coroutine, not a function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Callback function wrappers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Continuation storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Coroutines under the hood

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Continuation-passing style

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

A very simple function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

A function with a state

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

A function resumed with a value

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

The call stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Suspending functions in other contexts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

The actual code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

The performance of suspending functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: What is stored by a continuation?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Coroutines: built-in support vs library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Part 2: Kotlin Coroutines library

Now that we understand how built-in support works, it is time to concentrate on the kotlinx.coroutines library. In this part, we will learn everything that is needed to use it properly. We will explore coroutine builders, different coroutine contexts, and how cancellation works. We will finish with practical knowledge on how to set up coroutines, how to test them, and how to safely access a shared state.

Starting coroutines

Suspending functions are not coroutines; they are functions that can suspend a coroutine, which means they must be called from a coroutine. There are three main ways to start coroutines:

- Asynchronous coroutine builders (`launch` and `async`), which start an asynchronous coroutine on a scope.
- Blocking coroutine builders (`runBlocking` and `runTest`), which start a coroutine on the current thread and block it until the coroutine is done.
- Coroutine scope functions, which create a synchronous coroutine; in practice, this means that coroutine scope functions suspend the current coroutine until the new one is completed.

Each of those groups of methods has completely different use cases. In this chapter, we will learn about them and how we use them in practice.

Asynchronous coroutine builders

To start an asynchronous coroutine, we use the `launch` and `async` functions, both of which are extension functions on `CoroutineScope`, which for now can be seen as a "configuration and parent of a coroutine". We will learn about constructing custom scopes later in this book, but for now we will use the `GlobalScope` object as a placeholder for a scope. Once we learn how to construct a scope, we will stop using `GlobalScope` as it is considered an anti-pattern.

So let's start with an example of how `launch` works. It might remind you of the `thread` function as it starts a new asynchronous process. The execution of `launch` does not wait for the coroutine to finish, so it is immediately followed by the next line of code. This is why "Hello" in the code below is printed immediately, and "World!" is printed three times after a second.

```
fun main() {
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    Thread.sleep(2000L)
}
// Hello,
// (1 sec)
// World!
// World!
// World!
```

One thing you might have noticed is that we need to call `Thread.sleep` at the end of the `main` function. Without doing this, this function would end immediately after launching the coroutines, so these coroutines wouldn't have a chance to do their job. This is because `delay` does not block the thread: it suspends a coroutine. You might remember from the *How does suspension work?* chapter that `delay` just sets a timer to resume after a set amount of time and suspends a coroutine until then. If the thread is not blocked, nothing stops the program from finishing. So, we need to block the thread to wait for the coroutines to finish. Is there any better way to do this? Yes, there is. Each `launch` function returns a `Job` object that can be used to await coroutine completion using the `join` function, which suspends until a coroutine is completed. If we add a `suspend` modifier to our `main` function, we can use `join` to explicitly await completion of all asynchronous coroutines.

```
suspend fun main() {
    val job1 = GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    val job2 = GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
```

```
    val job3 = GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    job1.join()
    job2.join()
    job3.join()
}
// Hello,
// (1 sec)
// World!
// World!
// World!
```

The `launch` builder does not return any value, and its lambda expression returns `Unit`. This is the key difference between the `launch` and `async` builders. The `async` builder behaves just like `launch`, but its lambda expression is expected to return a value[2]. As a result, `async` builder returns a `Deferred<T>` object that also implements `Job` but additionally has an `await` function that is used to wait and get the result of the coroutine. So `await`, just like `join`, suspends until the coroutine is completed, but it also returns the result of the coroutine, or an exception if it has thrown one.

```
suspend fun main() {
    val value1 = GlobalScope.async {
        delay(2000L)
        1
    }
    val value2 = GlobalScope.async {
        delay(2000L)
        2
    }
    val value3 = GlobalScope.async {
```

[2]Generally speaking, `async` could be used to replace all `launch` usages, but it is not recommended because `async` is supposed to be used to return a value, so it is better to use `launch` when we do not need a value. Also, there are minor differences between how some features treat `launch` and `async`, for instance `CoroutineExceptionHandler` assumes `async` catches all its exceptions and represents them in the resulting `Deferred` object, so exceptions that break `async` are not considered uncaught, while exceptions that break `launch` are considered uncaught. In general, is better to use `launch` when we do not need a value, especially when we start a new hierarchy of coroutines.

```
        delay(2000L)
        3
    }
    println("Calculating")
    print(value1.await())
    print(value2.await())
    print(value3.await())
}
// Calculating
// (2 sec)
// 123 (order is guaranteed, as we await for values in orde\
r)
```

It is important to note that the `launch` and `async` builders start a coroutine immediately. Both `join` and `await` only await the completion of those coroutines. This is why "Calculating" is printed immediately in the example above, and then we wait only 2 seconds to see all the values.

If calculating `value1`, `value2`, and `value3` took 3 seconds, 2 seconds, and 1 second, respectively, then we would see "123" after 3 seconds. This is because we would wait for the first value to be ready for 3 seconds, and the other two would be calculated in the meantime.

If calculating `value1`, `value2` and `value3` took 1 second, 2 seconds, and 3 seconds, respectively, then we would see "1" after 1 second, "2" after another 1 second, and "3" after another second. This is because we wait for values in order.

If we call `await` on an already completed `Deferred`, then there is no suspending; `await` just returns the value. This is why calling `await` multiple times in succession does not suspend multiple times.

```
suspend fun main() {
    val value = GlobalScope.async {
        delay(2000L)
        1
    }
    println("Calculating")
    print(value.await())
    print(value.await())
    print(value.await())
}
// Calculating
// (2 sec)
// 111
```

The `async` builder is often used to asynchronously execute two processes and then combine their results.

```
scope.launch {
    val news = async { newsRepo.getNews() }
    val newsSummary = async { newsRepo.getNewsSummary() }
    view.showNews(newsSummary.await(), news.await())
}
```

Blocking coroutine builders

The general rule is that coroutines should never block threads, only suspend them. On the other hand, there are cases in which blocking is necessary, such as in the main function, where we need to block the thread otherwise our program will end too early. For such cases, we might use a blocking coroutine builder like `runBlocking`.

`runBlocking` starts a coroutine and blocks the current thread until this coroutine is completed[3]. Consider the code below. When the first `runBlocking` is called, it starts a coroutine and blocks the thread until it is done. This is why we first wait 1 second, and then we see "World!". Then we call the second `runBlocking`, which blocks the thread until its coroutine is done. So, we wait another second, and then we see "World!" again. Then we call the third `runBlocking`, so again we wait 1 second, and then we see "World!" again. Only then do we see "Hello" printed.

```
fun main() {
    runBlocking {
        delay(1000L)
        println("World!")
    }
    runBlocking {
        delay(1000L)
        println("World!")
    }
    runBlocking {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
// (1 sec)
// World!
// (1 sec)
// World!
```

[3] This thread is not completely blocked because `runBlocking` uses it by default to execute the coroutine, but it is blocked from executing other code.

```
// (1 sec)
// World!
// Hello,
```

This means that `runBlocking` starts a coroutine but is effectively synchronous because its execution takes as long as the coroutine needs to complete.

There are a couple of specific use cases in which `runBlocking` is used. The first is the main function, where we must block the thread because otherwise the program will end. Another common use case is unit tests, where we need to block the thread for the same reason.

```
fun main() = runBlocking {
    // ...
}

class MyTests {

    @Test
    fun `a test`() = runBlocking {

    }
}
```

`runBlocking` used to be an important builder, but it is used rather rarely in modern programming. In unit tests, we often use its successor, `runTest`, which makes coroutines operate in virtual time (a powerful feature for testing that we will describe in the *Testing coroutines* chapter). For the `main` function, we nowadays often just add the `suspend` modifier instead of using `runBlocking`. `runBlocking` is used only in places in our program where we do want to block the thread, such as when we need a regular function to call suspending functions and await their completion.

```
fun runDataMigrationScript() = runBlocking {
    val sourceData = readDataFromSource()
    val transformedData = transformData(sourceData)
    writeDataToTarget(transformedData)
}
```

Structured Concurrency

If you see the definition of `runBlocking`, you might notice that its lambda expression has a receiver of type `CoroutineScope`. The same can be said about the `launch` and `async` builders, which are also extension functions on `CoroutineScope`.

```
fun <T> runBlocking(
    context: CoroutineContext,
    block: suspend CoroutineScope.() -> T
): T

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

fun <T> CoroutineScope.async(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T>
```

This scope is used to implement one of the most important concepts in the kotlinx.coroutines library: *structured concurrency*. The key idea is that when a coroutine is started on a scope, it becomes a child of this scope. Each coroutine builder provides a scope that can be used to start other coroutine builders on it. As a result, these coroutines become children of the coroutine that provided their scope. This creates a hierarchy of coroutines, where each coroutine has a parent and can have children, such as in the example below, where `runBlocking` is a parent of three `launch` coroutines because they are all started implicitly on its scope (on the scope of `runBlocking`, because we call `launch` on implicit `this`).

```
fun main() = runBlocking {
    launch { // same as this.launch
        delay(1000L)
        println("World!")
    }
    launch { // same as this.launch
        delay(1000L)
        println("World!")
    }
    launch { // same as this.launch
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
// Hello,
// (1 sec)
```

```
// World!
// World!
// World!
```

The parent-child relationship has a couple of important consequences:

1. Children inherit context from their parent (but they can also overwrite it, as will be explained in the *Coroutine context* chapter).
2. A parent cannot complete until all its children have completed (this will be explained in the *Job and coroutine lifecycle* chapter).
3. When the parent is cancelled, its child coroutines are cancelled too (this will be explained in the *Cancellation* chapter).
4. When a child completes with an exception, this exception is passed to the parent (this will be explained in the *Exception handling* chapter).

In the above example, we can see "World!" printed three times after 1 second. We would not see it if `runBlocking` were not a parent of these coroutines because then it would not wait for them to finish. However, when `runBlocking` is a parent, it must wait for all its children to complete so we can be sure that our program will end after all the coroutines have completed.

Starting coroutines on `GlobalScope` would break the structured concurrency because there would be no relationship between `runBlocking` and the coroutines started on `GlobalScope`.

```
fun main() = runBlocking {
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
// Hello,
```

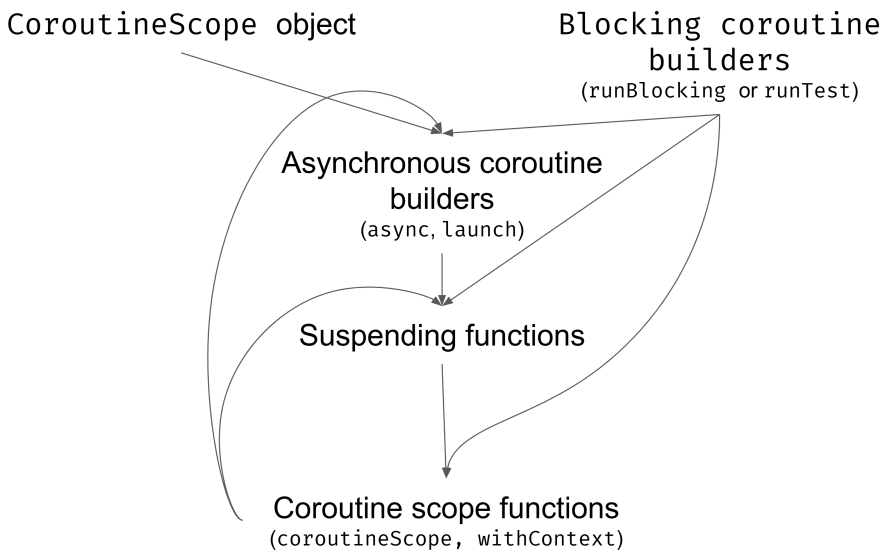
`GlobalScope` is just a placeholder for the lack of a parent. It is literally an empty scope that configures nothing and builds no relationship with coroutines started on it, therefore it is considered bad practice to use it because it can easily break our relationships and it cannot be used to control coroutines started on it.

```
object GlobalScope : CoroutineScope {
    override val coroutineContext: CoroutineContext
        get() = EmptyCoroutineContext
}
```

`runBlocking` is not an extension function on `CoroutineScope`, therefore it cannot be a child: it can only be used as a root coroutine (the parent of all the children in a hierarchy).

Suspending functions need to be called from other suspending functions, but this needs to start with a coroutine builder such as `runBlocking`, which starts a hierarchy but is rarely used in practice since it blocks a thread. Instead, we use the `launch` and `async` builders, both of which start asynchronous coroutines. However, they need a scope, so we need to create one (we will explain how to do this in the *Constructing coroutine scope* chapter).

So, in real-life applications, everything starts with the scope in which asynchronous coroutines are started. We can call other builders from these coroutines, but we primarily call suspending functions; these can call other suspending functions as well as other functions, but what if we want to start new coroutines from suspending functions? We cannot use the `launch` or `async` builders because they need a scope. We cannot use `runBlocking` either because it would block the thread. We don't want to use `GlobalScope` or any other external scope because this would break the structured concurrency. This is where we need to use coroutine scope functions that start a synchronous coroutine and provide a scope that can be used to start asynchronous coroutines. These functions are essential to maintain structured concurrency, and they allow us to see the whole picture of how coroutines are used in practice.



A diagram showing how different kinds of elements of the kotlinx.coroutines library are used. We generally start with a scope that is used by asynchronous coroutine builders, which call suspending functions. Suspending functions call other suspending functions but cannot directly call asynchronous coroutine builders. Whenever they need to start an asynchronous process, they use coroutine scope functions to create a scope that can be used to start asynchronous coroutines.

Let's discuss how coroutine scope functions, the last piece of this puzzle, work.

Coroutine scope functions

Coroutine scope functions[4] are suspending functions that start a synchronous coroutine. To be more precise, they start a new coroutine but suspend the current one until the new one is completed. In practice, they behave a lot like `runBlocking`, but instead of blocking the thread they suspend the coroutine. The most basic representative of this group is `coroutineScope`.

[4]Some people call them "scoping functions", but I find this confusing as I am not sure what is meant by "scoping". I guess that whoever started using this term just wanted to make it different from "scope functions" (functions like `let`, `with` or `apply`), but it is not really helpful as these two terms are still often confused. This is why I decided to use the term "coroutine scope functions", which is longer but should cause fewer misunderstandings, and I find it more correct. Also, some people call them "suspending coroutine builders", which is a pretty good name but makes them hard to distinguish from asynchronous coroutine builders, and they both serve completely different purposes. "Coroutine scope functions" is a clear and descriptive name, so I will use it in this book.

```
suspend fun main() {
    coroutineScope {
        delay(1000L)
        println("World!")
    }
    coroutineScope {
        delay(1000L)
        println("World!")
    }
    coroutineScope {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
// (1 sec)
// World!
// (1 sec)
// World!
// (1 sec)
// World!
// Hello,
```

Coroutine scope functions return whatever is returned from their lambda expression.

```
suspend fun main() {
    println("A")
    val a: Int = coroutineScope {
        delay(1000L)
        10
    }
    println("B")
    val b: Int = coroutineScope {
        delay(1000L)
        20
    }
    println("C")
    println(a + b)
}
// A
// (1 sec)
// B
// (1 sec)
```

```
// C
// 30
```

I want you to make sure that you understand this. Coroutine scope functions have very little to do with asynchronous coroutine builders. They both start coroutines, but coroutine scope functions are synchronous coroutines because they suspend the current coroutine until the coroutine they've started has completed. Coroutine scope functions are suspending functions that do not require a scope, while coroutine builders are regular functions that do require a scope. If you wanted to use `async` instead of `coroutineScope` in the above example, you would need to pass a scope to it, so you would need to use a coroutine scope function anyway. The result from `async` would be `Deferred<Int>`, so you would need to call `await` on it to get the result. Finally, delays would happen after "A", "B", and "C" are printed.

```
suspend fun main() = coroutineScope {
    println("A")
    val a: Deferred<Int> = async {
        delay(1000L)
        10
    }
    println("B")
    val b: Deferred<Int> = async {
        delay(1000L)
        20
    }
    println("C")
    println(a.await() + b.await())
}
// A
// B
// C
// (2 sec)
// 30
```

In practice, coroutine scope functions are used to create a scope for asynchronous coroutines. For example, imagine that in some service function you need to asynchronously load two resources, such as user data and a list of articles. In this case, you want to return only those articles that should be seen by the user. To call `async`, you need a scope, which you can create using a coroutine scope function.

```
suspend fun getUserProfile(
    userId: String
): UserProfile = coroutineScope {
    val user = async { getPublicUserDetails(userId) }
    val articles = async { getUserArticles(userId) }
    UserProfile(user.await(), articles.await())
}
```

Coroutine builders can be started in lambda expressions. We can, for instance, call `async` inside `map` to start an asynchronous coroutine for each element of a collection. To await a list of differed values, we can use the `awaitAll` function, which suspends until all the results are ready.

```
suspend fun getPublicUserDetails(
    userId: String
): List<ArticleDetails> = coroutineScope {
    articleRepo.getArticles(userId)
        .filter { it.isPublic }
        .map { async { getArticleDetails(it.id) } }
        .awaitAll()
}
```

It is important to understand that coroutine scope functions maintain structured concurrency. In both the examples above, the coroutines started by `async` are direct children of the coroutine started by `coroutineScope`, which is a direct child of the coroutine that called `getPublicUserDetails`. Structured concurrency is maintained with all its consequences. Coroutine scope functions:

1. inherit context from their parent,
2. cannot complete until all their children are completed,
3. cancel their children when they get cancelled,
4. cancel and throw an exception when they receive an exception from a child (this will be explained in greater detail in the *Exception handling* chapter).

I want you to consider the consequences of the second rule. Since coroutine scope functions cannot complete until all their children are completed, a suspend function must await all the coroutines started on it. Consider the next example. We can be sure that "After" will be printed when all the coroutines started on `longTask` have completed because `coroutineScope` must await the completion of all its children.

```
suspend fun longTask() = coroutineScope {
    launch {
        delay(1000)
        println("Finished task 1")
    }
    launch {
        delay(2000)
        println("Finished task 2")
    }
}

suspend fun main() {
    println("Before")
    longTask()
    println("After")
}
// Before
// (1 sec)
// Finished task 1
// (1 sec)
// Finished task 2
// After
```

This is a general rule for all suspending functions: they must await the completion of all the coroutines started on them. The only way to start a process that is not awaited is to start it on an external scope, but this breaks structured concurrency.

Also note that using `launch` in the last line of a coroutine scope function makes no sense[5]. It will be awaited anyway, so using `launch` is only confusing and we could just inline its body. If we wanted to start a coroutine that is not awaited, we would need to start it on an external scope.

```
suspend fun updateUser() = coroutineScope {
    // ...

    // Don't
    launch { sendEvent(UserSunchronized) }
    // should be (to call synchronously)
    // sendEvent(UserSunchronized)
    // or (to call asynchronously), if we have scope proper\
ty
```

[5]It only makes a difference in terms of exception handling, when used in `supervisorScope`, what we will explain in the *Exception handling* chapter.

```
    // scope.launch { sendEvent(UserSunchronized) }
}
```

`coroutineScope` is the most basic coroutine scope function. All the others behave like `coroutineScope`, but they have some additional features:

- `withContext` behaves just like `coroutineScope` but can change the context of the coroutine, so we will explain it in the *Coroutine context* section.
- `supervisorScope` behaves just like `coroutineScope` but ignores its children's exceptions, so we will explain it in the *Exception handling* section.
- `withTimeout` behaves just like `coroutineScope` but cancels the coroutine after a timeout, so we will explain it in the *Cancellation* section.

These features are important for suspending functions, as we will learn in the next chapters.

Summary

In this chapter, we learned about the three main ways of starting coroutines:

- Asynchronous coroutine builders (`launch` and `async`), which start asynchronous coroutines on a scope.
- Blocking coroutine builders (`runBlocking` and `runTest`), which start a new coroutine and run it on the current thread, blocking this thread until the coroutine is done.
- Coroutine scope functions, which create a synchronous coroutine, therefore they are used to create a scope that can be used to start asynchronous coroutines.

This knowledge is enough for most uses of Kotlin coroutines. In most cases, we just have suspended functions calling other suspending or normal functions. If we need to introduce concurrent processing, we wrap a function with `coroutineScope` and use builders on its scope. Everything needs to start with some builders called on some scope. Later, we will learn how to construct such a scope, but for most projects it needs to be defined once and is touched rarely.

Even though we have learned the essentials, there is still much more. In the next chapters, we will dive deeper into coroutines: we will learn to use different contexts, how to tame cancellations and exception handling, how to test coroutines, etc. There are still a lot of great features to discover.

Exercise: UserDetailsRepository

In the `UserDetailsRepository` class, implement the `getUserDetails` function, which return details of the current user. It should:
- Check if the user is available in the database. If so, it should return this data.
- If the user is not available in the database, it should asynchronously fetch the user details from the client, use them to create a `UserDetails` object, and return it.
- After fetching new user data, it should asynchronously save it in the database. `getUserDetails` function should be able to complete its execution without waiting for the data to be saved.

Starting code:

```
class UserDetailsRepository(
    private val client: UserDataClient,
    private val userDatabase: UserDetailsDatabase,
    private val backgroundScope: CoroutineScope,
) {
    suspend fun getUserDetails(): UserDetails {
        TODO()
    }
}

interface UserDataClient {
    suspend fun getName(): String
    suspend fun getFriends(): List<Friend>
    suspend fun getProfile(): Profile
}

interface UserDetailsDatabase {
    suspend fun load(): UserDetails?
    suspend fun save(user: UserDetails)
}

data class UserDetails(
    val name: String,
    val friends: List<Friend>,
    val profile: Profile
)

data class Friend(val id: String)
data class Profile(val description: String)
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/starting/UserDetailsRepository.kt`. You can clone this project and solve this exercise locally.

Exercise: BestStudentUseCase

In the `BestStudentUseCase` class, implement the `getBestStudent` function, which fetches all students from the given semester and then finds the one with the best result. For this, it first needs to use `StudentsRepository` to find the IDs of students in the semester, after which it needs to fetch these users' details. Fetching details should be done asynchronously. The best user is the one with the highest `result` value. If there are no students in the given semester, the function should throw `IllegalStateException`.

```
class BestStudentUseCase(
    private val repo: StudentsRepository
) {
    suspend fun getBestStudent(semester: String): Student =\
 TODO()
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/starting/BestStudent.kt`. You can clone this project and solve this exercise locally.

Exercise: CommentService

Implement the following functions from the `CommentService` class:

- `addComment` - should read a user id from a token, transform the body to a `CommentDocument`, and add it to the comment repository.
- `getComments` - should get comments with users and return them as a collection.

```
class CommentService(
    private val commentRepository: CommentRepository,
    private val userService: UserService,
    private val commentFactory: CommentFactory
) {
    suspend fun addComment(
        token: String,
        collectionKey: String,
        body: AddComment
    ) {
        TODO()
    }

    suspend fun getComments(
        collectionKey: String
    ): CommentsCollection = TODO()
}
```

Starting code can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/comment/CommentService.kt`. You can clone this project and solve this exercise locally.

You can assume that `findUserById` from `userService` can be called multiple times for the same user id because it is cached. Alternatively, you can refactor this service to make sure it is not called more than once for the same id by the same `getComments` call. The second option is more complex.

Exercise: mapAsync

Practice shows that many projects require asynchronous mapping of elements in a collection. To avoid repeating this pattern, implement an `mapAsync` function, which should map all elements in a collection asynchronously.

```
suspend fun <T, R> Iterable<T>.mapAsync(
    transformation: suspend (T) -> R
): List<R> = TODO()
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/recipes/MapAsync.kt`. You can clone this project and solve this exercise locally.

The bigger picture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Coroutine context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

`CoroutineContext` interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Finding elements in CoroutineContext

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Adding contexts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Empty coroutine context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Subtracting elements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Folding context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Coroutine context and builders

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Accessing context in a suspending function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Changing context in suspending functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Creating our own context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Coroutines and thread elements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Understanding context propagation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: CounterContext

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Dispatchers

The Kotlin Coroutines library offers an important functionality that lets us decide which thread (or pool of threads) a coroutine should be running on (starting and resuming). This is done using dispatchers.

In the English dictionary, a dispatcher is defined as "a person who is responsible for sending people or vehicles to where they are needed, especially emergency vehicles". In Kotlin coroutines, `CoroutineContext` determines which thread a certain coroutine will run on.

> Dispatchers in Kotlin Coroutines are a similar concept to RxJava Schedulers.

Default dispatcher

If you don't set any dispatcher, the one chosen by default by asynchronous coroutine builders is `Dispatchers.Default`, which is designed to run CPU-intensive operations. It has a pool of threads whose size is equal to the number of cores in the machine your code is running on (but not less than two). At least theoretically, this is the optimal number of threads, assuming you are using them efficiently, i.e., performing CPU-intensive calculations and not blocking threads. To see this dispatcher in action, run the following code:

```
suspend fun main() = coroutineScope {
    repeat(1000) {
        launch { // or launch(Dispatchers.Default) {
            // To make it busy
            List(1_000_000) { Random.nextLong() }.maxOrNull\
()

            val threadName = Thread.currentThread().name
            println("Running on thread: $threadName")
        }
    }
}
```

Example result on my machine (I have 12 cores, so there are 12 threads in the pool):

```
Running on thread: DefaultDispatcher-worker-1
Running on thread: DefaultDispatcher-worker-5
Running on thread: DefaultDispatcher-worker-7
Running on thread: DefaultDispatcher-worker-6
Running on thread: DefaultDispatcher-worker-11
Running on thread: DefaultDispatcher-worker-2
Running on thread: DefaultDispatcher-worker-10
Running on thread: DefaultDispatcher-worker-4
...
```

> Warning: `runBlocking` sets its own dispatcher if no other one is set; so, inside its scope the `Dispatcher.Default` is not the one that is chosen automatically. If we used `runBlocking` instead of `coroutineScope` in the above example, all coroutines would be running on "main".

Limiting the default dispatcher

Let's say that you have an expensive process and you suspect that it might use all `Dispatchers.Default` threads and starve other coroutines using the same dispatcher. In such cases, we can use `limitedParallelism` on `Dispatchers.Default` to make a dispatcher that runs on the same threads but is limited to using not more than a certain number of them at the same time.

```
private val dispatcher = Dispatchers.Default
    .limitedParallelism(5)
```

If you've seen `limitedParallelism` before, I should warn you that this function has completely different behavior for `Dispatchers.Default` than for `Dispatchers.IO`. We will discuss it later.

> `limitedParallelism` was introduced in kotlinx-coroutines version `1.6`, so it is quite a new feature and you won't find it being used in older projects.

Main dispatcher

Android and many other application frameworks have the concept of a main or UI thread, which is generally the most important thread as it is the only one that can be used to interact with the UI on Android. Therefore, it needs to be used very often but also with great care. When the Main thread is blocked, the whole application is frozen. To run a coroutine on the Main thread, we use `Dispatchers.Main`.

```
suspend fun showUserName(name: String) =
    withContext(Dispatchers.Main) {
        userNameTextView.text = name
    }
```

`Dispatchers.Main` is available on Android if we use the `kotlinx-coroutines-android` artifact. Similarly, it's available on JavaFX if we use `kotlinx-coroutines-javafx`, and on Swing if we use `kotlinx-coroutines-swing`. If you do not have a dependency that defines the main dispatcher, it is not available and cannot be used.

Notice that frontend libraries are typically not used in unit tests, so `Dispatchers.Main` is not defined there. To be able to use it, you need to set a dispatcher using `Dispatchers.setMain(dispatcher)` from `kotlinx-coroutines-test`.

```
class SomeTest {

    private val dispatcher = Executors
        .newSingleThreadExecutor()
        .asCoroutineDispatcher()

    @Before
    fun setup() {
        Dispatchers.setMain(dispatcher)
    }

    @After
    fun tearDown() {
        // reset the Main dispatcher to
        // the original Main dispatcher
        Dispatchers.resetMain()
        dispatcher.close()
    }

    @Test
    fun testSomeUI() = runBlocking {
        launch(Dispatchers.Main) {
            // ...
        }
    }
}
```

On Android, we typically use the Main dispatcher as the default one. If you use libraries that are suspending instead of blocking and you don't do any complex

calculations, in practice you can only use `Dispatchers.Main`. If you do some CPU-intensive operations, you should run them on `Dispatchers.Default`. These two are enough for many applications, but what if you need to block the thread because, for example, you need to perform long I/O operations (e.g., read big files) or use a library with blocking functions? You cannot block the Main thread because your application would freeze. If you block your default dispatcher, you risk blocking all the threads in the thread pool, in which case you won't be able to do any calculations. This is why we need a different dispatcher for such a situation, and this dispatcher is `Dispatchers.IO`.

IO dispatcher

`Dispatchers.IO` is designed to be used when we block threads with I/O operations, such as when we read/write files or call blocking functions. The code below takes around 1 second because `Dispatchers.IO` allows more than 50 active threads at the same time.

```
suspend fun main() {
    val time = measureTimeMillis {
        coroutineScope {
            repeat(50) {
                launch(Dispatchers.IO) {
                    Thread.sleep(1000)
                }
            }
        }
    }
    println(time) // ~1000
}
```

> `Dispatchers.IO` is only needed if you have an API that blocks threads. If you use suspending functions, you can use any dispatcher. You do not need to use `Dispatchers.IO` if you want to use a network or database library that provides suspending functions. In many projects, this means you might not need to use `Dispatchers.IO` at all.

How does IO dispatcher work? Imagine an unlimited pool of threads that is initially empty but more threads are created as we need them and kept active until they are not used for some time. Such a pool exists, but it wouldn't be safe to use it directly because too many active threads cause performance to degrade in a slow but unlimited manner, eventually causing out-of-memory errors. This is why all basic dispatchers have a limited number of threads they can use at the same time. `Dispatchers.Default` is limited by the number of cores in your processor. The limit of `Dispatchers.IO` is 64 (or the number of cores if there are more).

```
suspend fun main() = coroutineScope {
    repeat(1000) {
        launch(Dispatchers.IO) {
            Thread.sleep(200)

            val threadName = Thread.currentThread().name
            println("Running on thread: $threadName")
        }
    }
}
// Running on thread: DefaultDispatcher-worker-1
//...
// Running on thread: DefaultDispatcher-worker-53
// Running on thread: DefaultDispatcher-worker-14
```

As we have mentioned, both `Dispatchers.Default` and `Dispatchers.IO` share the same pool of threads. This is an important optimization. Threads are reused, and redispatching is often not needed. For instance, let's say you are running on `Dispatchers.Default` and then execution reaches `withContext(Dispatchers.IO) { ... }`. Most often, you will stay on the same thread[6], but what changes is that this thread counts towards not the `Dispatchers.Default` limit but the `Dispatchers.IO` limit. Their limits are independent, so they will never starve each other.

```
suspend fun main(): Unit = coroutineScope {
    launch(Dispatchers.Default) {
        println(Thread.currentThread().name)
        withContext(Dispatchers.IO) {
            println(Thread.currentThread().name)
        }
    }
}
// DefaultDispatcher-worker-2
// DefaultDispatcher-worker-2
```

To see this more clearly, imagine that you use both `Dispatchers.Default` and `Dispatchers.IO` to the maximum. As a result, your number of active threads will be the sum of their limits. If you allow 64 threads in `Dispatchers.IO` and you have 8 cores, you will have 72 active threads in the shared pool. This means we have efficient thread reuse and both dispatchers have strong independence.

The most typical case in which we use `Dispatchers.IO` is when we need to call blocking functions from libraries. The best practice is to wrap them with

[6] This mechanism is not deterministic.

`withContext(Dispatchers.IO)` to make them suspending functions, which can be used without any special care: they can be treated like all other properly implemented suspending functions.

```
class DiscUserRepository(
    private val discReader: DiscReader
) : UserRepository {
    override suspend fun getUser(): UserData =
        withContext(Dispatchers.IO) {
            UserData(discReader.read("userName"))
        }
}
```

To better understand why `Dispatchers.IO` must have a limit, imagine that you have a periodic task that needs to start a large number of coroutines, each of which needs to block a thread. Your task might be sending a newsletter using a blocking API, like SendGrid. If `Dispatchers.IO` had no limit, this process would start as many threads as there are emails to send: if you have 100,000 emails to send, it will try to start 100,000 threads, which would require 100 GB of RAM, so it would crash your application. This is why we need to set a limit for `Dispatchers.IO`.

```
class NewsletterService {
    private val sendGrid = SendGrid(API_KEY)

    suspend fun sendNewsletter(
        newsletter: Newsletter,
        emails: List<Email>
    ) = withContext(Dispatchers.IO) {
        emails.forEach { email ->
            launch {
                sendGrid.api(createNewsletter(email, newsle\
tter))
            }
        }
    }

    // ...
}
```

This limit protects our resources but also makes processes take longer. If sending each email takes on average 100 ms and we have 100,000 emails to send, with `Dispatchers.IO` limited to 64 threads it will take nearly 3 minutes to send all

emails. The problem with `Dispatchers.IO` is that it has one limit for the whole application, so one service might block another. Imagine that in the same application you have another service that needs to send registration confirmation emails. If both services used `Dispatchers.IO`, then users trying to register would wait in a queue for threads until all newsletter emails have been sent. This should never happen, so we need to create dispatchers with custom independent limits.

Dispatcher with a custom limit

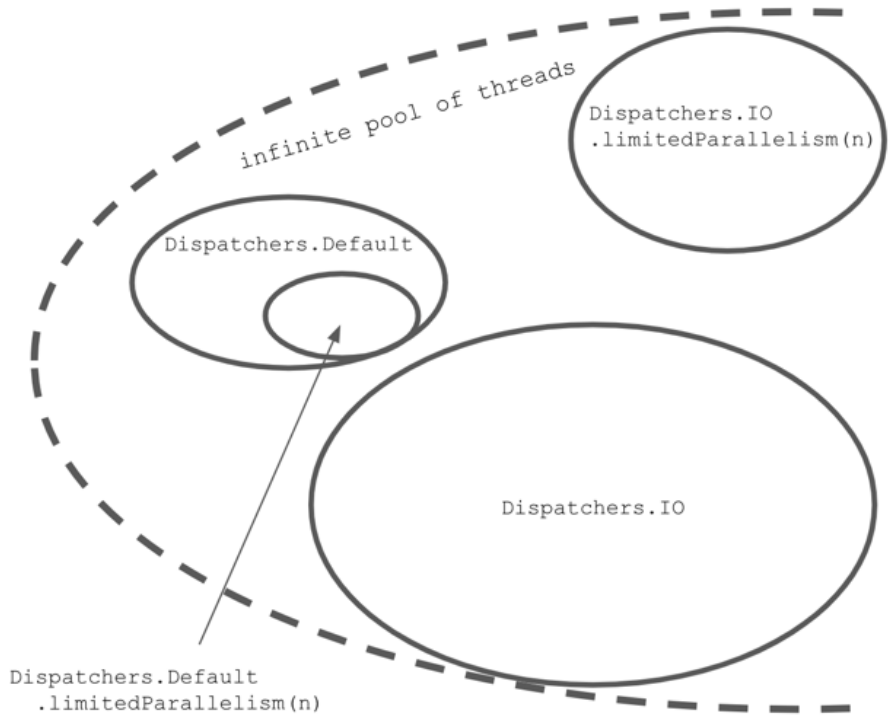
`Dispatchers.IO` has a special behavior defined for the `limitedParallelism` function that creates a new dispatcher with an independent thread limit. For example, imagine you start 100 coroutines, each of which blocks a thread for a second. If you run these coroutines on `Dispatchers.IO`, it will take 2 seconds. If you run them on `Dispatchers.IO` with `limitedParallelism` set to 100 threads, it will take 1 second. The execution time of both dispatchers can be measured at the same time because the limits of these two dispatchers are independent anyway.

```
suspend fun main(): Unit = coroutineScope {
    launch {
        printCoroutinesTime(Dispatchers.IO)
        // Dispatchers.IO took: 2074
    }

    launch {
        val dispatcher = Dispatchers.IO
            .limitedParallelism(100)
        printCoroutinesTime(dispatcher)
        // LimitedDispatcher@XXX took: 1082
    }
}

suspend fun printCoroutinesTime(
    dispatcher: CoroutineDispatcher
) {
    val test = measureTimeMillis {
        coroutineScope {
            repeat(100) {
                launch(dispatcher) {
                    Thread.sleep(1000)
                }
            }
        }
    }
```

```
    println("$dispatcher took: $test")
}
```

Conceptually, there is an unlimited pool of threads that is used by `Dispatchers.Default` and `Dispatchers.IO`, but they both have limited access to these threads. When we use `limitedParallelism` on `Dispatchers.IO`, we create a new dispatcher with an independent pool of threads (completely independent of `Dispatchers.IO` limit). If we use `limitedParallelism` on `Dispatchers.Default` or any other dispatcher, we create a dispatcher with an additional limit that is still limited, just like the original dispatcher.

```
// Dispatcher with an unlimited pool of threads
private val pool = ...
Dispatchers.IO = pool limited to 64
Dispatchers.IO.limitedParallelism(x) = pool limited to x
Dispatchers.Default = pool limited to coresNum
Dispatchers.Default.limitedParallelism(x) =
    Dispatchers.Default limited to x
```



Dispatchers.Default is limited to the number of cores. Dispatchers.IO is limited to 64 (or the number of cores). Using limitedParallelism on Dispatchers.Default makes a dispatcher with an additional limit to Dispatchers.Default, whereas using it on Dispatcher.IO makes a dispatcher with a limit independent of Dispatcher.IO. However, all these dispatchers share the same infinite pool of threads.

Many developers find it a bit confusing that `limitedParallelism` has different behavior for `Dispatchers.IO`, which has nothing to do with the `Dispatchers.IO` limit. I think we can make this much more intuitive by just adding a simple function to name the creation of a dispatcher that has an independent thread limit:

```
fun limitedDispatcher(threadLimit: Int) = Dispatchers.IO
    .limitedParallelism(threadLimit)
```

The best practice for classes that might intensively block threads is to define their own dispatchers that have their own independent limits. How big should this limit be? You need to decide for yourself, but remember that many threads are inefficient use of our resources. On the other hand, waiting for an available thread is not good for performance. What is most essential is that this limit is independent of `Dispatcher.IO` and other dispatchers' limits, therefore one service will not block another.

```
class DiscUserRepository(
    private val discReader: DiscReader
) : UserRepository {
    private val dispatcher = Dispatchers.IO
        .limitParallelism(5)

    override suspend fun getUser(): UserData =
        withContext(dispatcher) {
            UserData(discReader.read("userName"))
        }
}
```

Let's get back to the example with the newsletter and the registration confirmation emails. Each of these services should have its own dispatcher with a limit. This way, they will not block each other. The question is what the size of their limits should be. This is a hard question that has no simple answer. It depends on what we care about more: performance or resource usage. If we care about performance, we should set the limit to the number of threads that will be used most of the time. If we care about resource usage, we should set the limit to the number of threads that will be used at the peak time. In this case, I don't care that the newsletters take a bit longer to send; it would be perfectly fine even if it took an hour, so I could use a small limit, like 5. On the other hand, I care about the registration confirmation emails being sent as soon as possible, and I don't think this service will ever be used too intensively, so I could set a much higher limit, like 50. Even in most extreme cases, I do not want to block too many threads, because it would expose us to attacks (a big number of requests could block enough threads to cause out-of-memory errors).

```
class NewsletterService {
    private val dispatcher = Dispatchers.IO.limitedParallel\
ism(5)
    private val sendGrid = SendGrid(API_KEY)

    suspend fun sendNewsletter(
        newsletter: Newsletter,
        emails: List<Email>
    ) = withContext(dispatcher) {
        emails.forEach { email ->
            launch {
                sendGrid.api(createNewsletter(email, newsle\
tter))
            }
        }
    }
```

```
    // ...
}

class AuthorizationService {
    private val dispatcher = Dispatchers.IO.limitedParallel\
ism(50)

    suspend fun sendAuthEmail(
        user: User
    ) = withContext(dispatcher) {
        sendGrid.api(createConfirmationEmail(user))
    }

    // ...
}
```

Dispatcher with a fixed pool of threads

Some developers like to have more control over the pools of threads they use, and Java offers a powerful API for that. For example, on JVM we can create a fixed or cached pool of threads with the `Executors` class. These pools implement the `ExecutorService` or `Executor` interfaces, which we can transform into a dispatcher using the `asCoroutineDispatcher` function.

```
private val NUMBER_OF_THREADS = 20
val dispatcher = Executors
    .newFixedThreadPool(NUMBER_OF_THREADS)
    .asCoroutineDispatcher()
```

> `limitedParallelism` was introduced in `kotlinx-coroutines` version 1.6; in previous versions, we often created dispatchers with independent pools of threads using the `Executors` class.

Creating an executor with a fixed pool of threads can also be done using experimental multiplatform function `newFixedThreadPoolContext`.

```
private val NUMBER_OF_THREADS = 20
val dispatcher = newFixedThreadPoolContext(
    nThreads = 10,
    name = "background-thread"
)
```

The biggest problem with this approach is that a dispatcher created with both those ways needs to be closed with the `close` function. Developers often forget about this, which leads to leaking threads. Another problem is that when you create a fixed pool of threads, you are not using them efficiently because you will keep unused threads alive without sharing them with other services.

Dispatcher limited to a single thread

For all dispatchers using multiple threads, we need to consider the shared state problem. In the example below, notice that 10,000 coroutines are increasing `i` by 1. So, its value should be 10,000, but it is a smaller number. This is a result of a shared state (`i` property) modification on multiple threads at the same time.

```
var i = 0

suspend fun main(): Unit = coroutineScope {
    repeat(10_000) {
        launch(Dispatchers.IO) { // or Default
            i++
        }
    }
    delay(1000)
    println(i) // ~9930
}
```

There are many ways to solve this problem (most will be described in the *The problem with shared states* chapter), but one option is to use a dispatcher with just a single thread. If we use just a single thread at a time, we do not need any other synchronization. The classic way to do this used to be to create such a dispatcher using `Executors`. The problem with this is that this dispatcher keeps an extra thread active that needs to be closed when it is not used anymore. A modern solution is to use `Dispatchers.Default` or `Dispatchers.IO` (if we block threads) with parallelism limited to 1.

```
val dispatcher = Dispatchers.IO
    .limitedParallelism(1)

// previously:
// val dispatcher = Executors.newSingleThreadExecutor()
//     .asCoroutineDispatcher()


var i = 0

suspend fun main(): Unit = coroutineScope {
    val dispatcher = Dispatchers.Default
        .limitedParallelism(1)

    repeat(10000) {
        launch(dispatcher) {
            i++
        }
    }
    delay(1000)
    println(i) // 10000
}
```

Because we can use only one thread, the biggest disadvantage of using a dispatcher limited to a single thread is that our calls will be handled sequentially if we block it.

```
suspend fun main(): Unit = coroutineScope {
    val dispatcher = Dispatchers.Default
        .limitedParallelism(1)

    val launch = launch(dispatcher) {
        repeat(5) {
            launch {
                Thread.sleep(1000)
            }
        }
    }
    val time = measureTimeMillis { launch.join() }
    println("Took $time") // Took 5006
}
```

Using virtual threads from Project Loom

The JVM platform introduced a new technology known as Project Loom[7]. Its biggest innovation is the introduction of *virtual threads*, which are much lighter than regular threads. It costs much less to have blocked virtual threads than to have a regular thread blocked.

Project Loom does not have much to offer us developers who know Kotlin Coroutines because they have many more amazing features, like effortless cancellation or virtual time for testing[8]. However, Project Loom can be truly useful when we use its virtual threads instead of `Dispatcher.IO`, where we cannot avoid blocking threads[9].

To use Project Loom, we need to use JVM in the version over 19, and we currently need to enable preview features using the `--enable-preview` flag. Then, we should be able to create an executor using `newVirtualThreadPerTaskExecutor` from `Executors` and transform it into a coroutine dispatcher.

```
val LoomDispatcher = Executors
    .newVirtualThreadPerTaskExecutor()
    .asCoroutineDispatcher()
```

Such a dispatcher can also be implemented as an object declaration:

```
object LoomDispatcher : ExecutorCoroutineDispatcher() {

    override val executor: Executor = Executor { command ->
        Thread.startVirtualThread(command)
    }

    override fun dispatch(
        context: CoroutineContext,
        block: Runnable
    ) {
        executor.execute(block)
    }

    override fun close() {
```

[7]The stable version of Project Loom was released in September 2023 along with Java 21. Java 21 officially added the main features of Project Loom such as Virtual Threads (JEP 444) and Structured Concurrency (JEP 453).

[8]We will discuss this in the *Testing Kotlin Coroutines* chapter.

[9]The solution was inspired by the article *Running Kotlin coroutines on Project Loom's virtual threads by Jan Vladimir Mostert*.

```
        error("Cannot be invoked on Dispatchers.LOOM")
    }
}
```

To use this dispatcher similarly to other dispatchers, we can define an extension property on the `Dispatchers` object. This should also help this dispatcher's discoverability.

```
val Dispatchers.Loom: CoroutineDispatcher
    get() = LoomDispatcher
```

Now we only need to test if our new dispatcher really is an improvement. We expect it to take less memory and processor time than other dispatchers when we are blocking threads. We could set up the environment for precise measurements, or we could construct an example so extreme that anyone can observe the difference. For this book, I decided to use the latter approach. I started 100,000 coroutines, each blocked for 1 second. Making them do something else, like printing something or incrementing some value, should not change the result much. The execution of all these coroutines on `Dispatchers.Loom` took a bit more than two seconds.

```
suspend fun main() = measureTimeMillis {
    coroutineScope {
        repeat(100_000) {
            launch(Dispatchers.Loom) {
                Thread.sleep(1000)
            }
        }
    }
}.let(::println) // 2 273
```

Let's compare this new dispatcher to an alternative. Using pure `Dispatchers.IO` would not be fair as it is limited to 64 threads, and such function execution would take over 26 minutes. We should increment the thread limit to the number of coroutines. When I did that, such code execution took over 23 seconds, so ten times more. Of course, it consumed much more memory and processor time than the `Dispatchers.Loom` version. Whenever possible, we should use `Dispatchers.Loom` instead of `Dispatchers.IO`.

```
suspend fun main() = measureTimeMillis {
    val dispatcher = Dispatchers.IO
        .limitedParallelism(100_000)
    coroutineScope {
        repeat(100_000) {
            launch(dispatcher) {
                Thread.sleep(1000)
            }
        }
    }
}.let(::println) // 23 803
```

At the moment, Project Loom is still young and hard to use, but I must say it is an exciting substitution for `Dispatchers.IO`. However, you will likely not need it explicitly in the future as the Kotlin Coroutines team has expressed their willingness to use virtual threads by default once Project Loom is stable. I hope this happens soon.

Unconfined dispatcher

The last dispatcher we need to discuss is `Dispatchers.Unconfined`, which is different from all the other dispatchers as it does not change any threads. When it is started, it runs on the thread on which it was started. If it is resumed, it runs on the thread that resumed it.

```
//sampleStart
fun main() {
    var continuation: Continuation<Unit>? = null

    thread(name = "Thread1") {
        runBlocking(Dispatchers.Unconfined) {
            println(Thread.currentThread().name) // Thread1

            suspendCancellableCoroutine {
                continuation = it
            }

            println(Thread.currentThread().name) // Thread2

            delay(1000)

            println(Thread.currentThread().name)
            // kotlinx.coroutines.DefaultExecutor (used by \
```

```
delay)
        }
    }

    thread(name = "Thread2") {
        Thread.sleep(1000)
        continuation?.resume(Unit)
    }
}
```

In older projects, you can find `Dispatchers.Unconfined` used in unit tests. Imagine that you need to test a function that calls `launch`, for which synchronizing the time might not be easy. One solution is to use `Dispatchers.Unconfined` instead of all other dispatchers. If it is used in all scopes, everything runs on the same thread, and we can more easily control the order of operations. This trick is not needed anymore, as we have much better tools for testing coroutines. We will discuss this later in the book.

From the performance point of view, this dispatcher is cheapest as it never requires thread switching, therefore we might choose it if we do not care at all which thread our code runs on. However, it is not considered good to use it so recklessly in practice. What if, by accident, we miss a blocking call and we are running on the `Main` thread? This could lead to blocking the entire application. This is why we avoid using `Dispatchers.Unconfined` in production code, except for some special cases.

Immediate main dispatching

There is a cost associated with dispatching a coroutine. When `withContext` is called, the coroutine needs to be suspended, possibly wait in a queue, and then be resumed. This is a small but unnecessary cost if we are already on this thread. Look at the function below:

```
suspend fun showUser(user: User) =
    withContext(Dispatchers.Main) {
        userNameElement.text = user.name
        // ...
    }
```

If this function had already been called on the main dispatcher, we would have an unnecessary cost of redispatching. What is more, if there were a long queue for the Main thread because of `withContext`, the user data might be shown after some delay (this coroutine would need to wait for other coroutines to do their job first).

To prevent this, there is `Dispatchers.Main.immediate`, which dispatches only if it is needed. So, if the function below is called on the Main thread, it won't be re-dispatched: it will be called immediately.

```
suspend fun showUser(user: User) =
    withContext(Dispatchers.Main.immediate) {
        userNameElement.text = user.name
        // ...
    }
```

We prefer `Dispatchers.Main.immediate` as the `withContext` argument whenever this function might have already been called from the main dispatcher. Currently, the other dispatchers do not support immediate dispatching.

Continuation interceptor

Dispatching works based on the mechanism of continuation interception, which is built into the Kotlin language. There is a coroutine context named `ContinuationInterceptor`, whose `interceptContinuation` method is used to modify a continuation when a coroutine is suspended[10]. It also has a `releaseInterceptedContinuation` method that is called when a continuation is ended.

```
public interface ContinuationInterceptor :
    CoroutineContext.Element {

    companion object Key :
        CoroutineContext.Key<ContinuationInterceptor>

    fun <T> interceptContinuation(
        continuation: Continuation<T>
    ): Continuation<T>

    fun releaseInterceptedContinuation(
        continuation: Continuation<*>
    ) {
    }

    //...
}
```

[10] Wrapping needs to happen only once per continuation thanks to the caching mechanism.

The capability to wrap a continuation gives a lot of control. Dispatchers use `interceptContinuation` to wrap a continuation with `DispatchedContinuation`, which runs on a specific pool of threads. This is how dispatchers work.

The problem is that the same context is also used by many testing libraries, such as `runTest` from `kotlinx-coroutines-test`. Each element in a context has to have a unique key, which is why we sometimes inject dispatchers into unit tests to replace them with test dispatchers. We will get back to this topic in the chapter dedicated to coroutine testing.

```
class DiscUserRepository(
    private val discReader: DiscReader,
    private val dispatcher: CoroutineContext = Dispatchers.\
IO,
) : UserRepository {
    override suspend fun getUser(): UserData =
        withContext(dispatcher) {
            UserData(discReader.read("userName"))
        }
}

class UserReaderTests {

    @Test
    fun `some test`() = runTest {
        // given
        val discReader = FakeDiscReader()
        val repo = DiscUserRepository(
            discReader,
            // one of coroutines testing practices
            this.coroutineContext[ContinuationInterceptor]!!
        )
        //...
    }
}
```

Performance of dispatchers when executing different tasks

To show how different dispatchers perform in different tasks, I made some benchmarks. In all these cases, the task is to run 100 independent coroutines with the same task. The columns represent different tasks: suspending for a second, blocking for a second, CPU-intensive operation, and memory-intensive

operation (where the majority of the time is spent accessing, allocating, and freeing memory). Different rows represent the different dispatchers used for running these coroutines. The table below shows the average execution time in milliseconds.

	Suspending	**Blocking**	**CPU**	**Memory**
Single thread	1 002	100 003	39 103	94 358
Default (8 threads)	1 002	13 003	8 473	21 461
IO (64 threads)	1 002	2 003	9 893	20 776
100 threads	1 002	1 003	10 379	21 004

There are a few important observations you can make:

1. When we are just suspending, it doesn't really matter how many threads we are using.
2. When we are blocking, the more threads we are using, the faster all these coroutines will be finished.
3. With CPU-intensive operations, `Dispatchers.Default` is the best option[11].
4. If we are dealing with a memory-intensive problem, more threads might provide some (but not significant) improvement.

Here is how the tested functions look[12]:

```
fun cpu(order: Order): Coffee {
    var i = Int.MAX_VALUE
    while (i > 0) {
        i -= if (i % 2 == 0) 1 else 2
    }
    return Coffee(order.copy(customer = order.customer + i))
}

fun memory(order: Order): Coffee {
    val list = List(1_000) { it }
    val list2 = List(1_000) { list }
    val list3 = List(1_000) { list2 }
    return Coffee(
```

[11]The main reason is that the more threads we use, the more time the processor needs to spend switching between them, thus it has less time to do meaningful operations. Also `Dispatchers.IO` should not be used for CPU-intensive operations because it is used to block operations, and some other process might block all its threads.

[12]The whole code can be found at https://kt.academy/dispatchers-benchmarks

```
        order.copy(
            customer = order.customer + list3.hashCode()
        )
    )
}

fun blocking(order: Order): Coffee {
    Thread.sleep(1000)
    return Coffee(order)
}

suspend fun suspending(order: Order): Coffee {
    delay(1000)
    return Coffee(order)
}
```

Summary

Dispatchers determine which thread or thread pool a coroutine will run (starting and resuming) on. The most important options are:

- `Dispatchers.Default`, which we use for CPU-intensive operations;
- `Dispatchers.Main`, which we use to access the Main thread on Android, Swing, or JavaFX;
- `Dispatchers.Main.immediate`, which runs on the same thread as `Dispatchers.Main` but is not redispatched if it is not necessary;
- `Dispatchers.IO`, which we use when we need to do some blocking operations;
- `Dispatchers.IO` with limited parallelism or a custom dispatcher with a pool of threads, which we use when we might have many blocking calls;
- `Dispatchers.Default` or `Dispatchers.IO` with parallelism limited to 1, or a custom dispatcher with a single thread, which is used when we need to secure shared state modifications;
- `Dispatchers.Unconfined`, which does not change threads and is used in some special cases;

Exercise: Using dispatchers

For each of the functions below, decide if you need to set a dispatcher, and if so, which one:

- `createInvoice` - prepares an invoice and sends it to the SaaS API using the Retrofit library. In this library, we define suspending functions. The `toFakturowniaInvoiceData` function is a simple mapping.
- `sendEmail` - prepares an email and sends it using the `api` method of the `SendGrid` class. This is a blocking operation that returns a `Response` object.
- `getUserOrders` - gets orders made by the user from the database using the MongoDB Kotlin driver. `toList` is a suspending function that returns a list of orders.
- `upscaleImage` - a function that uses the TensorFlow library to upscale an image.

```
override suspend fun createInvoice(
    invoiceData: InvoiceData,
) {
    invoiceApi.postInvoice(
        invoiceData.toFakturowniaInvoiceData()
    )
}

interface InvoiceApi {
    @POST("invoices.json")
    suspend fun postInvoice(
        @Body invoice: SendInvoiceData
    ): InvoiceCreationResponse
}

private val invoiceApi = retrofit2.Retrofit.Builder()
    .baseUrl(invoiceBaseUrl)
    .build()
    .create(InvoiceApi::class.java)

private val sendGrid = SendGrid(API_KEY)

suspend fun sendEmail(email: Email) {
    val request = Request().apply {
        method = Method.POST
        endpoint = "mail/send"
        body = email.toSendGridEmail()
    }
    sendGrid.api(request)
}

private val orderCollection = mongoClient
    .getDatabase("shop")
    .getCollection<Order>("orders")
```

```
suspend fun getUserOrders(userId: String): List<Order> =
    orderCollection
        .find(eq("userId", userId))
        .toList()

val model = TensorFlowModel()

suspend fun upscaleImage(image: Image): Image =
    model.upscale(image)
```

Exercise: DiscNewsRepository

`DiscNewsRepository` is a simple class that uses `DiscReader` to read stored news from disk. The problem is that reading data from disk is a blocking operation, therefore the current implementation of `DiscNewsRepository` blocks threads in suspending functions, which leads to performance problems in your application. Fix this problem! Assume that `getNews` is used intensively and you want to support 200 parallel reads. Starting code:

```
class DiscNewsRepository(
    private val discReader: DiscReader
) : NewsRepository {
    override suspend fun getNews(newsId: String): News {
        val (title, content) = discReader.read("user/$newsI\
d")
        return News(title, content)
    }
}
```

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/dispatcher/DiscNewsRepository.kt`. You can clone this project and solve this exercise locally.

Exercise: Experiments with dispatchers

Experiment with how dispatcher choice influences execution time when you start 100 coroutines, each of which performs the following operations:

- CPU-intensive `cpuHeavy`
- Blocking `Thread.sleep(1000)`
- Suspending `delay(1000)`

```
val dispatcher = Dispatchers.IO.limitedParallelism(1)
//val dispatcher = Dispatchers.Default
//val dispatcher = Dispatchers.IO
//val dispatcher = Dispatchers.IO.limitedParallelism(100)

val operation = ::cpu1
//val operation = ::blocking
//val operation = ::suspending

fun cpu1() {
    var i = Int.MAX_VALUE
    while (i > 0) i -= if (i % 2 == 0) 1 else 2
}

fun blocking() {
    Thread.sleep(1000)
}

suspend fun suspending() {
    delay(1000)
}
```

Test the following dispatchers:

- Dispatcher limited to 1 thread
- `Dispatchers.Default`
- `Dispatchers.IO`
- Dispatcher limited to 100 threads

Test all combinations and fill the table below with the results. Then, try to explain the results.

Dispatcher	CPU-intensive	Blocking	Suspending
1 thread			
Dispatchers.Default			
Dispatchers.IO			
100 threads			

You can use the following code to measure execution time:

```
suspend fun main() = measureTimeMillis {
    coroutineScope {
        repeat(100) {
            launch(dispatcher) {
                operation()
                println("Done $it")
            }
        }
    }
}.let { println("Took $it") }
```

Starting code and example usage can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `coroutines/dispatcher/Experiments.kt`. You can clone this project and solve this exercise locally.

Job and coroutine lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Job and relationships

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Coroutine lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Awaiting job completion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

The Job factory function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Synchronizing coroutines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Cancellation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Basic cancellation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

The `finally` block

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

invokeOnCompletion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Cancellation of children

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Cancellation in a coroutine scope

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Just one more call

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Stopping the unstoppable

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

CancellationException is special

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

CancellationException does not propagate to its parent

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

withTimeout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

`suspendCancellableCoroutine`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Correct mistakes with cancellation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exception handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exceptions and structured concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

SupervisorJob

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Do not use SupervisorJob as a builder argument

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

supervisorScope

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Do not use `withContext(SupervisorJob())`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exceptions and await call

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

CoroutineExceptionHandler

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Constructing a coroutine scope

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

CoroutineScope factory function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Constructing a background scope

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Constructing a scope on Android

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: NotificationSender

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: BaseViewModel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Synchronizing access to mutable state

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Using atomic values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Volatile annotation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Using concurrent collections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Synchronized blocks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Using a dispatcher limited to a single thread

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Mutex

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Semaphore

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: CompanyDetailsRepository

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: CancellingRefresher

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: TokenRepository

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Suspended lazy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: mapAsync with concurrency limit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testing Kotlin Coroutines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testing time dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

`TestCoroutineScheduler` and `StandardTestDispatcher`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

runTest

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Background scope

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testing cancellation and context passing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

`UnconfinedTestDispatcher`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Using mocks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testing functions that change a dispatcher

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testing what happens during function execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testing functions that launch new coroutines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testing classes that need scope in runTest

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Replacing the main dispatcher

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testing Android functions that launch coroutines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Setting a test dispatcher with a rule

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Test UserDetailsRepository

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Testing mapAsync

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Testing the NotificationSender class

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Testing a View Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Part 3: Channel and Flow

In my workshops, many attendees are eager to finally hear about Flow - also known as coroutines' reactive streams. We are finally getting there: in this chapter, we will learn about Channel and Flow, both of which are useful tools that are worth knowing. We will start with Channel, as it is a bit more of a basic concept, then we will get deeply into Flow.

Channel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Channel types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

On buffer overflow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

On undelivered element handler

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Fan-out

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Fan-in

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Practical usage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: UserRefresher

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Cafeteria simulation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Select

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Selecting deferred values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Selecting from channels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: raceOf

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Hot and cold data sources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Hot vs cold

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Hot channels, cold flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Flow introduction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Comparing flow to other ways of representing values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

The characteristics of Flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Flow nomenclature

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Real-life use cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Understanding `Flow`

Kotlin Coroutines Flow is a much simpler concept than most developers think. It is just a definition of which operations to execute. It's similar to a suspending lambda expression, but with some extra elements. In this chapter, I will show you how to implement the `Flow` interface and `flow` builder by transforming a lambda expression step by step. This should give you a deep understanding of how Flow works. This chapter is for curious minds who like to truly understand the tools they use. If this isn't you, feel free to skip this chapter. If you decide to continue reading it, I hope you enjoy it.

Understanding `Flow`

We'll start our story with a simple lambda expression. Each lambda expression can be defined once and then called multiple times.

```
fun main() {
    val f: () -> Unit = {
        print("A")
        print("B")
        print("C")
    }
    f() // ABC
    f() // ABC
}
```

To make it a bit spicier, let's make our lambda expression `suspend` and add some delay inside it. Notice that each call of such a lambda expression is sequential, so you shouldn't make another call until the previous one is finished.

```
suspend fun main() {
    val f: suspend () -> Unit = {
        print("A")
        delay(1000)
        print("B")
        delay(1000)
        print("C")
    }
    f()
    f()
}
// A
// (1 sec)
// B
// (1 sec)
// C
// A
// (1 sec)
// B
// (1 sec)
// C
```

A lambda expression might have a parameter that can represent a function. We will call this parameter `emit`. So, when you call the lambda expression `f`, you need to specify another lambda expression that will be used as `emit`.

```
suspend fun main() {
    val f: suspend ((String) -> Unit) -> Unit = { emit ->
        emit("A")
        emit("B")
        emit("C")
    }
    f { print(it) } // ABC
    f { print(it) } // ABC
}
```

The fact is that `emit` should also be a suspending function. Our function type is already getting quite complex, so we'll simplify it by defining a `FlowCollector` function interface with an abstract method named `emit`. We will use this interface instead of the function type. The trick is that functional interfaces can be defined with lambda expressions, therefore we don't need to change the `f` call.

```
fun interface FlowCollector {
    suspend fun emit(value: String)
}

suspend fun main() {
    val f: suspend (FlowCollector) -> Unit = {
        it.emit("A")
        it.emit("B")
        it.emit("C")
    }
    f { print(it) } // ABC
    f { print(it) } // ABC
}
```

Calling `emit` on `it` is not convenient; instead, we'll make `FlowCollector` a receiver. Thanks to that, inside our lambda expression there is a receiver (`this` keyword) of type `FlowCollector`. This means we can call `this.emit` or just `emit`. The `f` invocation still stays the same.

```
fun interface FlowCollector {
    suspend fun emit(value: String)
}

suspend fun main() {
    val f: suspend FlowCollector.() -> Unit = {
        emit("A")
        emit("B")
        emit("C")
    }
    f { print(it) } // ABC
    f { print(it) } // ABC
}
```

Instead of passing around lambda expressions, we prefer to have an object implementing an interface. We will call this interface `Flow` and wrap our definition with an object expression.

```
fun interface FlowCollector {
    suspend fun emit(value: String)
}

interface Flow {
    suspend fun collect(collector: FlowCollector)
}

suspend fun main() {
    val builder: suspend FlowCollector.() -> Unit = {
        emit("A")
        emit("B")
        emit("C")
    }
    val flow: Flow = object : Flow {
        override suspend fun collect(
            collector: FlowCollector
        ) {
            collector.builder()
        }
    }
    flow.collect { print(it) } // ABC
    flow.collect { print(it) } // ABC
}
```

Finally, let's define the `flow` builder function to simplify our flow creation.

```
fun interface FlowCollector {
    suspend fun emit(value: String)
}

interface Flow {
    suspend fun collect(collector: FlowCollector)
}

fun flow(
    builder: suspend FlowCollector.() -> Unit
) = object : Flow {
    override suspend fun collect(collector: FlowCollector) {
        collector.builder()
    }
}

suspend fun main() {
```

```
    val f: Flow = flow {
        emit("A")
        emit("B")
        emit("C")
    }
    f.collect { print(it) } // ABC
    f.collect { print(it) } // ABC
}
```

The last change we need is to replace `String` with a generic type parameter in order to allow emitting and collecting any type of value.

```
fun interface FlowCollector<T> {
    suspend fun emit(value: T)
}

interface Flow<T> {
    suspend fun collect(collector: FlowCollector<T>)
}

fun <T> flow(
    builder: suspend FlowCollector<T>.() -> Unit
) = object : Flow<T> {
    override suspend fun collect(
        collector: FlowCollector<T>
    ) {
        collector.builder()
    }
}

suspend fun main() {
    val f: Flow<String> = flow {
        emit("A")
        emit("B")
        emit("C")
    }
    f.collect { print(it) } // ABC
    f.collect { print(it) } // ABC
}
```

That's it! This is nearly exactly how `Flow`, `FlowCollector`, and `flow` are implemented. When you call `collect`, you invoke the lambda expression from the `flow` builder call. When this expression calls `emit`, it calls the lambda expression specified when `collect` was called. This is how it works.

The presented builder is the most basic way to create a flow. Later we'll learn about other builders, but they generally just use `flow` under the hood.

```
public fun <T> Iterator<T>.asFlow(): Flow<T> = flow {
    forEach { value ->
        emit(value)
    }
}

public fun <T> Sequence<T>.asFlow(): Flow<T> = flow {
    forEach { value ->
        emit(value)
    }
}

public fun <T> flowOf(vararg elements: T): Flow<T> = flow {
    for (element in elements) {
        emit(element)
    }
}
```

How `Flow` processing works

`Flow` can be considered a bit more complicated than suspending lambda expressions with a receiver. However, its power lies in all the functions defined for its creation, processing, and observation. Most of them are actually very simple under the hood. We will learn about them in the next chapters, but I want you to have the intuition that most of them are very simple and can be easily constructed using `flow`, `collect`, and `emit`.

Consider the `map` function that transforms each element of a flow. It creates a new flow, so it uses the `flow` builder. When its flow is started, it needs to start the flow it wraps; so, inside the builder, it calls the `collect` method. Whenever an element is received, `map` transforms this element and then emits it to the new flow.

```
fun <T, R> Flow<T>.map(
    transformation: suspend (T) -> R
): Flow<R> = flow {
    collect {
        emit(transformation(it))
    }
}

suspend fun main() {
    flowOf("A", "B", "C")
        .map {
            delay(1000)
            it.lowercase()
        }
        .collect { println(it) }
}
// (1 sec)
// a
// (1 sec)
// b
// (1 sec)
// c
```

The behavior of most of the methods that we'll learn about in the next chapters is just as simple. It is important to understand this because it not only helps us better understand how our code works but also teaches us how to write similar functions.

```
fun <T> Flow<T>.filter(
    predicate: suspend (T) -> Boolean
): Flow<T> = flow {
    collect {
        if (predicate(it)) {
            emit(it)
        }
    }
}

fun <T> Flow<T>.onEach(
    action: suspend (T) -> Unit
): Flow<T> = flow {
    collect {
        action(it)
        emit(it)
```

```
    }
}

// simplified implementation
fun <T> Flow<T>.onStart(
    action: suspend () -> Unit
): Flow<T> = flow {
    action()
    collect {
        emit(it)
    }
}
```

Flow is synchronous

Notice that Flow is synchronous by nature, just like suspending functions: the `collect` call is suspended until the flow is completed. This also means that a flow doesn't start any new coroutines. Its concrete steps can do it, just like suspending functions can start coroutines, but this is not the default behavior for suspending functions. Most flow processing steps are executed synchronously, which is why a `delay` inside `onEach` introduces a delay **between** each element, not before all elements.

```
suspend fun main() {
    flowOf("A", "B", "C")
        .onEach { delay(1000) }
        .collect { println(it) }
}
// (1 sec)
// A
// (1 sec)
// B
// (1 sec)
// C
```

That can be changed with functions like `buffer` or `conflate`, that start a new coroutine for everything above, or with coroutine builders like `channelFlow`, that start a new coroutine for its body. But the default behavior is synchronous.

Flow and shared state

When you implement more complex algorithms for flow processing, you should know when you need to synchronize access to variables. Let's analyze the most important use cases. When you implement some custom flow processing functions, you can define mutable states inside the flow without any mechanism for synchronization because a flow step is synchronous by nature.

```
fun <T, K> Flow<T>.distinctBy(
    keySelector: (T) -> K
) = flow {
    val sentKeys = mutableSetOf<K>()
    collect { value ->
        val key = keySelector(value)
        if (key !in sentKeys) {
            sentKeys.add(key)
            emit(value)
        }
    }
}
```

Here is an example that is used inside a flow step and produces consistent results; the counter variable is always incremented to 1000.

```
fun Flow<*>.counter() = flow<Int> {
    var counter = 0
    collect {
        counter++
        // to make it busy for a while
        List(100) { Random.nextLong() }.shuffled().sorted()
        emit(counter)
    }
}

suspend fun main(): Unit = coroutineScope {
    val f1 = List(1000) { "$it" }.asFlow()
    val f2 = List(1000) { "$it" }.asFlow()
        .counter()

    launch { println(f1.counter().last()) } // 1000
    launch { println(f1.counter().last()) } // 1000
    launch { println(f2.last()) } // 1000
    launch { println(f2.last()) } // 1000
}
```

It is a common mistake to extract a variable from outside a flow step into a function. Such a variable is shared between all the coroutines that are collecting from the same flow. **It requires synchronization and is flow-specific, not flow-collection-specific**. Therefore, `f2.last()` returns around 2000, not 1000, because it is a result of counting elements from two flow executions in parallel.

```
fun Flow<*>.counter(): Flow<Int> {
    var counter = 0
    return this.map {
        counter++
        // to make it busy for a while
        List(100) { Random.nextLong() }.shuffled().sorted()
        counter
    }
}

suspend fun main(): Unit = coroutineScope {
    val f1 = List(1_000) { "$it" }.asFlow()
    val f2 = List(1_000) { "$it" }.asFlow()
        .counter()

    launch { println(f1.counter().last()) } // 1000
    launch { println(f1.counter().last()) } // 1000
    launch { println(f2.last()) } // less than 2000
    launch { println(f2.last()) } // less than 2000
}
```

Finally, just as suspending functions using the same variables need synchronization, a variable used in a flow needs synchronization if it's defined outside a function, on the scope of a class, or at the top-level.

```
var counter = 0

fun Flow<*>.counter(): Flow<Int> = this.map {
    counter++
    // to make it busy for a while
    List(100) { Random.nextLong() }.shuffled().sorted()
    counter
}

suspend fun main(): Unit = coroutineScope {
    val f1 = List(1_000) { "$it" }.asFlow()
    val f2 = List(1_000) { "$it" }.asFlow()
        .counter()
```

```
    launch { println(f1.counter().last()) } // less than 40\
00
    launch { println(f1.counter().last()) } // less than 40\
00
    launch { println(f2.last()) } // less than 4000
    launch { println(f2.last()) } // less than 4000
}
```

Conclusion

`Flow` can be considered a bit more complicated than a suspending lambda expression with a receiver, and its processing functions just decorate it with new operations. There is no magic here: how `Flow` and most of its methods are defined is simple and straightforward.

Flow building

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Flow from raw values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Converters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Converting a function to a flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Flow and Reactive Streams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Flow builders

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Understanding flow builder

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

channelFlow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

callbackFlow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Flow utils

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: All users flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: distinct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Flow lifecycle functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

onEach

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

onStart

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

onCompletion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

onEmpty

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

catch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Uncaught exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

`retry`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

flowOn

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

launchIn

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: TemperatureService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: NewsViewModel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Flow processing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

`map`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

`filter`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

`take` and `drop`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

How does collection processing work?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

`merge`, `zip` and `combine`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

fold and scan

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

flatMapConcat, flatMapMerge and flatMapLatest

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Distinct until changed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

buffer and conflate

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

debounce

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

sample

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Terminal operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: ProductService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Flow Kata

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: MessageService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

SharedFlow and StateFlow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

SharedFlow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

`shareIn`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

StateFlow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

`stateIn`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Update ProductService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Update TemperatureService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: LocationService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: PriceService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: NewsViewModel using stateIn

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testing flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Transformation functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testing infinite flows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Determining how many connections were opened

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Testing view models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Exercise: Flow testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Part 4: Kotlin Coroutines in practice

At this point, you have all the puzzles to use coroutines in real-life projects. The previous chapters cover all the elements, from using suspending functions to operating on flow. We've also seen some use cases and common mistakes. In this chapter, we will review them, and try to build a deeper understanding of how Kotlin Coroutines should be used in real-life projects, and how they should not be used. We will cover the most typical use cases and best practices. This chapter is built on top of the previous chapters, and it will often reference them. It should be your essential guideline and your summary.

Common use cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Data/Adapters Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Callback functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Blocking functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Observing with Flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Domain Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Concurrent calls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Flow transformations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Presentation/API/UI layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Creating custom scope

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Using runBlocking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Working with Flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Using coroutines from other languages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Threads on different platforms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Transforming suspending into non-suspending functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Transforming suspend functions into blocking functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Transforming suspend functions into callback functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Platform-specific options

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Calling suspending functions from other languages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Flow and Reactive Streams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Launching coroutines vs. suspending functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Best practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Don’t use async with an immediate await

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Use awaitAll

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Suspending functions should be safe to call from any thread

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Use `Dispatchers.Main.immediate` instead of `Dispatchers.Main`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Remember to use `yield` in heavy functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Understand that suspending functions await completion of their children

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Understand that `Job` is not inherited: it is used as a parent

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Don't break structured concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Use SupervisorJob when creating CoroutineScope

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Consider cancelling scope children

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Before using a scope, consider under which conditions it is cancelled

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Don't use GlobalScope

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Avoid using `Job` builder, except for constructing a scope

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Functions that return `Flow` should not be suspending

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Prefer a suspending function instead of Flow when you expect only one value

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

The End

All good things must come to an end, and the same is sadly true of this book. Of course, there is still much more to say about Kotlin Coroutines, but I believe we've covered the essentials well. Now it is your turn to start using this knowledge in practice and deepen your understanding.

Exercise solutions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Factorial sequence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Prime numbers sequence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Callback function wrappers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Continuation storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: What is stored by a continuation?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: UserDetailsRepository

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: BestStudentUseCase

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: CommentService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: mapAsync

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Understanding context propagation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: CounterContext

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Using dispatchers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: DiscNewsRepository

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Experiments with dispatchers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Correct mistakes with cancellation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: NotificationSender

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: BaseViewModel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: CompanyDetailsRepository

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: CancellingRefresher

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: TokenRepository

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Suspended lazy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: mapAsync with concurrency limit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Test UserDetailsRepository

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Testing mapAsync

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Testing the NotificationSender class

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: UserRefresher

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Cafeteria simulation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: raceOf

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Flow utils

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: All users flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: distinct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: TemperatureService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: NewsViewModel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: ProductService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Flow Kata

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: MessageService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Update ProductService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: Update TemperatureService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: LocationService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: PriceService

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.

Solution: NewsViewModel using stateIn

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/coroutines.