Chapter 6: Threads and Coroutines

In the previous chapter, we had a glance at how our application can efficiently serve thousands of requests per second—to discuss why immutability is important, we introduced a race condition problem using two threads.

In this chapter, we'll dive deeper into how to launch new threads in Kotlin and the reasons why coroutines can scale much better than threads. We will discuss how the Kotlin compiler treats coroutines and the relationship between coroutine scopes and dispatchers. We'll discuss the concept of **structured concurrency**, and how it helps us prevent resource leaks in our programs.

We'll cover the following topics in this chapter:

- Looking deeper into threads
- Introducing coroutines and suspend functions
- Starting coroutines
- Jobs
- Coroutines under the hood
- Dispatchers
- Structured concurrency

After reading this chapter, you'll be familiar with Kotlin's concurrency primitives and how to best utilize them.

Technical requirements

In addition to the requirements from the previous chapters, you will also need a **Gradle**-enabled **Kotlin** project to be able to add the required dependencies.

You can find the source code for this chapter here: https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter06.

Looking deeper into threads

Before we dive into the nuances, let's discuss what kinds of problems threads can solve.

In your laptop, you have a CPU with multiple cores – probably four of them, or even eight. This means that it can do four different computations *in parallel*, which is pretty amazing considering that 15 years ago, a single-core CPU was the default and even two cores were only for enthusiasts.

But even back then, you were not limited to doing only a single task at a time, right? You could listen to music and browse the internet at the same time, even on a single-core CPU. *How does your CPU manage to pull that off?* Well, the same way your brain does. It juggles tasks. When you're reading a book while listening to your friend talking, part of the time, you're not reading, and part of the time, you're not listening – that is, until we get at least two cores in our brains.

The servers you run your code on have pretty much the same CPU. This means that they can serve four requests simultaneously. *But what if you have 10,000 requests per second?* You can't serve them in parallel because you don't have 10,000 CPU cores. But you can try and serve them concurrently.

The most basic concurrency model provided by JVM is known as a **thread**. Threads allow us to run code concurrently (but not necessarily in parallel) so that we can make better use of multiple CPU cores, for example. They are more lightweight than processes. One process may spawn hundreds of threads. Unlike processes, sharing data between threads is easy. But that also introduces a lot of problems, as we'll see later.

Let's learn how to create two threads in Java first. Each thread will output numbers between `0` and `100`:

```
for (int t = 0; t < 2; t++) {
    int finalT = t;
    new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            System.out.println("Thread " + finalT +
":               " + i);
        }
    }).start();
}
```

The output will look something like this:

```
> ...
> T0: 12
> T0: 13
> T1: 60
> T0: 14
> T1: 61
> T0: 15
> T1: 16
> ...
```

Note that the output will vary between executions and that at no point is it guaranteed to be interleaved.

The same code in Kotlin would look as follows:

```
repeat(2) { t ->
    thread {
        for (i in 1..100) {
            println("T$t: $i")
        }
    }
}
```

In Kotlin, there's less boilerplate because there's a function that helps us create a new thread. Notice that, unlike Java, we don't need to call **`start()`** to launch the thread. It starts by default. If we would like to postpone it for later, we can set the **`start`** parameter to **`false`**:

```
val t = thread(start = false)
...
// Later
t.start()
```

Another useful concept from Java is **daemon threads**. These threads don't prevent JVM from exiting and are very good for non-critical background tasks.

In Java, the API is not fluent, so we'll have to assign our thread to a variable, set it to be a daemon thread, and then start it. In Kotlin, this is much simpler:

```
thread(isDaemon = true) {
    for (i in 1..1_000_000) {
        println("daemon thread says: $i")
    }
}
```

Notice that although we asked this thread to print numbers up to one million, it prints only a few hundred. That's because it's a daemon thread. When the parent thread stops, all the daemon threads stop as well.

Thread safety

There are entire books written about **thread safety** and there are good reasons for this. Concurrency bugs that are caused by a lack of thread safety are the hardest ones to track. They're hard to reproduce because you'll usually need a lot of threads competing for the same resource in order for an actual race to happen. Because this book is about Kotlin and not thread safety in general, we'll only scratch the surface of this topic. If you're interested in the topic of thread safety in the JVM language, you should check out the book *Java Concurrency in Practice*, by Brian Goetz.

We'll start with the following example, which creates 100,000 threads to increment a **`counter`**. To make sure that all the threads complete their work before we check the value, we'll use **`CountDownLatch`**:

```
var counter = 0
val latch = CountDownLatch(100_000)
repeat(100) {
    thread {
        repeat(1000) {
            counter++
            latch.countDown()
        }
    }
}
latch.await()
println("Counter $counter")
```

The reason this code doesn't print the correct number is that we introduced a data race since the **`++`** operation is not atomic. So, if more threads try to increment our counter, then there are more chances for data races.

Unlike Java, there's no **`synchronized`** keyword in Kotlin. The reason for this is that Kotlin designers believe that a language shouldn't be tailored to a particular concurrency model. Instead, there's a **`synchronized()`** function we can use:

```
thread {
    repeat(1000) {
        synchronized(latch) {
            counter++
            latch.countDown()
            }
        }
    }
```

Now, our code prints **`100,000`**, as expected.

If you miss the synchronized methods from Java, there's the **`@Synchronized`** annotation in Kotlin. Java's **`volatile`** keyword is also replaced by the **`@Volatile`** annotation instead. The following table shows us an example of this comparison:

Java	**Kotlin**
`synchronized void doSomething()`	`@Synchronized fun doSomething()`

`volatile int sharedCounter = 0;`	`@Volatile`
	 `var sharedCounter: Int = 0`

Table 6.1 – Comparison between Java and Kotlin (synchronized and volatile methods)

The reason **`Synchronized`** and **`Volatile`** are annotations and not keywords is because Kotlin can be compiled on other platforms in addition to JVM. But the concepts of **`synchronized`** methods or **`volatile`** variables exist for JVM specifically.

Why are threads expensive?

There is a price to pay whenever we create a new thread. Each thread needs a new memory stack.

What if we simulate some work inside each thread by putting it to sleep?

In the following piece of code, we'll attempt to create 10,000 threads, each sleeping for a relatively short period:

```
val counter = AtomicInteger()
try {
    for (i in 0..10_000) {
        thread {
            counter.incrementAndGet()
            Thread.sleep(100)
        }
    }
} catch (oome: OutOfMemoryError) {
    println("Spawned ${counter.get()} threads before
      crashing")
    System.exit(-42)
}
```

Each thread requires one megabyte of RAM for its stack. Creating so many threads will require lots of communication with your operating system and a lot of memory. We attempt to identify whether we ran out of memory by catching the relevant exception.

Depending on your operating system, this will result in either **`OutOfMemoryError`** or the entire system becoming very slow.

Of course, there are ways to limit how many threads are run at once using the **Executors API**. This API was introduced back in **Java 5**, so it should be pretty well-known to you.

Using that API, we can create a new thread pool of a specified size. Try setting the **`pool`** size to **`1`**, the number of cores on your machine to **`100`** and **`2000`**, and see what happens:

```
val pool = Executors.newFixedThreadPool(100)
```

Now, we would like to submit a new task. We can do this by calling **`pool.submit()`**:

```
val counter = AtomicInteger(0)
val start = System.currentTimeMillis()
for (i in 1..10_000) {
    pool.submit {
        // Do something
        counter.incrementAndGet()
```

```
            // Simulate wait on IO
            Thread.sleep(100)
            // Do something again
            counter.incrementAndGet()
        }
    }
```

By incrementing **`counter`** once before **`sleep`** and once after, we are simulating some business logic – for example, preparing some JSON and then parsing the response – while **`sleep`** itself simulates a network operation.

Then, we need to make sure that the pool terminates and give it **`20`** seconds to do so by using the following lines:

```
pool.awaitTermination(20, TimeUnit.SECONDS)
pool.shutdown()
println("Took me ${System.currentTimeMillis() -
start}   millis to complete ${counter.get() / 2}
tasks")
```

Notice that it took us 20 seconds to complete. That's because a new task cannot begin until the previous tasks *wake up* and finish their jobs.

And that's exactly what happens in a multithreaded system that is not concurrent enough.

In the next section, we'll discuss how coroutines try to solve this problem.

Introducing coroutines

In addition to the threading model provided by Java, Kotlin also has a **coroutines** model. Coroutines might be considered lightweight threads, and we'll see what advantages they provide over an existing model of threads shortly.

The first thing you need to know is that coroutines are not part of the language. They are simply another library provided by JetBrains. For that reason, if we want to use them, we need to specify this in our Gradle configuration file; that is, **`build.gradle.kts`**:

```
dependencies {
    ...
    implementation("org.jetbrains.kotlinx:kotlinx-
   coroutines-core:1.5.1")
}
```

IMPORTANT NOTE:

By the time you read this book, the latest version of the Coroutines library will be **1.6** *or greater.*

First, we will compare starting a new thread and a new coroutine.

Starting coroutines

We've already seen how to start a new thread in Kotlin in the *Looking deeper into threads* section. Now, let's start a new coroutine instead.

We'll create almost the same example we did with threads. Each coroutine will increment some counter, sleep for a while to emulate some kind of I/O, and then increment it again:

```
val latch = CountDownLatch(10_000)
val c = AtomicInteger()
val start = System.currentTimeMillis()
for (i in 1..10_000) {
    GlobalScope.launch {
        c.incrementAndGet()
        delay(100)
        c.incrementAndGet()
        latch.countDown()
    }
}
```

```
  latch.await(10, TimeUnit.SECONDS)
  println("Executed ${c.get() / 2} coroutines in
    ${System.currentTimeMillis() - start}ms")
```

The first way of starting a new coroutine is by using the **`launch()`** function. Again, note that this is simply another function and not a language construct.

Another interesting point here is the call to the **`delay()`** function, which we use to simulate some I/O-bound work, such as fetching something from a database or over the network.

Like the **`Thread.sleep()`** method, it puts the current coroutine to sleep. But unlike **`Thread.sleep()`**, other coroutines can work while it sleeps soundly. This is because **`delay()`** is marked with a **`suspend`** keyword, which we'll discuss in the *Jobs* section.

If you run this code, you'll see that the task takes about 200 ms with coroutines, while with threads, it either takes 20 seconds or runs out of memory. And we didn't have to change our code that much. That's all thanks to the fact that coroutines are highly concurrent. They can be suspended without blocking the thread that runs them. Not blocking a thread is great because we can use fewer OS threads (which are expensive) to do more work.

If you run this code in your IntelliJ IDEA, you'll notice that **`GlobalScope`** is marked as a **delicate API**. This means that **`GlobalScope`** shouldn't be used in real-world projects unless the developer understands how it works under the hood. Otherwise, it may cause unintended leaks. We'll learn about better ways of launching coroutines later in this chapter.

Although we've seen that coroutines are much more concurrent than threads, there's nothing magical in them. Now, let's learn about another way of starting a coroutine, as well as some issues coroutines may still suffer from.

The **launch()** function that we just discussed starts a coroutine that doesn't return anything. In contrast, the **async()** function starts a coroutine that returns some value.

Calling **launch()** is much like calling a function that returns **Unit**. But most of our functions return some kind of result. For that purpose, we have the **async()** function. It also launches a coroutine, but instead of returning a job, it returns **Deferred<T>**, where **T** is the type that you expect to get later.

For example, the following function will start a coroutine that generates a UUID asynchronously and returns it:

```
fun fastUuidAsync() = GlobalScope.async {
    UUID.randomUUID()
}
println(fastUuidAsync())
```

If we run the following code from our **main** method, though, it won't print the expected result. The result that this code prints instead of some UUID value is as follows:

```
> DeferredCoroutine{Active}
```

The returned object from a coroutine is called a job. Let's understand what this is and how to use it correctly.

Jobs

The result of running an asynchronous task is called a **job**. Much like the **Thread** object represents an actual OS thread, the **job** object represents an actual coroutine.

This means that what we tried to do is this:

```
val job: Job = fastUuidAsync()
println(job)
```

job has a simple life cycle. It can be in one of the following states:

- **New**: Created but not started yet.
- **Active**: Just created by the `launch()` function, for example. This is the default state.
- **Completed**: Everything went well.
- **Canceled**: Something went wrong.

Two more states are relevant to jobs that have child jobs:

- **Completing**: Waiting to finish executing children before completing
- **Canceling**: Waiting to finish executing children before canceling

If you want to learn more about parent and child jobs, jump to the *Parent jobs* section of this chapter.

The job we've confused with its value is in the Active state, meaning that it hasn't finished computing our UUID yet.

A job that has a value is known as being `Deffered`:

```
val job: Deferred<UUID> = fastUuidAsync()
```

We'll discuss the `Deferred` value in more detail in *Chapter 8*, *Designing for Concurrency*.

To wait for a job to complete and get the actual value, we can use the `await()` function:

```
val job: Deferred<UUID> = fastUuidAsync()
println(job.await())
```

This code doesn't compile, though:

```
> Suspend function 'await' should be called only from
a coroutine or another suspend function
```

The reason for this is that, as stated in the error itself, our `main()` function is not marked with a `suspend` keyword and isn't a coroutine either.

We can fix this by wrapping our code in a `runBlocking` function:

```
runBlocking {
    val job: Deferred<UUID> = fastUuidAsync()
    println(job.await())
}
```

This function will block our main thread until all the coroutines finish. It is an implementation of the Bridge design pattern from Chapter 4, *Getting Familiar with Behavioral Patterns*, which allows us to connect between regular code and code that uses coroutines.

Running this code now will produce the expected output of some random UUID.

IMPORTANT NOTE:

In this chapter, while discussing coroutines, we will sometimes omit **runBlocking** *for conciseness. You can always find the full working examples in this book's GitHub repository.*

The **job** object also has some other useful methods, which we'll discuss in the following sections.

Coroutines under the hood

So, we've mentioned the following facts a couple of times:

- Coroutines are like lightweight threads. They need fewer resources than regular threads, so you can create more of them.
- Instead of blocking an entire thread, coroutines suspend themselves, allowing the thread to execute another piece of code in the meantime.

But how do coroutines work?

As an example, let's take a look at a function that composes a user profile:

```
fun profileBlocking(id: String): Profile {
    // Takes 1s
    val bio = fetchBioOverHttpBlocking(id)
    // Takes 100ms
    val picture = fetchPictureFromDBBlocking(id)
    // Takes 500ms
    val friends = fetchFriendsFromDBBlocking(id)
    return Profile(bio, picture, friends)
}
```

Here, our function takes around 1.6 seconds to complete. Its execution is completely sequential, and the executing thread will be blocked for the entire time.

We can redesign this function so that it works with coroutines, as follows:

```
suspend fun profile(id: String): Profile {
    // Takes 1s
    val bio = fetchBioOverHttpAsync(id)
    // Takes 100ms
    val picture = fetchPictureFromDBAsync(id)
    // Takes 500ms
    val friends = fetchFriendsFromDBAsync(id)
    return Profile(bio.await(), picture.await(),
      friends.await())
}
```

Without the `suspend` keyword, our asynchronous code simply won't compile. We'll cover what the `suspend` keyword means later in this section.

To understand what each of the asynchronous functions looks like, let's take a look at one of them as an example:

```
fun fetchFriendsFromDBAsync(id: String) =
GlobalScope.async
{
    delay(500)
    emptyList<String>()
}
```

Now, let's compare the performance of the two functions: one that is written in a blocking manner, and another that uses coroutines.

We can wrap both functions using a **`runBlocking`** function, as we've seen previously, and measure the time it takes them to complete using **`measureTimeMillis`**:

```
runBlocking {
    val t1 = measureTimeMillis {
        blockingProfile("123")
    }
    val t2 = measureTimeMillis {
        profile("123")
    }
    println("Blocking code: $t1")
    println("Async: $t2")
}
```

The output will be something like this:

```
> Blocking code: 1623
> Coroutines: 1021
```

The execution time of the concurrent coroutines is the maximum of the longest coroutine, while with sequential code, it's the sum of all functions.

Having understood the first two examples, let's look at another way to write the same code.

We'll mark each of the functions with the **`suspend`** keyword:

```
suspend fun fetchFriendsFromDB(id: String):
List<String> {
    delay(500)
    return emptyList()
}
```

If you run this example, the performance will be the same as the blocking code. *So, why would we want to use suspendable functions?*

Suspendable functions don't block the thread. Looking at the bigger picture, by using the same number of threads, we can serve far more users, all thanks to the smart way Kotlin rewrites suspendable functions.

When the Kotlin compiler sees the **`suspend`** keyword, it knows it can split and rewrite the function, like this:

```
fun profile(state: Int, id: String, context:
ArrayList<Any>): Profile {
    when (state) {
        0 -> {
            context += fetchBioOverHttp(id)
            profile(1, id, context)
        }
        1 -> {
            context += fetchPictureFromDB(id)
            profile(2, id, context)
        }
        2 -> {
            context += fetchFriendsFromDB(id)
            profile(3, id, context)
        }
        3 -> {
            val (bio, picture, friends) = context
            return Profile(bio, picture, friends)
        }
    }
}
```

This rewritten code uses the **State design pattern** from *Chapter 4*, *Getting Familiar with Behavioral Patterns*, to split the execution of the function into many steps. By doing so, we can release the thread that executes coroutines at every stage of the state machine.

IMPORTANT NOTE:

This is not a perfect depiction of the generated code. The goal is to demonstrate the idea behind what the Kotlin compiler does, but some subtle implementation details are omitted for brevity.

Note that unlike the asynchronous code we produced earlier, the state machine itself is sequential and takes the same amount of time as the blocking code to execute all its steps.

It is a fact that none of these steps block any threads, which is important in this example.

Canceling a coroutine

If you are a Java developer, you may know that stopping a thread is quite complicated.

For example, the **`Thread.stop()`** method is deprecated. There's **`Thread.interrupt()`**, but not all threads are checking this flag, not to mention setting a **`volatile`** flag, which is often suggested but is very cumbersome.

If you're using a thread pool, you'll get **`Future`**, which has the **`cancel(boolean mayInterruptIfRunning)`** method. In Kotlin, the **`launch()`** function returns a job.

This job can be canceled. The same rules from the previous example apply, though. If your coroutine never calls another **`suspend`** function or the **`yield`** function, it will disregard **`cancel()`**.

To demonstrate that, we'll create one coroutine that yields once in a while:

```
val cancellable = launch {
    try {
        for (i in 1..10_000) {
            println("Cancellable: $i")
            yield()
        }
    }
    catch (e: CancellationException) {
        e.printStackTrace()
    }
}
```

As you can see, after each **print** statement, the coroutine calls the **yield** function. If it was canceled, it will print the stack trace.

We'll also create another coroutine that doesn't yield:

```
val notCancellable = launch {
    for (i in 1..10_000) {
        if (i % 100 == 0) {
            println("Not cancellable $i")
        }
    }
}
```

This coroutine never yields and prints its results every **100** iterations to avoid spamming the console.

Now, let's try cancelling both coroutines:

```
println("Canceling cancellable")
cancellable.cancel()
println("Canceling not cancellable")
notCancellable.cancel()
```

Then, we'll wait for the results:

```
runBlocking {
    cancellable.join()
    notCancellable.join()
}
```

By invoking **join()**, we can wait for the execution of the coroutine to complete.

Let's look at the output of our code:

```
> Canceling cancellable
> Cancellable: 1
> Not cancellable 100
>...
> Not cancellable 1000
> Canceling not cancellable
```

A few interesting points we can learn from this experiment regarding the behavior of coroutines are as follows:

- Canceling the **cancellable** coroutine doesn't happen immediately. It may still print a line or two before being canceled.
- We can catch **CancellationException**, but our coroutine will be marked as canceled anyway. Catching that exception doesn't automatically allow us to continue.

Now, let's understand what happened. The coroutine checks whether it was canceled, but only when it is switching between states. Since the non-cancellable coroutine didn't have any suspending functions, it never checked if it was asked to stop.

In the **cancellable** coroutine, we used a new function: **yield()**. We could have called **yield()** on every loop iteration, but decided to do that every 100th one. This function checks whether there is anybody else that wants to do some work. If there's nobody else, the execution of the current coroutine will resume. Otherwise, another coroutine will start or resume from the point where it stopped earlier.

Note that without the **suspend** keyword on our function or a coroutine generator, such as **launch()**, we can't call **yield()**. This is true for any function marked with **suspend**: it should either be called from another **suspend** function or from a coroutine.

Setting timeouts

Let's consider the following situation. *What if, as happens in some cases, fetching the user's profile takes too long? What if we decided that if the profile takes more than 0.5 seconds to return, we'll just show no profile?*

This can be achieved using the **withTimeout()** function:

```
val coroutine = async {
    withTimeout(500) {
        try {
            val time = Random.nextLong(1000)
            println("It will take me $time to do")
            delay(time)
            println("Returning profile")
            "Profile"
        }
        catch (e: TimeoutCancellationException) {
            e.printStackTrace()
        }
    }
}
```

We set the timeout to be **500** milliseconds, and our coroutine will delay for between **0** and **1000** milliseconds, giving it a 50 percent chance of failing.

We'll **await** the results from the coroutine and see what happens:

```
val result = try {
    coroutine.await()
}
catch (e: TimeoutCancellationException) {
    "No Profile"
}
println(result)
```

Here, we benefit from the fact that **`try`** is an expression in Kotlin. So, we can return a result immediately from it.

If the coroutine manages to return before the timeout, the value of **`result`** becomes **`profile`**. Otherwise, we receive **`TimeoutCancellationException`** and set the value of **`result`** to **`no profile`**.

A combination of timeouts and **`try`**-**`catch`** expressions is a really powerful tool that allows us to create robust interactions.

Dispatchers

When we ran our coroutines using the **`runBlocking`** function, their code was executed on the main thread.

You can check this by running the following code:

```
runBlocking {
    launch {
        println(Thread.currentThread().name) //
Prints
          "main"
    }
}
```

In contrast, when we run a coroutine using **`GlobalScope`**, it runs on something called **`DefaultDispatcher`**:

```
GlobalScope.launch {
    println("GlobalScope.launch:
      ${Thread.currentThread().name}")
}
```

This prints the following output:

```
> DefaultDispatcher-worker-1
```

DefaultDispatcher is a thread pool that is used for short-lived coroutines.

Coroutine generators, such as **launch()** and **async()**, rely on default arguments, one of which is the dispatcher they will be launched on. To specify an alternative dispatcher, you can provide it as an argument to the coroutine builder:

```
runBlocking {
    launch(Dispatchers.Default) {
        println(Thread.currentThread().name)
    }
}
```

The preceding code prints the following output:

```
> DefaultDispatcher-worker-1
```

In addition to the **Main** and **Default** dispatchers, which we've already discussed, there is also an **IO** dispatcher, which is used for long-running tasks. You can use it similarly for other dispatchers by providing it to the coroutine builder, like so:

```
async(Dispatchers.IO) {
    // Some long running task here
}
```

Structured concurrency

It is a very common practice to spawn coroutines from inside another coroutine.

The first rule of structured concurrency is that the parent coroutine should always wait for all its children to complete. This prevents resource leaks, which is very common in languages that don't have the **structured concurrency** concept.

This means that if we look at the following code, which starts 10 child coroutines, the parent coroutine doesn't need to wait explicitly for all of them to complete:

```
val parent = launch(Dispatchers.Default) {
    val children = List(10) { childId ->
        launch {
            for (i in 1..1_000_000) {
                UUID.randomUUID()
                if (i % 100_000 == 0) {
                    println("$childId - $i")
                    yield()
                }
            }
        }
    }
}
```

Now, let's decide that one of the coroutines throws an exception after some time:

```
...
if (i % 100_000 == 0) {
    println("$childId - $i")
    yield()
}
if (childId == 8 && i == 300_000) {
    throw RuntimeException("Something bad happened")
}
...
```

If you run this code, something interesting happens. Not only does the coroutine itself terminate, but also all its siblings are terminated as well.

What happens here is that an uncaught exception bubbles up to the parent coroutine and cancels it. Then, the parent coroutine terminates all the other child coroutines to prevent any resource leaks.

Usually, this is the desired behavior. If we'd like to prevent child exceptions from stopping the parent as well, we can use **`supervisorScope`**:

```
val parent = launch(Dispatchers.Default) {
    supervisorScope {
        val children = List(10) { childId ->
            ...
        }
    }
}
```

By using **`supervisorScope`**, even if one of the coroutines fails, the parent job won't be affected.

The parent coroutine can still terminate all its children by using the **`cancel()`** function. Once we invoke **`cancel()`** on the parent job, all of its children are canceled too.

Now that we've discussed the benefits of structured concurrency, let's reiterate one point from the start of this chapter: using **`GlobalScope`** and the fact that it's marked as a **delicate API**. Although **`GlobalScope`** exposes functions such as **`launch()`** and **`async()`**, it doesn't benefit from structured concurrency principles and is prone to resource leaks when used incorrectly. For that reason, you should avoid using **`GlobalScope`** in real-world applications.

Summary

In this chapter, we covered how to create threads and coroutines in Kotlin, as well as the benefits of coroutines over threads.

Kotlin has simplified syntax for creating threads, compared to Java. But it still has the overhead of memory and, often, performance. Coroutines can solve these issues; use coroutines whenever you need to execute some code concurrently in Kotlin.

At this point, you should know how to start a coroutine and how to wait for it to complete, getting its results in the process. We also covered how coroutines are structured and learned about how they interact with dispatchers.

Finally, we touched upon the topic of structured concurrency, a modern idea that helps us prevent resource leaks in concurrent code easily.

In the next chapter, we'll discuss how we can use these concurrency primitives to create scalable and robust systems that suit our needs.

Questions

1. What are the different ways to start a coroutine in Kotlin?
2. With structured concurrency, if one of the coroutines fails, all the siblings will be canceled as well. How can we prevent that behavior?
3. What is the purpose of the **`yield()`** function?