Running Concurrently Using Coroutines

We'll cover the following

In Kotlin coroutines are first-class citizens. They're built into the language, but the convenience functions to work with coroutines are part of a library. Coroutines offer some capabilities that aren't possible with subroutines. They're used in infinite sequences, event loops, and cooperating functions, for example. To learn about the benefits of coroutines, we'll start with an example that runs sequentially and then change the code to run concurrently with coroutines.

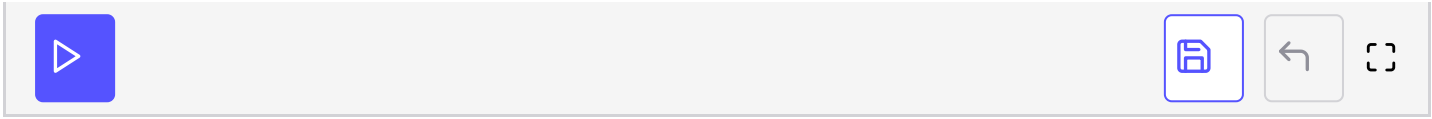
When executing code, you can decide if the code should run sequentially or if it should run concurrently, either within the context of the current execution or as a separate coroutine. Let's explore how we can configure these options.

Starting with sequential execution

Let's start with a piece of code that executes function calls sequentially—this will serve as a building block for introducing coroutines.

```
fun task1() {
  println("start task1 in Thread ${Thread.currentThread()}")
  println("end task1 in Thread ${Thread.currentThread()}")
}

fun task2() {
  println("start task2 in Thread ${Thread.currentThread()}")
  println("end task2 in Thread ${Thread.currentThread()}")
}

println("start")

run {
  task1()
  task2()
```

```
  task2()

  println("called task1 and task2 from ${Thread.currentThread()}")
}

println("done")
```



sequential.kts

Each of the two functions `task1()` and `task2()` print the details about their thread of execution upon entry and before exit. The `run()` function is an extension function of the `Any` class. We invoke `run` and pass a lambda expression as argument to it. Within the lambda we call the two functions and then print a message that we've invoked the functions.

Here's the output from the code:

```
start
start task1 in Thread Thread[main,5,main]
end task1 in Thread Thread[main,5,main]
start task2 in Thread Thread[main,5,main]
end task2 in Thread Thread[main,5,main]
called task1 and task2 from Thread[main,5,main]
done
```

The function calls are executing sequentially, like we expect common programs to do, with `task1()` completing, then `task2()` starting and then running to completion, and, finally, the last line within the lambda passed to `run()` executing. Nothing new there, but the code serves as a starting point for playing with coroutines.

Creating a coroutine

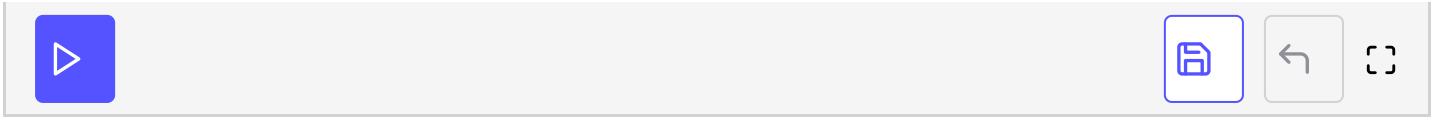
In the previous example, the main thread executed sequentially the code within the lambda passed to `run()`. We'll change the program in such a way that the code in that lambda executes concurrently within a coroutine.

Before we look at the code, download the necessary coroutines extension library kotlinx-coroutines-core-1.2.2.jar file and place it in an easily accessible location on your system. The library is available as a Maven package, and you may download it using Maven or Gradle, as well, instead of manually downloading it. On my

machine, the library is located under the `/opt/kotlin` directory.

Let's change the previous code to run concurrently the code within the lambda:

```
import kotlinx.coroutines.*

fun task1() {
  println("start task1 in Thread ${Thread.currentThread()}")
  println("end task1 in Thread ${Thread.currentThread()}")
}

fun task2() {
  println("start task2 in Thread ${Thread.currentThread()}")
  println("end task2 in Thread ${Thread.currentThread()}")
}

println("start")

runBlocking {
  task1()
  task2()

  println("called task1 and task2 from ${Thread.currentThread()}")
}

println("done")
```



coroutine.kts

You'll find only two differences between this code and the previous sequential version. First is the import from the `kotlinx.coroutines.*` package. Second, we replaced `run()` with `runBlocking()`, which is a function in the `kotlinx.coroutines` package that contains convenience functions and classes to help us program with coroutines. The `runBlocking()` function takes a lambda as an argument and executes that within a coroutine.

The imported package is from an extension library. While coroutines are part of the language's standard library, you'll have to download an additional library to make use of that package, which contains functions to easily create and work with coroutines.

To run the code, specify the location of that coroutines extension jar file to the `kotlinc-jvm` command, like so:

```
kotlinc-jvm -classpath /opt/kotlin/kotlinx-coroutines-core-1.2.2.jar \ -script
 coroutine.kts
```

Replace `/opt/kotlin/` with the correct location of where the jar file is located on your system. On Windows, use `\` instead of `/` for path separator.

Let's take a look at the output of this code:

```
start
start task1 in Thread Thread[main,5,main]
end task1 in Thread Thread[main,5,main]
start task2 in Thread Thread[main,5,main]
end task2 in Thread Thread[main,5,main]
called task1 and task2 from Thread[main,5,main]
done
```

No, you're not reading it wrong; the output of the sequential version of code is the same as the output of the version that uses coroutines. What's the point? you may wonder. Coroutines run code concurrently—thus, the code within the lambda ran in the *main thread* interleaved between the code before the call to `runBlocking()` and the code after the call to that function.

Really?

That's a reasonable response to have when looking at this example. It's understandable that you want to see some action that is different from sequential execution—that's fair; we'll do that next.

Launching a task

Let's launch the two functions, `task1()` and `task2()`, to execute in two different coroutines and then display a message that we've invoked the tasks. Let's see if that shows any difference from the execution of the sequential version. Start with the previous code and make changes only to the two function calls to `task1()` and `task2()`, within the lambda passed to `runBlocking()`, like so:

```
runBlocking {
  launch { task1() }
  launch { task2() }
  println("called task1 and task2 from ${Thread.currentThread()}")
}
println("done")
```

The `launch()` function starts a new coroutine to execute the given lambda, much like the `runBlocking()` function does, except the invoking code isn't blocked for the completion of the coroutine. And, unlike the `runBlocking()` function, the `launch()` function returns a `job`, which can be used to wait on for completion or to cancel the task. Let's take a look at the output of the above version of code:

```
start
called task1 and task2 from Thread[main,5,main]
start task1 in Thread Thread[main,5,main]
end task1 in Thread Thread[main,5,main]
start task2 in Thread Thread[main,5,main]
end task2 in Thread Thread[main,5,main]
done
```

Whew! That's different from the sequential version, albeit only slightly. The message that the two tasks were called is printed right after the line with the `start` message. We then see `task1()` run to completion and then `task2()`, followed by the message `done` at the end.

All the code still executes in the `main` thread, but we can see how the last line within the lambda executed before either `task1()` or `task2()`. That's at least a sign of concurrency, more than in the previous version.

Interleaving calls with suspension points

More action, you demand. The previous example showed a bit of the evidence of concurrency, but let's create an example that will remove all doubt.

Kotlin coroutines library comes with suspension points—a function that will suspend execution of the current task and let another task execute. Calling such a function is like you pointing the microphone toward a colleague, in the middle of your speech, for them to say a few words. There are two functions to achieve this in the `kotlinx.coroutines` library: `delay()` and `yield()`.

The `delay()` function will pause the currently executing task for the duration of milliseconds specified. The `yield()` method doesn't result in any explicit delays. But both these methods will give an opportunity for another pending task to execute.

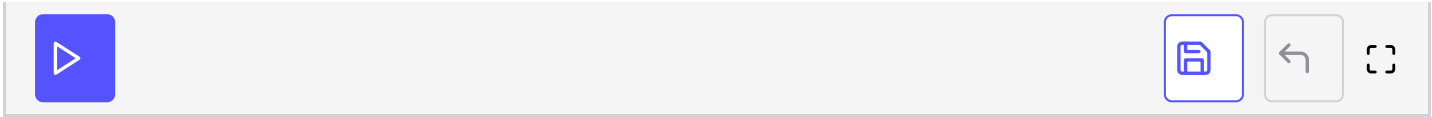
The `yield()` function is like the `nice` command in Unix-like systems. By being nice you may lower your processes priority on these systems. In Kotlin, your task can

be nice, using `yield()`, to other tasks that may have more important things to do.

Let's use `yield()` in both the functions `task1()` and `task2()`. Kotlin will permit the use of suspension points only in functions that are annotated with the `suspend` keyword. Marking a function with `suspend` doesn't automatically make the function run in a coroutine or concurrently, however.

Let's modify the previous code to use `yield()`—the changes are only to the two functions:

```
import kotlinx.coroutines.*

suspend fun task1() {
  println("start task1 in Thread ${Thread.currentThread()}")
  yield()
  println("end task1 in Thread ${Thread.currentThread()}")
}

suspend fun task2() {
  println("start task2 in Thread ${Thread.currentThread()}")
  yield()
  println("end task2 in Thread ${Thread.currentThread()}")
}

println("start")

runBlocking {
  launch { task1() }
  launch { task2() }

  println("called task1 and task2 from ${Thread.currentThread()}")
}

println("done")
```



interleave.kts

We annotate both the functions with the keyword suspend. Then, within each of the functions, right after the first call to `println()`, we call `yield()`. Now, each of the functions will yield the flow of execution to other tasks, if any. Within the lambda passed to the `runBlocking()` function, we launch the two tasks, in two separate coroutines, like we did before. Let's take a look at the output after this change:

```
start
called task1 and task2 from Thread[main,5,main]
```

```
start task1 in Thread Thread[main,5,main]
start task2 in Thread Thread[main,5,main]

end task1 in Thread Thread[main,5,main]
end task2 in Thread Thread[main,5,main]
done
```

Much better—the differences between this output and the output for previous versions are vivid. We see that the `task1()` function executed its first line and yielded the execution flow. Then the `task2()` function stepped in, ran its first line, also in the `main` thread, and then yielded, so `task1()` can continue its execution. This is like working in an office where everyone is nice, highly polite, and very considerate of each other. OK, enough dreaming—let's get back to coding.

The previous example showed how the single thread `main` is running all that code, but the function calls aren't executed sequentially. We can clearly see the interleaving of execution—that served as a good example of concurrent execution.

The examples illustrated the behavior of coroutines, but it leaves us with the question, When will we use them? Suppose we have multiple tasks that can't be run in parallel, maybe due to potential contention of shared resources used by them. Running the tasks sequentially one after the other may end up starving all but a few tasks. Sequential execution is especially not desirable if the tasks are long running or never ending. In such cases, we may let multiple tasks run cooperatively, using coroutines, and make steady progress on all tasks. We can also use coroutines to build an unbounded stream of data—see Creating Infinite Sequences.

In all the examples so far, the execution has been in the `main` thread. In the next lesson, let's see how to dispatch the execution of a coroutine in a different thread.