Debugging Coroutines

We'll cover the following

- Running in debug mode
- Assigning names to coroutines

"Prefer testing over debugging" is a good mantra to follow, but occasionally we have to debug code. Even when we follow good test-driven development practices, we'll have to dig into the execution to look at why things aren't working the way we expect. We'll look at testing coroutines in Chapter 19, Unit Testing with Kotlin, but for now let's figure out a way to find out which coroutines are running our code.

Running in debug mode

Kotlin provides a command-line option `-Dkotlinx.coroutines.debug` to display the details of the coroutine executing a function. When you print the details of a thread, the coroutine that's running in that thread is displayed. Let's use that option on the previous code.

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

When run with the command-line debug option, we see more details in the output than we have seen so far:

```
starting in Thread Thread[main @coroutine#1,5,main]
start task1 in Thread Thread[DefaultDispatcher-worker-1 @coroutine#1,5,main]
end task1 in Thread Thread[DefaultDispatcher-worker-3 @coroutine#1,5,main]
ending in Thread Thread[main @coroutine#1,5,main]
start task2 in Thread Thread[main @coroutine#2,5,main]
end task2 in Thread Thread[main @coroutine#2,5,main]
```

The output shows that `withContext()` didn't start a new coroutine—the code

within `task1()` is running in the same coroutine as the code within `runBlocking()`.

On the other hand, since we used a call to `launch()`, the code within `task2()` is running in a different coroutine.

When the debugging flag is set, Kotlin assigns consecutive identifiers for each of the coroutines it creates. This is useful for quickly looking up log messages from code running within the same coroutines. But, from the debugging point of view, instead of seeing a number like `42` for an identifier, it's better to see a logical name like `searching for meaning of life` for a coroutine that is executing a complex algorithm.

Assigning names to coroutines

We can assign a name by passing an instance of `CoroutineName()` to `runBlocking()` and `launch()`. If we're already passing a context, we can append the name. Let's take a look at how to achieve this by changing the previous code:

```
//...import, task1, and task2 functions like in previous code...
runBlocking(CoroutineName("top")) {
  println("running in Thread ${Thread.currentThread()}")
  withContext(Dispatchers.Default) { task1() }
  launch(Dispatchers.Default + CoroutineName("task runner")) { task2() }
  println("running in Thread ${Thread.currentThread()}")
}
```

The only changes are to the `runBlocking()` call and the `launch()` call. To the `runBlocking()` we passed an instance of `CoroutineName()`. For the ``launch() call, we pass the name in addition to a `CoroutineContext`.

Let's take a look at the output of running the modified code with the command-line debug option:

```
running in Thread Thread[main @top#1,5,main]
start task1 in Thread Thread[DefaultDispatcher-worker-1 @top#1,5,main]
end task1 in Thread Thread[DefaultDispatcher-worker-3 @top#1,5,main]
start task2 in Thread Thread[DefaultDispatcher-worker-3 @task runner#2,5,mai
n]
end task2 in Thread Thread[DefaultDispatcher-worker-3 @task runner#2,5,main]
running in Thread Thread[main @top#1,5,main]
```

In the output, the names of the coroutines are "top" and "task runner" instead of

the mere "coroutine”. The identifier that Kotlin creates for each coroutine is still

preserved and presented in the output. Once again we can see that `withContext()` switched the context of the currently executing coroutine.

Not only can we use coroutines to perform actions asynchronously, but we can also perform computations and get results back on the caller side, as we’ll see in the next lesson.