async and await

We'll cover the following

Why `async` & `await` are used?

The `launch()` function returns a Job object that can be used to await termination of the coroutine or to cancel. But there’s no way to return a result from the coroutine that was started using `launch()`. If you want to execute a task asynchronously and get the response, then use `async()` instead of `launch()`.

The `async()` function takes the same parameters as `launch()`, so we can configure the context and start properties of coroutines that are created using `async()` and `launch()` in the same way. The difference, though, is that `async()` returns a `Deferred<T>` future object which has an `await()` method, among other methods, to check the status of the coroutine, cancel, and so on. A call to `await()` will block the flow of execution but not the thread of execution. Thus, the code in the caller and the code within the coroutine started by `async()` can run concurrently. The call to `await()` will eventually return the result of the coroutine started using `async()`. If the coroutine started using `async()` throws an exception, then that exception will be propagated to the caller through the call to `await()`.

Using `async` & `await`

We’ll use `async()` and `await()` in the next chapter, but here, let’s get a quick taste of these functions. In the code that follows we get the number of cores available on a system asynchronously. The `Dispatchers.Default` argument is optional and, if left out, the coroutine will run in the dispatcher of the scope it inherits. In this example, we have a single-threaded scope and thus the coroutine will run in the same thread as the caller. But when run in a scope of a multi-threaded dispatcher, the coroutine will execute in any of the threads of that dispatcher.

```
import kotlinx.coroutines.*


runBlocking {
  val count: Deferred<Int> = async(Dispatchers.Default) {
    println("fetching in ${Thread.currentThread()}")
    Runtime.getRuntime().availableProcessors()
  }

  println("Called the function in ${Thread.currentThread()}")

  println("Number of cores is ${count.await()}")
}
```

asyncawait.kts

Once the request has been dispatched, the main thread will execute the print statement after the call to async(). The call to await() will wait for the coroutine started by async() to complete. The last print statement will then print the response received from the coroutine. Let's take a look at this in the output:

```
Called the function in Thread[main,5,main]
fetching in Thread[DefaultDispatcher-worker-1,5,main]
Number of cores is 8
```

The output shows the number of cores on my system; your output may vary. We see that the coroutines ran in the `DefaultDispatcher` pool's thread. Try removing the `Dispatchers.Default` argument from the call to `async()`, and run the code again. Notice that the coroutine, in that case, also runs in the `main` thread.

Coroutines run asynchronously, can switch threads, suspend and resume, and yet return the results back to where they are expected. What's the magic that makes this happen? It's time to unveil that secret in the next lesson.