A Peek at Continuations

We'll cover the following

Preserving the state

Methods marked with the suspend annotation may return data. However, coroutines may suspend execution and may switch threads. How in the world does the state get preserved and propagated between threads?

To explore this further, let's take a look at an example that brings out this concern clearly. Instead of creating .kts files, we'll create .kt files so it's easy to compile to Java bytecode and examine.

```
import kotlinx.coroutines.*

class Compute {
  fun compute1(n: Long): Long = n * 2
  suspend fun compute2(n: Long): Long {
    val factor = 2
    println("$n received : Thread: ${Thread.currentThread()}")
    delay(n * 1000)
    val result = n * factor
    println("$n, returning $result: Thread: ${Thread.currentThread()}")
    return result
  }
}
```

Compute.kt

The Compute class has two methods, one marked with `suspend`. The `compute1()` method is simple; it returns the double of the given input. The `compute2()` method does the same thing, except it yields the flow of execution to other pending tasks and may potentially switch threads midway.

Let's create a `main()` function to execute the `compute2()` function, a couple of times, within coroutines.

Compute.kt

UseCompute.kt

All code files are copied to end of the page...

We assign a single expression function to the `main()` function with a call to `runBlocking<Unit>`. So far we used only `runBlocking()` without the parametric `Unit`, but since the return type of `main()` is `Unit`, we have to convey that the call to `runBlocking()` is returning the same. Otherwise we'll get a compilation error. Within the lambda passed to `runBlocking<Unit>()` we create two coroutines and invoke the `compute2()` method in each. Let's take a look at the output of this code:

```
2 received : Thread: Thread[DefaultDispatcher-worker-1,5,main]
1 received : Thread: Thread[DefaultDispatcher-worker-2,5,main]
1, returning 2: Thread: Thread[DefaultDispatcher-worker-2,5,main]
2, returning 4: Thread: Thread[DefaultDispatcher-worker-4,5,main]
```

The coroutine that's running `compute2()` with an input value of 2 switched threads. The first part, before the call to `delay()`, is executing in one thread and the second part, after the `delay()`, is running in a different thread. But the value of `factor` got passed correctly from before the `delay()` to after. It's not magic, it's continuations.

What are continuations?

Using continuations, which are highly powerful data structures, programs can capture and preserve the state of execution in one thread and restore them when needed in another thread. Programming languages that support continuations generate special code to perform this seamlessly for the programmers.

At the risk of overly simplifying, think of a continuation as a closure—see Closures and Lexical Scoping—that captures the lexical scope. We can imagine how the code after a suspension point may be wrapped into a closure and saved. Then that closure may be invoked whenever the function execution has to be resumed.

To see how this unfolds at the compilation level, we can use `javap -c` to explore the bytecode of the compiled `Compute` class. Let's take a look at the details for only the two `compute...` methods.

```
public final long compute1(long);
public final java.lang.Object compute2(long,

  kotlin.coroutines.Continuation<? super java.lang.Long>);
```

Looking at the source code, the signature of both `compute1()` and `compute2()` were identical except for the `suspend` annotation. At the bytecode level, though, they are worlds apart. The `compute1()` function doesn't have anything unexpected. It takes as input a `long` and returns a value of the same type. The `compute2()` function, on the other hand, looks a lot different. Even though in the source code `compute2()` only took one parameter, in the compiled version we see that it takes two parameters: `long` and `Continuation<?superLong>`. Furthermore, it returns `Object` instead of `long`. The `Continuation` encapsulates the results of the partial execution of the function so that the result can be delivered to the caller using the `Continuation` callback. The compiler engages continuations to implement the machinery of coroutines, to switch context between executing different tasks, to switch threads, and to restore states. The net result is that as programmers, we can focus on using continuations and leave all the heavy lifting to the compiler.

We've had a glimpse of what's going on under the hood when we use coroutines. In the next lesson, let's focus on a practical application of coroutines.

Code Files Content !!!

⤋ ⤋

```
------------------------------------------------------------------------
|  Compute.kt [1]
------------------------------------------------------------------------


import kotlinx.coroutines.*

class Compute {
  fun compute1(n: Long): Long = n * 2
  suspend fun compute2(n: Long): Long {
    val factor = 2
    println("$n received : Thread: ${Thread.currentThread()}")
    delay(n * 1000)
    val result = n * factor
```

```
    println("$n, returning $result: Thread: ${Thread.currentThread()}")
    return result
  }
}


-------------------------------------------------------------------------
|  UseCompute.kt [1]
-------------------------------------------------------------------------


import kotlinx.coroutines.*

fun main() = runBlocking {
  val compute = Compute()

  launch(Dispatchers.Default) {
    compute.compute2(2)
  }
  launch(Dispatchers.Default) {
    compute.compute2(1)
  }
}




*************************************************************************************
```