Coroutine Context and Threads

We'll cover the following

The call to the `launch()` and `runBlocking()` functions resulted in the coroutines executing in the same thread as the caller's coroutine scope. That's the default behavior of these function since they carry a coroutine context from their scope. You may, however, vary the context and the thread of execution of the coroutines where you like.

Explicitly setting a context

You may pass a CoroutineContext to the launch() and runBlocking() functions to set the execution context of the coroutines these functions start.

The value of `Dispatchers.Default` for the argument of type `CoroutineContext` instructs the coroutine that is started to execute in a thread from a `DefaultDispatcher` pool. The number of threads in this pool is either 2 or equal to the number of cores on the system, whichever is higher. This pool is intended to run computationally intensive tasks.

The value of `Dispatchers.IO` can be used to execute coroutines in a pool that is dedicated to running IO intensive tasks. That pool may grow in size if threads are blocked on IO and more tasks are created.

`Dispatchers.Main` can be used on Android devices and Swing UI, for example, to run tasks that update the UI from only the `main` thread.

To get a feel for how to set the context for `launch()`, let's take the previous example and make a change to one of the `launch()` calls, like so:

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

After this change, the code in `task1()` will run in a different thread than the rest of the code that still runs in the `main` thread. We can verify this in the output—the output you see may be slightly different since the order of multiple threads running in parallel is nondeterministic:

```
start
start task1 in Thread Thread[DefaultDispatcher-worker-1,5,main] end task1 in T
hread Thread[DefaultDispatcher-worker-2,5,main] called task1 and task2 from Th
read[main,5,main]
start task2 in Thread Thread[main,5,main]
end task2 in Thread Thread[main,5,main]
done
```

In this case, the code within the lambda passes to `runBlocking()`, and the code within `task2()` runs concurrently, but the code within `task1()` is running in parallel. Coroutines may execute concurrently or in parallel, depending on their context.

Running in a custom pool

You know how to set a context explicitly, but the context we used in the previous example was the built-in `DefaultDispatcher`. If you'd like to run your coroutines in your own single thread pool, you can do that as well. Since you'll have a single thread in the pool, the coroutines using this context will run concurrently instead of in parallel. This is a good option if you're concerned about resource contention among the tasks executing as coroutines.

To set a single thread pool context, we first have to create a single thread executor. For this we can use the JDK `Executors` concurrency API from the `java.util.concurrent` package. Once we create an executor, using the JDK library, we can use Kotlin's extension functions to get a `CoroutineContext` from it using an `asCoroutineDispatcher()` function. Let's give that a shot.

First, import the necessary package:

```
// single.kts
import kotlinx.coroutines.*
import java.util.concurrent.Executors
//...task1 and task2 function definitions as before...
```

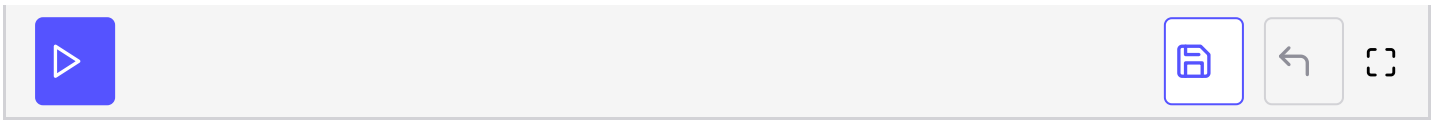
You may be tempted to create a dispatcher from the single thread executor and pass that directly to `launch()`, but there's a catch. If we don't close the executor, our program may never terminate. That's because there's an active thread in the executor's pool, in addition to main, and that will keep the JVM alive. We need to keep an eye on when all the coroutines complete and then close the executor. But that code can become hard to write and error prone. Thankfully, there's a nice `use()` function that will take care of those steps for us. The use() function is akin to the *try-with-resources* feature in Java. The code to use the context can then go into the lambda passed to the `use()` function, like so:

```
Executors.newSingleThreadExecutor().asCoroutineDispatcher().use { context ->
  println("start")

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

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

  println("done")
}
```



single.kts

We first created an executor using the `Executors.newSingleThreadExecutor()` method of the JDK and then obtained a `CoroutineContext` using the `asCoroutineDispatcher()` extension function added by the `kotlinx.coroutines` library. Then we call the `use()` method and passed a lambda to it. Within the lambda we obtain a reference to the context, using the `context` variable, and pass that to the first call to `launch()`. The coroutines started by this call to `launch()`—that is, the execution of `task1()`—will run in the single thread pool managed by the executor we created. When we leave the lambda expression, the `use()` function will close the executor, knowing that all the coroutines have completed.

Let's take a look at the output and confirm that the code in `task1()` is running in the pool we created instead of the `DefaultDispatcher` pool:

```
start
start task1 in Thread Thread[pool-1-thread-1,5,main]
end task1 in Thread Thread[pool-1-thread-1,5,main]
called task1 and task2 from Thread[main,5,main]
start task2 in Thread Thread[main,5,main]
end task2 in Thread Thread[main,5,main]
done
```

If instead of using a single thread pool, you'd like to use a pool with multiple threads—say, as many threads as the number of cores on the system—you may change the line:

```
Executors.newSingleThreadExecutor().asCoroutineDispatcher().use { context ->
```

That change looks like this:

```
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors())
  .asCoroutineDispatcher().use { context ->
```

Now the coroutines that use this context will run in this custom pool with as many threads as the number of cores on the system running this code.

Switching threads after suspension points

What if you want a coroutine to start in the context of the caller but switch to a different thread after the suspension point? In other words, as long as the task involves quick computations, you may want to do that in the current thread, but in the instance we hit a time-consuming operation, we may want to delegate that to run on a different thread. We can achieve this by using the `CoroutineContext` argument along with a `CoroutineStart` argument.

To run the coroutine in the current context, you may set the value of the second optional argument of `launch()` to **DEFAULT**, which is of type `CoroutineStart`. Alternatively, use **LAZY** to defer execution until an explicit `start()` is called, *_ATOMIC* to run in a non-cancellable mode, and `UNDISPATCHED` to run initially in the current context but switch threads after the suspension point.

current context but switch threads after the suspension point.

Let's modify the call to the first `launch()` call in the previous code to use our own thread pool and also the `UNDISPATCHED` option for the second argument. Each of the parameters of `launch()` has default values, so we don't have to pass the first argument `context` in order to pass the second argument `start`. If we want to pass only the value for `start`, we can use the named argument feature. To illustrate this, let's use named arguments for both the first and the second argument in the code:

```
import kotlinx.coroutines.*
import java.util.concurrent.Executors

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()}")
}

Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors())
  .asCoroutineDispatcher().use { context ->
  println("start")

  runBlocking {
    @UseExperimental(ExperimentalCoroutinesApi::class)
    launch(context = context, start = CoroutineStart.UNDISPATCHED) { task1() }
    launch { task2() }

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

  println("done")
}
```

coroutinestart.kts

The `CoroutineStart.UNDISPATCHED` option is an experimental feature in the `kotlinx.coroutines` library, and to use it we have to annotate the expression with `@UseExperimental`, as we see in the previous code. Since we're using an experimental feature, we also have to set a command-line flag when calling the `-Xuse-experimental`.

```
kotlinc-jvm -Xuse-experimental=kotlin.Experimental \ -classpath /opt/kotlin/ko
tlinx-coroutines-core-1.2.2.jar \-script coroutinestart.kts
```

Go ahead and execute the code and take a look at the output:

```
start
start task1 in Thread Thread[main,5,main]
end task1 in Thread Thread[pool-1-thread-1,5,main] called task1 and task2 fro
m Thread[main,5,main] start task2 in Thread Thread[main,5,main]
end task2 in Thread Thread[main,5,main]
done
```

We see that the execution of `task1()` started in the `main` thread instead of in the `pool`-1's thread. But once the execution reached the suspension point, `yield()`, the execution switched over to the `pool`-1's thread, which is in the context specified to the `launch()` function.

Changing the coroutine context

The `runBlocking()` and `launch()` functions provide a nice way to set the context of a new coroutine, but what if you want to run a coroutine in one context and then change the context midway? Kotlin has a function for that: `withContext()`.

Using this function you can take a part of code and run it in an entirely different context than the rest of the code in the coroutine.

Let's create an example to illustrate this:

```
//...import, task1, and task2 functions like in previous code...

runBlocking {
  println("starting in Thread ${Thread.currentThread()}")
  withContext(Dispatchers.Default) { task1() }

  launch { task2() }

  println("ending in Thread ${Thread.currentThread()}")
}
```

withcontext.kts

Here's the output of this code:

```
starting in Thread Thread[main,5,main]
```

```
starting in Thread Thread[main,5,main]
start task1 in Thread Thread[DefaultDispatcher-worker-1,5,main]
end task1 in Thread Thread[DefaultDispatcher-worker-1,5,main]
ending in Thread Thread[main,5,main]
start task2 in Thread Thread[main,5,main]
end task2 in Thread Thread[main,5,main]
```

The output shows that all code except the code within the lambda provided to `withContext()` is running in the `main` thread. The code called from within the lambda provided to `withContext()`, however, is running in the thread that's part of the provided context.

We see that the threads are different, but did `withContext()` really change the context of the currently executing coroutine or did it merely create an entirely new coroutine? It changed the context, but how can we tell? "Trust me," is not a good response for such questions—we want to see it to believe it.

In the next lesson, it's time to dissect the execution.