Cancellations and Timeouts

We'll cover the following

Coroutines can be cancelled, which will stop the code within the coroutine from further execution. Coroutine cancellations aren't related to the thread terminations that we're used to in Java. Cancellations are lightweight and have effect across the hierarchy of coroutines that share context.

Both the Job object, which is returned by `launch()`, and the `Deferred<T>` object, which is returned by `async()`, have a `cancel()` and a `cancelAndJoin()` method. We can use these methods to explicitly cancel a coroutine, but there's a catch. A coroutine is cancelled only if it is currently in a suspension point. If a coroutine is busy executing code, it won't receive the cancellation notice and may not bail out. Let's discuss this further and find ways to deal with this behavior.

Kotlin provides structured concurrency, where coroutines that share a context form a hierarchical relationship. Coroutines that belong to a hierarchy follow some rules and exhibit prescribed behaviors:

- A coroutine is considered to be a child of another coroutine if it shares the context of the coroutine that creates it.
- A parent coroutine completes only after all its children complete.
- Cancelling a parent cancels all its children.
- A coroutine that has entered a suspension point may receive a `CancellationException` thrown from the suspension point.

- A coroutine that is busy, that's not in a suspension point, may check an `isActive` property to see if it was cancelled when it was busy.
- If a coroutine has resources to clean up, that needs to happen within the `finally` block within the coroutine.
- An unhandled exception will result in the cancellation of a coroutine.
- If a child coroutine fails, it will result in the parent coroutine cancelling and thus result in cancellation of the coroutine's siblings. You can modify this behavior using supervised jobs that make cancellation unidirectional, from parent to children.

That's a lot of information in one place, but you can refer back to this when working with coroutines. Let's examine these ideas with some examples.

Cancelling coroutines

If we don't care for the completion of a task that we started in a coroutine, we may cancel it by calling `cancel()` or `cancelAndJoin()` on the `Job` or the `Deferred<T>` instance. However, the coroutine may not budge to that command right away. If the coroutine is busy executing, the command doesn't interrupt it. On the other hand, if the coroutine is parked in a suspension point, like in a call to `yeild()`, `delay()`, or `await()`, then it will be interrupted with a `CancellationException`.

When designing a coroutine, keep the previous constraints in mind. If you're performing a long-running computation, you may want to structure it so that you take a break frequently to check if the `isActive` property of the coroutine is still `true`. If you find it to be `false`, you can break out of the computation, honoring the request to cancel.

Sometimes you may not have the capability to check the `isActive` property because the function you internally call within the coroutine is blocking and doesn't involve a suspension point. In this case, you may consider delegating that blocking call to another coroutine—a level of indirection—and await on it, thus introducing a suspension point around your blocking call.

Let's explore these options with an example. We'll verify the behavior of the code that plays nicely with cancellation and also code that doesn't. This will give us a

clear view of how we should approach the design of code with a coroutine, from the cancellation point of view.

We'll first create a `compute()` function that runs in one of two modes: if the parameter passed in is true, it will check the `isActive` property in the middle of a long-running computation; if that parameter is `false`, it will run wild for a long time. Since we need access to the `isActive` property, the code needs to run in the context of a coroutine. For this, we'll wrap the code within the `compute()` function inside a call to `coroutineScope()` that carries the scope from the caller.

Here's the code for the `compute()` function:

```
import kotlinx.coroutines.*

suspend fun compute(checkActive: Boolean) = coroutineScope {
  var count = 0L
  val max = 10000000000

  while (if (checkActive) { isActive } else (count < max)) {
    count++
  }
  if (count == max) {
    println("compute, checkActive $checkActive ignored cancellation")
  } else {
    println("compute, checkActive $checkActive bailed out early")
  }
}
```

cancelandsuspension.kts

If the computation is long running, we may check `isActive` from time to time. But if in the coroutine we call a long-running function that can't be interrupted, then the coroutine won't get interrupted either. To illustrate this and to find a workaround, let's create a `fetchResponse()` function.

```
val url =  "http://httpstat.us/200?sleep=2000"

fun getResponse() = java.net.URL(url).readText()

suspend fun fetchResponse(callAsync: Boolean) = coroutineScope {
  try {
    val response = if (callAsync) {
      async { getResponse() }.await()
    } else {
      getResponse()
    }
```

```
    println(response)
  } catch(ex: CancellationException) {
    println("fetchResponse called with callAsync $callAsync: ${ex.message}")
  }
}
```

cancelandsuspension.kts

The `fetchResponse()` function sends a request to the mentioned URL that will return the HTTP code specified, `200` in this example, after a delay provided as a value to the `sleep` parameter, in milliseconds. If the `callAsync` parameter is `false`, then the call to the URL is made synchronously, thus blocking the caller and disallowing cancellation. But if that parameter is `true`, the call to the URL is made asynchronously, thus permitting immediate cancellation of the coroutine while it awaits.

Both the `compute()` function and the `fetchResponse()` function may or may not respond to cancellation, depending on how they're called. This is true, even though the former represents a long-running computation-intensive operation while the latter is performing a long-running IO operation.

Let's use these two functions within a few coroutines, let them run for a second, and then issue a stern `cancel` command. We may either use the cancel() method and then a call to `join()` or we may combine the two into one call with `cancelAndJoin()`. Let's give that a try:

```
runBlocking {
  val job = launch(Dispatchers.Default) {
    launch { compute(checkActive = false) }
    launch { compute(checkActive = true) }
    launch { fetchResponse(callAsync = false) }
    launch { fetchResponse(callAsync = true) }
  }

  println("Let them run...")
  Thread.sleep(1000)
  println("OK, that's enough, cancel")
  job.cancelAndJoin()
}
```

cancelandsuspension.kts

The `compute()` method call started with `checkActive` equal to true will terminate as soon as the cancel command is issued, since it's checking the `isActive` property.

The `compute()` method call with `checkActive` equal to `false` will ignore the cancel command from its parent coroutine and will behave like a typical teenager with a headset on. Likewise, the `fetchResponse()` function will exhibit a similar behavior. We can see this in the output:

```
Let them run...
OK, that's enough, cancel
compute, checkActive true bailed out early
fetchResponse called with callAsync true: Job was cancelled 200 OK
compute, checkActive false ignored cancellation
```

The version of the calls that saw the cancel message, either because of the exception thrown at it or by checking for `isActive`, quit early. The versions that had the headsets on–that is, the coroutines that ignored their cancellation message–ran to completion.

When creating coroutines, verify that the code handles cancellation correctly.

Do not disturb

Sometimes we don't want part of a task to be interrupted. For example, you may be performing a critical operation, and stopping it in the middle may have catastrophic consequences. To handle this, there's a special context.

If your critical task doesn't have any suspension points, you don't have to worry. When a coroutine is busy running that non-interruptible code, no cancellation command will affect it. You don't have to write any extra code to prevent cancellation in this case.

But if your critical code contains a suspension point, like a `yield()`, `delay()`, or `await()`, then you may want to convey that you're in the middle of a critical section and don't want to be interrupted. The `withContext(NonCancellable)` function call is like hanging a Do Not Disturb sign outside the door. Let's see this in action with an example:

```
import kotlinx.coroutines.*

suspend fun doWork(id: Int, sleep: Long) = coroutineScope {
  try {
    println("$id: entered $sleep")
    delay(sleep)
    println("$id: finished nap $sleep")

    withContext(NonCancellable) {
```

```
      println("$id: do not disturb, please")
      delay(5000)
      println("$id: OK, you can talk to me now")
    }

    println("$id: outside the restricted context")
    println("$id: isActive: $isActive")
  } catch(ex: CancellationException) {
    println("$id: doWork($sleep) was cancelled")
  }
}
```

donotdistrub.kts

The `doWork()` function runs in the context of the callers coroutine. First, the function delays the coroutine for the given amount of time and is cancellable during this phase. Second, it enters the `NonCancellable` context and the delay within this phase isn't cancellable. Any interruption during this time will be ignored, but the `isActive` property will be changed if a cancellation were to happen. Finally, in this code we print the `isActive` property.

Let's call this function a couple of times within separate coroutines, sleep for two seconds, cancel the coroutines's parent, and wait for the parent to complete.

```
runBlocking {
  val job = launch(Dispatchers.Default) {
    launch { doWork(1, 3000) }
    launch { doWork(2, 1000) }
  }

  Thread.sleep(2000)
  job.cancel()
  println("cancelling")
  job.join()
  println("done")
}
```

donotdistrub.kts

We can see from the output shown below that the coroutine is cancelled if the cancellation command occurs when the execution is outside of the `NonCancellable` context. But if the coroutine is currently inside this context, it isn't cancelled and it runs uninterrupted.

```
2: entered 1000
1: entered 3000
```

```
2: finished nap 1000
2: do not disturb, please

cancelling
1: doWork(3000) was cancelled
2: OK, you can talk to me now
2: outside the restricted context
2: isActive: false
done
```

When you leave the do-not-disturb `NonCancellable` context, you may check the `isActive` property to see if the coroutine was interrupted and decide on the course of action accordingly.

Bidirectional cancellation

When a coroutine runs into an unhandled non-cancellation exception, it's automatically cancelled. When a coroutine cancels, its parent is cancelled. When a parent is cancelled, all its children are then cancelled. All this behavior is automatically built-in to how coroutines cooperate. We can see this in the following example. The `fetchResponse()` function makes requests to the httpstat URL with the given code. Let's take a look at the code before discussing further.

```
import kotlinx.coroutines.*
import java.net.URL

suspend fun fetchResponse(code: Int, delay: Int) = coroutineScope {
  try {
    val response = async {
      URL("http://httpstat.us/$code?sleep=$delay").readText()
    }.await()

    println(response)
  } catch(ex: CancellationException) {
    println("${ex.message} for fetchResponse $code")
  }
}
runBlocking {
  val handler = CoroutineExceptionHandler { _, ex ->
    println("Exception handled: ${ex.message}")
  }

  val job = launch(Dispatchers.IO + SupervisorJob() + handler) {
    launch { fetchResponse(200, 5000) }
    launch { fetchResponse(202, 1000) }
    launch { fetchResponse(404, 2000) }
  }

  job.join()
}
```

cancellationbidirectional

If the given code is `404`, the service request will fail with an exception. Since the function doesn't handle that exception, the coroutine running the function will cancel. This will result in the eventual cancellation of all the siblings of this coroutine that haven't completed at this time.

```
202 Accepted
Parent job is Cancelling for fetchResponse 200
Exception handled: http://httpstat.us/404?sleep=2000
```

The coroutine that executes the request for code `202` finishes in about one second, before the failure of the `404` request, which will take about two seconds due to the `sleep` parameter passed to the URL. When the coroutine that runs the request for code `404` fails, it brings down the coroutine that is running the asynchronous request for code `200`.

The default behavior is that when a coroutine cancels, it brings down the house. This may not be desirable all the time. We can alter how coroutines communicate cancellation using a supervisory job. We've seen this pop up a few times already in the code; it's time to give it a closer look.

Supervisory job

Much like the way we passed the `handler` to a coroutine, we may also pass an instance, for example, `supervisor`, of a `SupervisorJob` like `launch(coroutineContext + supervisor)`. Alternatively, we may wrap the supervised children into a `supervisorScope` call. In either case, the supervised children won't propagate their cancellation to their parents. But if a parent were to cancel, then its children will be cancelled.

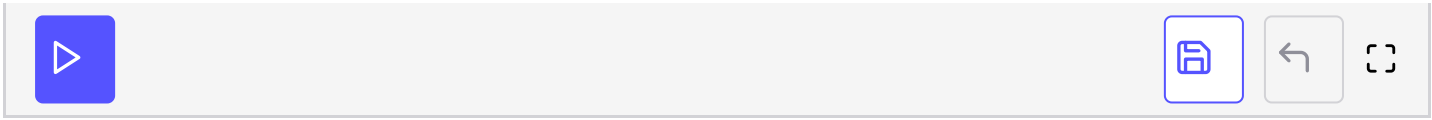
Let's change the previous code to use a supervisor and observe the behaviors described.

```
runBlocking {
    val handler = CoroutineExceptionHandler { _, ex ->
    println("Exception handled: ${ex.message}")
}

    val job = launch(Dispatchers.IO + handler) {
```

```
        supervisorScope {
            launch { fetchResponse(200, 5000) }
            launch { fetchResponse(202, 1000) }

            launch { fetchResponse(404, 2000) }
        }
    }
    Thread.sleep(4000)
    println("200 should still be running at this time")
    println("let the parent cancel now")
    job.cancel()
    job.join()
}
```



We wrapped the three nested calls to `launch()` within a `supervisorScope()` call. Then we let the coroutine run for four seconds. During this time, the coroutine that's processing code `202` should complete and the coroutine that's processing code `404` should cancel. The coroutine that's processing code `200` should be unaffected during this time. If we leave it alone, it will run to completion, not affected by its siblings. But at the end of 4 seconds we cancel the parent, which causes the child coroutine to cancel. We can see this behavior in the output:

```
202 Accepted
Exception handled: http://httpstat.us/404?sleep=2000
200 should still be running at this time
let the parent cancel now
Job was cancelled for fetchResponse 200
```

Use supervised coroutines only when you want to configure a clear top-down hierarchy of independent child tasks. When one of the children fails, in this case, you still want its siblings to continue unaffected. But if the parent is cancelled, you want to shut down the tasks. On the other hand, if you want the tasks to be fully cooperative, then rely upon the default behavior of coroutines.

Programming with timeouts

As a simple rule, both in life and programming, we should never do anything without timeouts. Coroutines are suitable for long-running cooperative tasks, but we don't want to wait an undue amount of time or forever. We can program timeouts easily when using coroutines.

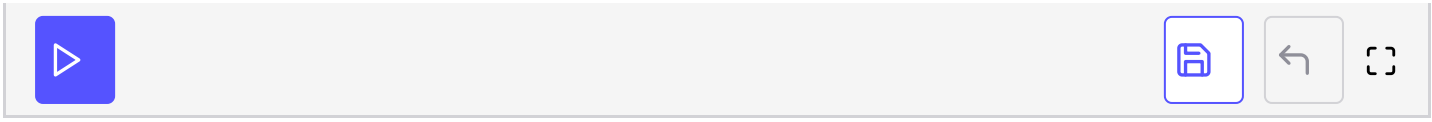
If a coroutine takes more time than the given allowance to complete, it'll receive a `TimeoutCancellationException`, which is a subclass of `CancellationException`. In

effect, the task that takes longer than the permitted time to complete will be

cancelled due to timeout, and all the rules we discussed about cancellation apply to this type of cancellation also.

Let's replace the code within `runBlocking()` in the previous example with the following. We wrap the inner calls to `launch()` with a call to `withTimeout()` and specify a time allowance of 3000 milliseconds. Then we wait for the coroutines to finish.

```
runBlocking {
    val handler = CoroutineExceptionHandler { _, ex ->
        println("Exception handled: ${ex.message}")
    }
    val job = launch(Dispatchers.IO + handler) {
        withTimeout(3000) {
            launch { fetchResponse(200, 5000) }
            launch { fetchResponse(201, 1000) }
            launch { fetchResponse(202, 2000) }
        }
    }
    job.join()
}
```



Let's run the code and take a look at the output:

```
201 Created
202 Accepted
Timed out waiting for 3000 ms for fetchResponse 200
```

The coroutines that took shorter than the given 3 seconds (and thankfully didn't run into any network errors) completed successfully. The coroutine that took its sweet time got cancelled.

The next lesson concludes the discussion for this chapter.