Exception Handling

We'll cover the following

Many things can go wrong when making requests to web services, updating databases, accessing a file, and so on. When delegating a task to run asynchronously, we have to be defensive and handle failures gracefully. How we deal with exceptions depends on how we start a coroutine—using `launch()` or `async()`.

> **Be Mindful of Structured Concurrency**
>
> Unless a scope is explicitly specified, coroutines run in the context and scope of their parent coroutines—called structured concurrency, where the hierarchical structure of coroutines matches the structure of code. This is a good thing as it's easier to monitor and manage the execution of coroutines that we start. A coroutine doesn't complete until all its children complete. But this also has some ramifications on how coroutines cooperate with each other and how failures are handled.
>
> By default, when a coroutine fails with an exception, it results in the failure of the parent coroutine as well. Later in this chapter we'll take a closer look at the behavior of coroutines in the context of structured concurrency and how we can prevent a child coroutine from cancelling a parent coroutine using a SupervisorJob context. In the examples in this section, we'll use this context to prevent a child coroutine from cancelling its parent when an exception occurs.

launch and exceptions

If we use `launch()`, then the exception won't be received on the caller side—it's a fire and forget model, though you may optionally wait for the coroutine to

complete. To illustrate this behavior and find a way to gracefully handle exceptions, let's query the airport status using some invalid airport codes.

LaunchErr.kt

Airport.kt

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

We used two valid airport codes and two invalid codes. Instead of using `async()`, we're using `launch()` in this example to see how it deals with exceptions. The `launch()` function returns a `Job` object that represents the coroutine it starts. We use that object to wait for the coroutine to complete either successfully or with failure. Then we query the `isCancelled` property of `Job` to check if the job completed successfully or it was cancelled due to failure.

Let's take a look at the output of running this code. Note that the output you see on your system may not exactly match the one shown here:

```
LAX delay: false
SEA delay: true
Exception in thread "DefaultDispatcher-worker-1" Exception in ...
```

That's sad—the `try-catch` we placed around the calls to `launch()` and the calls to `join()` had no effect. Instead, we see an unpleasant termination of the program with an exception message spilled on the console. The reason for this behavior is that coroutines that are started using `launch()` don't propagate exceptions to their caller. That's the nature of `launch()`, but we gotta do better when programming with this function.

If you're using `launch()`, then make sure to set up an exception handler. An exception handler `CoroutineExceptionHandler`, if registered, will be triggered with the details of the context and the exception. Let's create a handler and register it with the `launch()` calls.

LaunchErrHandle.kt

Airport.kt

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

Airport.kt

This version of code has only two changes, compared to the previous version. First, we create a handler which prints the details of the context of the coroutine that failed, along with the exception details of the failure. Second, in the call to `launch()`, we register the handler, along with a name for the coroutine for easy identification. Let's run this version and see how the program copes with the failures.

```
Caught: CoroutineName(PD-) Unable to instantiate Airport
Caught: CoroutineName(SF-) Unable to instantiate Airport
SEA delay: true
LAX delay: false
Cancelled: false
Cancelled: true
Cancelled: true
Cancelled: false
```

That's much better. The exceptions are handled gracefully by the registered handler. Instead of printing on the console, we can choose to log the error, send a notification to the support, play sad music… the options are endless once we have a handler of the failure. And the program doesn't quit ungracefully; instead it proceeds to display the output of whether the job is completed or cancelled.

The output also shows one other behavior of coroutine we haven't discussed—if a coroutine fails with an unhandled exception, then the coroutine is cancelled. We'll discuss cancellation further in Cancellations and Timeouts.

async and exceptions

`launch()` doesn't propagate the exception to its caller, so we should register an exception handler when using `launch()`. The `async()` function, however, returns a `Deferred<T>` instance, which will carry the exception and deliver to the caller when `await()` is called. It makes no sense to register an exception handler to `async()` if the handler, if registered, will be ignored.

Instead of using `launch()` with invalid airport codes, let's use `async()` to see how the exceptions are handled differently by the two functions. We handle the exception using `try-catch` around the call to `await()`, using exception-handling syntax that we're all familiar with.

AsyncErr.kt

Airport.kt

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

For the first iteration, we use `map()` and the functional style. But for the second iteration over the list of `Deferred<T>`, we use the imperative style `for`-loop instead of the functional style `forEach()`. Even though `forEach()` may be used for this iteration as well, the imperative style is a better choice here. If we use `forEach()`, then the lambda, with all the code to handle the exception, won't be concise and expressive. As we discussed in Chapter 11, Functional Programming with Lambdas, avoid creating multiline, verbose lambdas where possible.

Let's take a look at the output to see how `async()` dealt with the exception.

```
LAX false
Error: Unable to instantiate Airport
Error: Unable to instantiate Airport
SEA true
```

The output shows what we expect—`async()` and `await()` not only make it easier to make asynchronous calls, they also deal well with exceptions.

When working with the previous examples, we got a hint about how unhandled exceptions cancel coroutine. A coroutine may be cancelled intentionally when we no longer care for the task, it may be cancelled due to failure, or it may be forced to cancel by a parent coroutine. We also may want to put a timeout on how long a task takes. All these sound intriguing —let's examine them in the next lesson.

Code Files Content !!!

⇓ ⇓
⇓

⤋

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


import kotlinx.coroutines.*

fun main() = runBlocking {
  try {
    val airportCodes = listOf("LAX", "SF-", "PD-", "SEA")

    val jobs: List = airportCodes.map { anAirportCode ->
      launch(Dispatchers.IO + SupervisorJob()) {
        val airport = Airport.getAirportData(anAirportCode)
        println("${airport?.code} delay: ${airport?.delay}")
      }
    }

    jobs.forEach { it.join() }
    jobs.forEach { println("Cancelled: ${it.isCancelled}") }
  } catch(ex: Exception) {
    println("ERROR: ${ex.message}")
  }
}


-------------------------------------------------------------------------
|  Airport.kt [1]
-------------------------------------------------------------------------


import java.net.URL
import com.beust.klaxon.*

class Weather(@Json(name = "Temp") val temperature: Array)

class Airport(
  @Json(name = "IATA") val code: String,
  @Json(name = "Name") val name: String,
  @Json(name = "Delay") val delay: Boolean,
  @Json(name = "Weather") val weather: Weather) {

  companion object {
    fun getAirportData(code: String): Airport? {
      val url = "https://soa.smext.faa.gov/asws/api/airport/status/$code"
      return Klaxon().parse(URL(url).readText())
    }
  }
}




*********************************************************************************


-------------------------------------------------------------------------
```

```
|  LaunchErrHandle.kt [2]
-------------------------------------------------------------------------



import kotlinx.coroutines.*

fun main() = runBlocking {
  val handler = CoroutineExceptionHandler { context, ex ->
    println(
      "Caught: ${context[CoroutineName]} ${ex.message?.substring(0..28)}")
  }

  try {
    val airportCodes = listOf("LAX", "SF-", "PD-", "SEA")

    val jobs: List = airportCodes.map { anAirportCode ->
      launch(Dispatchers.IO + CoroutineName(anAirportCode) +
        handler + SupervisorJob()) {
        val airport = Airport.getAirportData(anAirportCode)
        println("${airport?.code} delay: ${airport?.delay}")
      }
    }

    jobs.forEach { it.join() }
    jobs.forEach { println("Cancelled: ${it.isCancelled}") }
  } catch(ex: Exception) {
    println("ERROR: ${ex.message}")
  }
}


-------------------------------------------------------------------------
|  Airport.kt [2]
-------------------------------------------------------------------------



import java.net.URL
import com.beust.klaxon.*

class Weather(@Json(name = "Temp") val temperature: Array)

class Airport(
  @Json(name = "IATA") val code: String,
  @Json(name = "Name") val name: String,
  @Json(name = "Delay") val delay: Boolean,
  @Json(name = "Weather") val weather: Weather) {

  companion object {
    fun getAirportData(code: String): Airport? {
      val url = "https://soa.smext.faa.gov/asws/api/airport/status/$code"
      return Klaxon().parse(URL(url).readText())
    }
  }
}




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

```
-------------------------------------------------------------------------
|  AsyncErr.kt [3]
-------------------------------------------------------------------------



import kotlin.system.*
import kotlinx.coroutines.*

fun main() = runBlocking {
  val airportCodes = listOf("LAX", "SF-", "PD-", "SEA")

  val airportData = airportCodes.map { anAirportCode ->
    async(Dispatchers.IO + SupervisorJob()) {
      Airport.getAirportData(anAirportCode)
    }
  }

  for (anAirportData in airportData) {
    try {
      val airport = anAirportData.await()

      println("${airport?.code} ${airport?.delay}")
    } catch(ex: Exception) {
      println("Error: ${ex.message?.substring(0..28)}")
    }
  }
}


-------------------------------------------------------------------------
|  Airport.kt [3]
-------------------------------------------------------------------------



import java.net.URL
import com.beust.klaxon.*

class Weather(@Json(name = "Temp") val temperature: Array)

class Airport(
  @Json(name = "IATA") val code: String,
  @Json(name = "Name") val name: String,
  @Json(name = "Delay") val delay: Boolean,
  @Json(name = "Weather") val weather: Weather) {

  companion object {
    fun getAirportData(code: String): Airport? {
      val url = "https://soa.smext.faa.gov/asws/api/airport/status/$code"
      return Klaxon().parse(URL(url).readText())
    }
  }
}




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