

Programming Asynchronously

We'll cover the following ^

- Starting sequentially
- Making asynchronous

When programming modern applications we often have to make calls to remote services, update databases, perform searches, and the list goes on. Many of these tasks aren't instantaneous. To improve program efficiency we may want to execute such operations asynchronously, in a non-blocking manner. Coroutines are intended to exactly solve that problem.

Starting sequentially

Let's create a program to go out and get the weather details to see if there are any delays at certain airports. As a first step, we'll create an `Airport` class that will hold the data, along with a `Weather` class to hold the temperature. We'll use the [Klaxon library](#) to parse the JSON response from the Federal Aviation Administration (FAA) airport status web service.

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

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

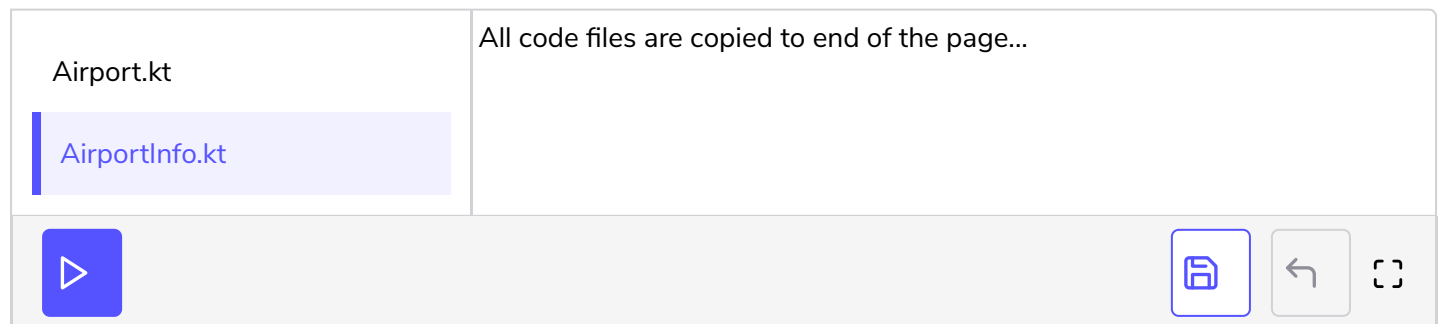
class Airport(
    @Json(name = "IATA") val code: String,
    @Json(name = "Name") val name: String,
    @Json(name = "Delay") val delay: Boolean,
    @Json(name = "Weather") val weather: Weather) {

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

In the `Airport` class we use the `@Json` annotation to map the properties in the JSON response to the properties in our class. In the `getAirportData()` method we fetch the data, extract the text response, and parse the JSON content to create an instance of `Airport`.

Given a list of airport codes, let's first fetch the data sequentially using the method just described. This will soon help us compare the sequential, synchronous code for this program with the asynchronous version.

We'll loop through the list of airports and fetch the airport status information, one at a time, and print the output in a perusable format.



In addition to printing the status details, we also measure the time the code takes to run, using the `measureTimeMillis()` function, which is a nice little convenience function in the Kotlin standard library. Let's take a look at the output of this code:

```
Code Temperature      Delay
LAX  68.0 F (20.0 C)  false
SFO  50.0 F (10.0 C)  true
PDX  56.0 F (13.3 C)  false
SEA  55.0 F (12.8 C)  true
Time taken 2112 ms
```

Nice weather, but fifty percent of the airports queried have delays—that gives us confidence the data represents reality and is reliable. The program took a bit more than 2 seconds to run. You may observe a different speed, depending on the network performance at the time of the run.

Making asynchronous

Each of the calls to the `getAirportData()` method in the previous code is blocking calls. When we make the call for “LAX” the program waits until that request is





completed before making the request for the next code “SFO” to be processed. We can make those calls non-blocking; that is, we don’t have to wait for the request for “LAX” to be completed before making a request for all the subsequent codes. We can fire off multiple requests and complete the first loop before any of the calls to `getAirportData()` have completed, or even started.

We want the `main()` function to block and wait for the asynchronous executions to complete. For this reason, we’ll use `runBlocking()` around the entire code in the body of `main()`.

To start a non-blocking call to `Airport`’s `getAirportData()`, we obviously can’t use `runBlocking()`. And `launch()` is intended to perform actions, not return results. So the `async()` function is the right choice here.

If we call `async()` directly within the iteration, then the coroutines will run in the currently executing coroutine context. Thus, the coroutine started will also run in the `main` thread. The calls to `getAirportData()` will be non-blocking and concurrent. In other words, the `main` thread won’t block and wait, but the execution of the calls will be interleaved for execution in the main thread. That gives us half the benefit—the non-blocking part—but won’t give us any performance advantage. Since each call will run on the same thread, the time it’ll take for the code to complete will be about the same as the sequential run. We have to do better.

We can start the coroutine by telling `async()` to use a different context, and thus a different pool of threads. We can easily do this by passing a `CoroutineContext`, like `Dispatchers.IO`, to `async()`. Let’s rewrite the previous code and use these ideas.

<div>AirportInfoAsync.kt</div> <div>Airport.kt</div>	All code files are copied to end of the page...
<div></div> <div>  </div>	

The structure of the asynchronous code in the file `AirportInfoAsync.kt` is identical to the structure of the code in the file `AirportInfo.kt`—this is a blessing since the asynchronous code is as easy to reason and understand as the synchronous version.

While the code structures are the same between the two versions, there are some

While the code structures are the same between the two versions, there are some differences. First, instead of a `List<Airport>` we create a `List<Deferred<Airport?>`

`>>`, because the result of a call to `async()` is an instance of `Deferred<T>`.

Within the first iteration we called the `getAirportData()` method inside of the lambda passed to the `async()` call. The call to `async()` returns immediately with the result of `Deferred<Airport?>`, which we store in the list.

In the second iteration, we go over the `Deferred<Airport?>` list and invoke `await()` to get the data. Calls to `await()` will block the flow of execution but won't block the thread of execution. Had we used `async()` directly without the argument, the `main` thread would be executing the coroutines after reaching the call to `await()`. In our case, since we used `async()` with the `Dispatchers.IO` context, the `main` thread can chill, get a beverage, and stare at the stars, while the worker threads in the `Dispatchers.IO` pool take care of making the call to the web service.

Let's take a look at the output to see how the asynchronous program did in comparison to the sequential version.

```
Code Temperature      Delay
LAX  68.0 F (20.0 C)   false
SFO  50.0 F (10.0 C)   true
PDX  56.0 F (13.3 C)   false
SEA  55.0 F (12.8 C)   true
Time taken 1676 ms
```

The airport data is the same between the two versions, but may be different if the two versions are run at different times. The asynchronous version took about half a second less to complete.

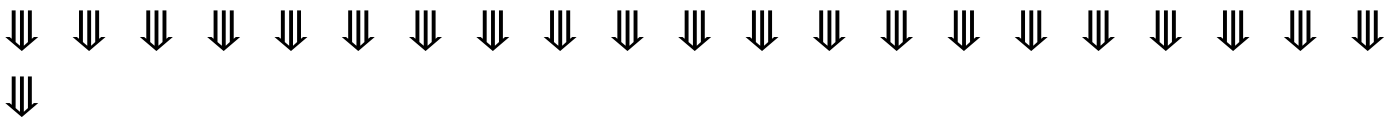
Use caution when you observe the time differences, however, since network speeds are often unreliable. If you see that the asynchronous version is occasionally slower than the sequential version, don't panic. Play with the asynchronous version to get a deeper understanding. Place print statements before the call to `getAirportData()` method to print the thread information. Notice how those calls are invoked in parallel when the program is run. Change `async(Dispatchers.IO)` to `async()` and observe how the calls run in the `main` thread, and the impact that has on the time the program takes.

Both the sequential and the asynchronous versions of the program focus on the happy path, assuming all things go well. Many things could go wrong though. The

happy path, assuming all things go well. Many things could go wrong, though. The web service may intermittently fail, the network may drop, the airport code the user provided may be invalid or may not be supported by the web service. The ways that Murphy's law may strike are without limit. We have to program defensively, and coroutines are exceptional on that front.

In the next lesson, let's explore how coroutines handle exceptions.

Code Files Content !!!



```
-----  
|  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())  
        }  
    }  
}
```

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

```
import kotlin.system.*  
  
fun main() {  
    val format = "%-10s%-20s%-10s"  
    println(String.format(format, "Code", "Temperature", "Delay"))  
}
```

```

val time = measureTimeMillis {
    val airportCodes = listOf("LAX", "SFO", "PDX", "SEA")

    val airportData: List =
        airportCodes.mapNotNull { anAirportCode ->
            Airport.getAirportData(anAirportCode)
        }

    airportData.forEach { anAirport ->
        println(String.format(format, anAirport.code,
            anAirport.weather.temperature.get(0), anAirport.delay))
    }
}

println("Time taken $time ms")
}

```

| AirportInfoAsync.kt [2]

```

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

fun main() = runBlocking {
    val format = "%-10s%-20s%-10s"
    println(String.format(format, "Code", "Temperature", "Delay"))

    val time = measureTimeMillis {
        val airportCodes = listOf("LAX", "SFO", "PDX", "SEA")

        val airportData: List< > =
            airportCodes.map { anAirportCode ->
                async(Dispatchers.IO) {
                    Airport.getAirportData(anAirportCode)
                }
            }

        airportData
            .mapNotNull { anAirportData -> anAirportData.await() }
            .forEach { anAirport ->
                println(String.format(format, anAirport.code,
                    anAirport.weather.temperature.get(0), anAirport.delay))
            }
    }

    println("Time taken $time ms")
}

```

| 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())
        }
    }
}
```
