Mobile Application Development

Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology
http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Coroutines in Kotlin

Agenda

- Background
- Introduction
- The Terminology
- Sample Code
- Case Study

Agenda

- Background
- Introduction
- The Terminology
- Sample Code
- Case Study

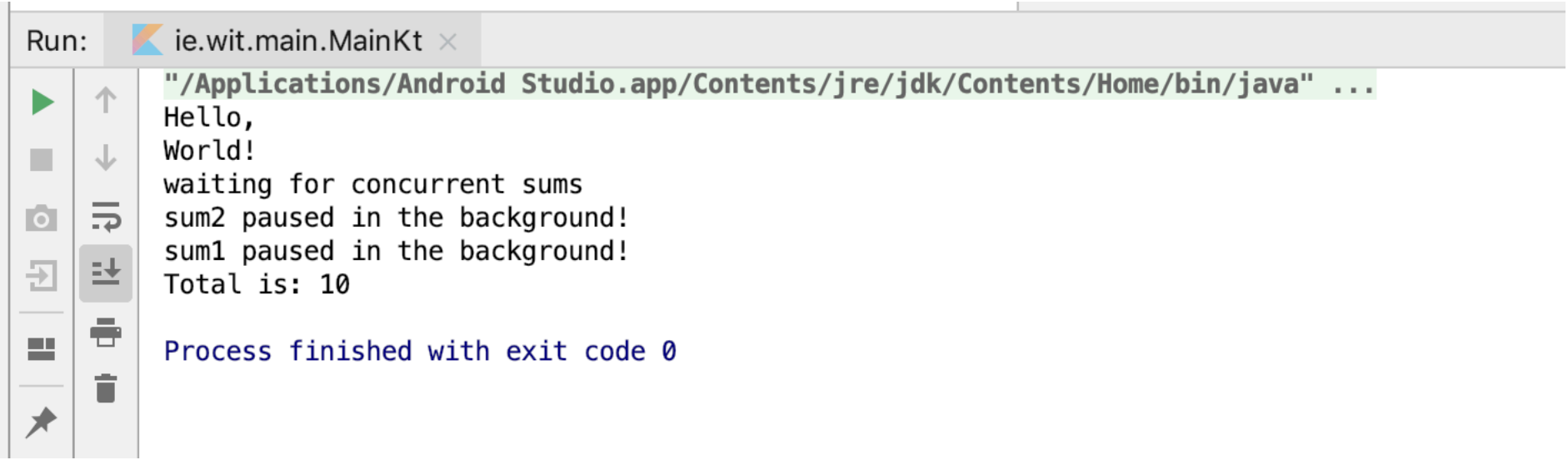
Sample Code

Recap / Intro

- coroutines are (more or less) light-weight threads. They are launched with **launch** (a *coroutine builder)* in a context of some **CoroutineScope**.
- In the next few examples we are launching new coroutines in the **GlobalScope**, meaning that the lifetime of the new coroutine is limited only by the lifetime of the whole application.

```kotlin
fun test1() {
    GlobalScope.launch {  // launch new coroutine in background and continue
        delay(3000L) // non-blocking delay for 3 seconds (default time unit is ms)
        println("World!") // print after delay
        val sum1 = async { // non blocking sum1
            delay(2000L)
            println("sum1 paused in the background!")
            2 + 2
        }
        val sum2 = async { // non blocking sum2
            delay(1000L)
            println("sum2 paused in the background!")
            3 + 3
        }
        println("waiting for concurrent sums")
        val total = sum1.await() + sum2.await() // execution stops until both sums
        println("Total is: $total")
    }
    println("Hello,")     // main thread continues while coroutine executes
    Thread.sleep(6000L)   // block main thread for 6 seconds to keep JVM alive
}
```

Output of ‘test1’



```
fun test2() {
    val keyword = "Ireland"
    GlobalScope.launch {  // launch new coroutine in background and continue
        delay(2000L) // non-blocking delay for 2 seconds (default time unit is ms)
        println("World!") // print after delay
        val wResult = async { wikipedia(keyword) }
        println("Wikipedia replied: ${wResult.await()}")
    }
    println("Hello,")     // main thread continues while coroutine executes
    Thread.sleep(3000L)   // block main thread for 3 seconds to keep JVM alive
}
```

Output of ‘test2’

Output of ‘test2’ (with Thread.sleep(2000L))

Writing Suspending Functions

Sequential by default

Assume that we have two suspending functions defined elsewhere that do something useful like some kind of remote service call or computation. We just pretend they are useful, but actually each one just delays for a second for the purpose of this example:

```
suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

Writing Suspending Functions

We use a normal sequential invocation, because the code in the coroutine, just like in the regular code, is *sequential* by default. The following example demonstrates it by measuring the total time it takes to execute both suspending functions:

```
val time = measureTimeMillis {
    val one = doSomethingUsefulOne()
    val two = doSomethingUsefulTwo()
    println("The answer is ${one + two}")
}
println("Completed in $time ms")
```

```
The answer is 42
Completed in 2010 ms
```

Target platform: JVM Running on kotlin v. 1.3.50

Writing Suspending Functions

What if there are no dependencies between invocations of `doSomethingUsefulOne` and `doSomethingUsefulTwo` and we want to get the answer faster, by doing both *concurrently*? This is where async comes to help.

```
val time = measureTimeMillis {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    println("The answer is ${one.await() + two.await()}")
}
println("Completed in $time ms")
```

```
The answer is 42
Completed in 1024 ms
```

Target platform: JVM Running on kotlin v. 1.3.50

Writing Suspending Functions

Conceptually, async is just like launch. It starts a separate coroutine which is a light-weight thread that works concurrently with all the other coroutines. The difference is that `launch` returns a Job and does not carry any resulting value, while `async` returns a Deferred — a light-weight non-blocking future that represents a promise to provide a result later. You can use `.await()` on a deferred value to get its eventual result, but `Deferred` is also a `Job`, so you can cancel it if needed.

Writing Suspending Functions

Optionally, async can be made lazy by setting its `start` parameter to CoroutineStart.LAZY. In this mode it only starts the coroutine when its result is required by await, or if its `Job`'s start function is invoked. Run the following example:

```
val time = measureTimeMillis {
    val one = async(start = CoroutineStart.LAZY) { doSomethingUsefulOne() }
    val two = async(start = CoroutineStart.LAZY) { doSomethingUsefulTwo() }
    // some computation
    one.start() // start the first one
    two.start() // start the second one
    println("The answer is ${one.await() + two.await()}")
}
println("Completed in $time ms")
```

```
The answer is 42
Completed in 1032 ms
```

Writing Suspending Functions

So, here the two coroutines are defined but not executed as in the previous example, but the control is given to the programmer on when exactly to start the execution by calling start. We first start `one`, then start `two`, and then await for the individual coroutines to finish.

Note that if we just call await in `println` without first calling start on individual coroutines, this will lead to sequential behavior, since await starts the coroutine execution and waits for its finish, which is not the intended use-case for laziness. The use-case for `async(start = CoroutineStart.LAZY)` is a replacement for the standard `lazy` function in cases when computation of the value involves suspending functions.

Writing Suspending Functions

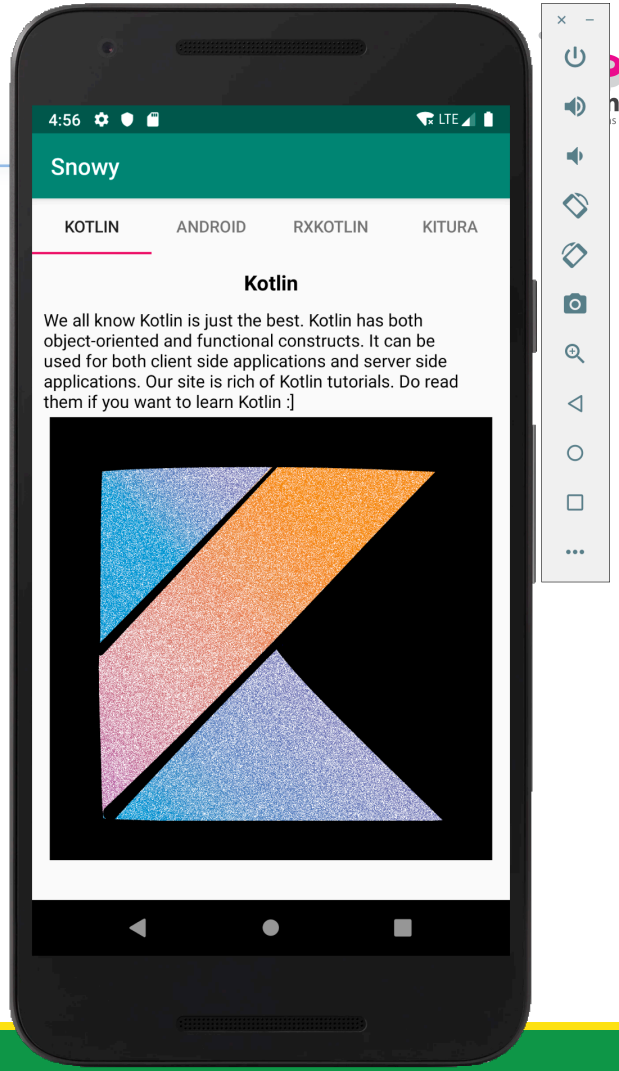
```
val time = measureTimeMillis {
    val one = async(start = CoroutineStart.LAZY) { doSomethingUsefulOne() }
    val two = async(start = CoroutineStart.LAZY) { doSomethingUsefulTwo() }
    // some computation
    //one.start() // start the first one
    //two.start() // start the second one
    println("The answer is ${one.await() + two.await()}")
}
println("Completed in $time ms")
```

```
The answer is 42
Completed in 2077 ms
```

Case Study - Snowy

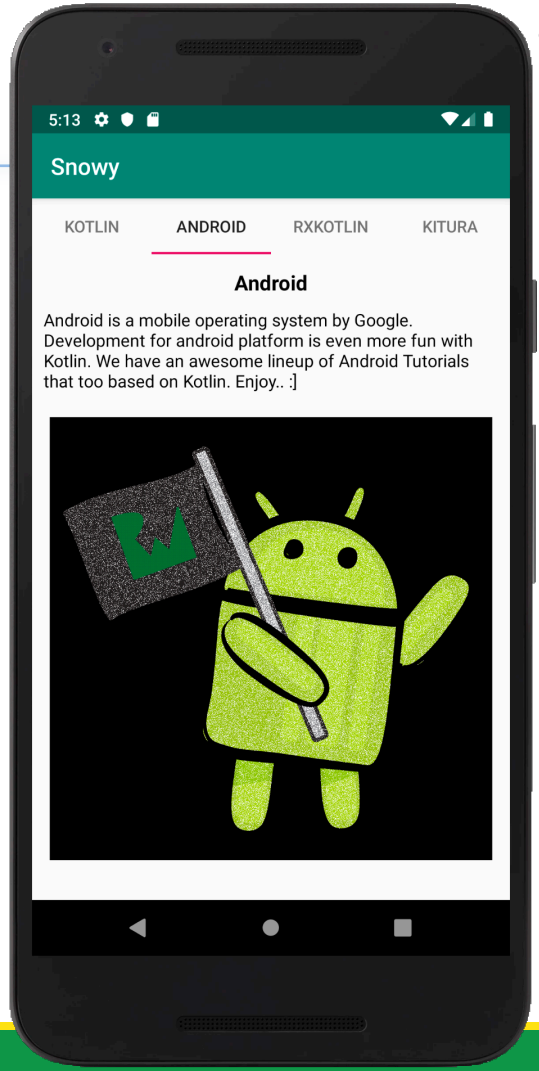
What is Snowy?

- Snowy is a photo editing app, which will allow you to download an image and then apply a snow filter to that image.
- To download the images, and process them, you'll need to perform asynchronous tasks.



What is Snowy?

- Snowy is a photo editing app, which will allow you to download an image and then apply a snow filter to that image.
- To download the images, and process them, you'll need to perform asynchronous tasks.



References

Sources: https://antonis.me/2018/12/12/an-introduction-to-kotlin-coroutines/
https://github.com/antonis/CoroutinesExamples
https://proandroiddev.com/kotlin-coroutines-channels-csp-android-db441400965f
https://superkotlin.com/coroutines/
https://www.codementor.io/jimmy_chuks/introduction-to-coroutines-xr674rfpo
https://www.raywenderlich.com/1423941-kotlin-coroutines-tutorial-for-android-getting-started
https://android.jlelse.eu/kotlin-coroutines-threads-concurrency-and-parallelism-101-78a56e09d373
https://kotlinlang.org/docs/reference/coroutines/composing-suspending-functions.html


Thanks.