Coroutines

Week 4

Kotlin

Kotlin, as a language, provides only minimal low-level APIs in its standard library to enable various other libraries to utilize coroutines. Unlike many other languages with similar capabilities, async and await are not keywords in Kotlin and are not even part of its standard library. Moreover, Kotlin's concept of suspending function provides a safer and less error-prone abstraction for asynchronous operations than futures and promises.

Coroutines basics

- A coroutine is an instance of suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.
- Coroutines can be thought of as light-weight threads, but there is a number of important differences that make their real-life usage very different from threads.

Your first coroutine

```
import kotlinx.coroutines.*

fun main() = runBlocking {      // this: CoroutineScope
    launch {        // launch a new coroutine and continue
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
```

Suspend function

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch { doWorld() }
    println("Hello")
}

suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

Suspending functions can be used inside coroutines just like regular functions, but their additional feature is that they can, in turn, use other suspending functions (like delay in this example) to suspend execution of a coroutine.

Scope builder

- In addition to the coroutine scope provided by different builders, it is possible to declare your own scope using the coroutineScope builder. It creates a coroutine scope and does not complete until all launched children complete.
- runBlocking and coroutineScope builders may look similar because they both wait for their body and all its children to complete. The main difference is that the runBlocking method blocks the current thread for waiting, while coroutineScope just suspends, releasing the underlying thread for other usages. Because of that difference, runBlocking is a regular function and coroutineScope is a suspending function

Scope builder

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    doWorld()
}
suspend fun doWorld() = coroutineScope {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
```

Scope builder and concurrency

- A coroutineScope builder can be used inside any suspending function to perform multiple concurrent operations. Let's launch two concurrent coroutines inside a doWorld suspending function:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    doWorld()
    println("Done")
}
```

```
suspend fun doWorld() = coroutineScope
{    launch {
        delay(2000L)
        println("World 2")
    }
    launch {
        delay(1000L)
        println("World 1")
    }
    println("Hello")
}
```

An explicit job

- A launch coroutine builder returns a Job object that is a handle to the launched coroutine and can be used to explicitly wait for its completion. For example, you can wait for completion of the child coroutine and then print "Done" string:

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    val job = launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
    job.join()
    println("Done")
}
```

Coroutines are light-weight

Coroutines are less resource-intensive than JVM threads. Code that exhausts the JVM's available memory when using threads can be expressed using coroutines without hitting resource limits

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    repeat(100_000) {
        launch {
            delay(5000L)
            print(".")
        }
    }
}
```

Cancelling coroutine execution

- In a long-running application you might need fine-grained control on your background coroutines.

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    val job = launch {
        repeat(1000) { i ->
            println("job: I'm sleeping $i ...")
        delay(500L)
        }
    }
    delay(1300L)
    println("main: I'm tired of waiting!")
    job.cancel()
    job.join()
    println("main: Now I can quit.")
}
```

Cancellation is cooperative

- Coroutine cancellation is cooperative. A coroutine code has to cooperate to be cancellable. All the suspending functions in kotlinx.coroutines are cancellable. They check for cancellation of coroutine and throw CancellationException when cancelled. However, if a coroutine is working in a computation and does not check for cancellation, then it cannot be cancelled.

Cancellation is cooperative

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    val startTime = System.currentTimeMillis()
    val job = launch(Dispatchers.Default) {
        var nextPrintTime = startTime
        var i = 0
        while (i < 5) {
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("job: I'm sleeping ${i++} ...")
                nextPrintTime += 500L
            }
        }
    }
    delay(1300L)
    println("main: I'm tired of waiting!")
    job.cancelAndJoin()
    println("main: Now I can quit.")
}
```

Making computation code cancellable

- There are two approaches to making computation code cancellable. The first one is to periodically invoke a suspending function that checks for cancellation. There is a yield function that is a good choice for that purpose. The other one is to explicitly check the cancellation status.

Making computation code cancellable

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    val startTime = System.currentTimeMillis()
    val job = launch(Dispatchers.Default) {
        var nextPrintTime = startTime
        var i = 0
        while (isActive) {
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("job: I'm sleeping ${i++} ...")
                nextPrintTime += 500L
            }
        }
    }
    delay(1300L)
    println("main: I'm tired of waiting!")
    job.cancelAndJoin()
    println("main: Now I can quit.")
}
```

Closing resources with finally

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    val job = launch {
        try {
        repeat(1000) { i ->
            println("job: I'm sleeping $i ...")
            delay(500L)
        }
        } finally {
            println("job: I'm running finally")
        }
    }
    delay(1300L)
    println("main: I'm tired of waiting!")
    job.cancelAndJoin()
    println("main: Now I can quit.")
}
```

Run non-cancellable block

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    val job = launch {
        try {
            repeat(1000) { i ->
            println("job: I'm sleeping $i ...")
            delay(500L)
            }
        } finally {
            withContext(NonCancellable) {
                println("job: I'm running finally")
                delay(1000L)
                println("job: And I've just delayed for 1 sec because I'm non-cancellable")
            }
        }
    }
    delay(1300L)
    println("main: I'm tired of waiting!")
    job.cancelAndJoin()
    println("main: Now I can quit.")
}
```

Timeout

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    withTimeout(1300L) {
        repeat(1000) { i ->
        println("I'm sleeping $i ...")
        delay(500L)
        }
    }
}
```

Concurrent using async

```
import kotlinx.coroutines.*
import kotlin.system.*
fun main() = runBlocking<Unit> {
    val time = measureTimeMillis {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
}
suspend fun doSomethingUsefulOne(): Int {
    delay(1000L)
    return 13
}
suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L)
    return 29
}
```

Dispatchers and threads

The coroutine context includes a coroutine dispatcher that determines what thread or threads the corresponding coroutine uses for its execution. The coroutine dispatcher can confine coroutine execution to a specific thread, dispatch it to a thread pool, or let it run unconfined.

```
import kotlinx.coroutines.*
fun main() = runBlocking<Unit> {
    launch { // context of the parent, main runBlocking coroutine
        println("main runBlocking : I'm working in thread ${Thread.currentThread().name}")
    }
    launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
        println("Unconfined : I'm working in thread ${Thread.currentThread().name}")
    }
    launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher
        println("Default : I'm working in thread ${Thread.currentThread().name}")
    }
    launch(newSingleThreadContext("MyOwnThread")) { // will get its own new thread
        println("newSingleThreadContext: I'm working in thread ${Thread.currentThread().name}")
    }
}
```