Kotlin Coroutines

Ryszard Szwajlik

```
git clone https://github.com/RyszardSzwajlik/coroutines-workshop.git
```

Agenda

Presentation

1. What are coroutines
2. Let’s create the first coroutine
3. Threads management
4. Blocking code
5. Flows
6. Spring integration
7. Read more

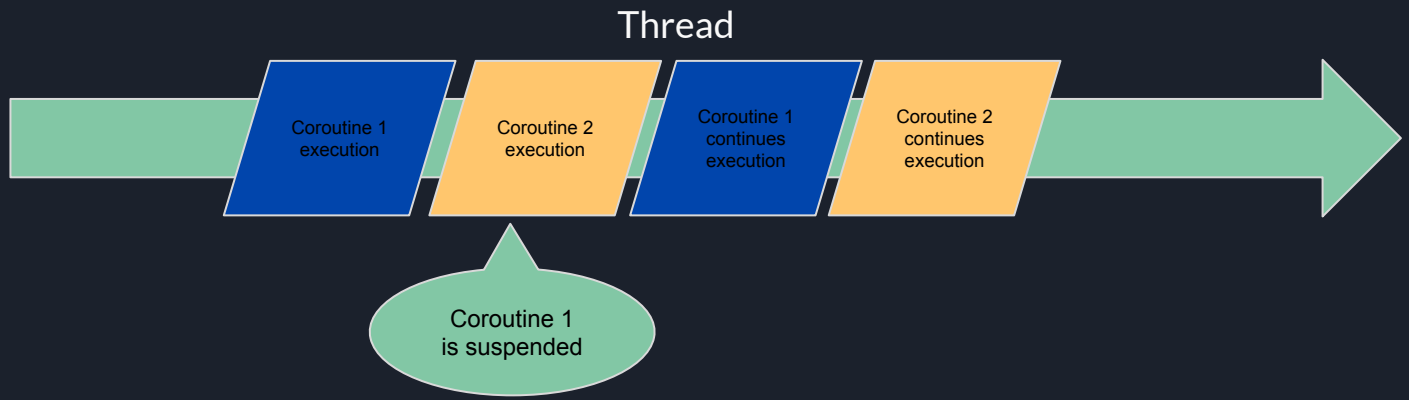
What are coroutines?

- Components that provide a way for achieving concurrency
- Lightweight threads or sub-programs
- Their execution can be suspended

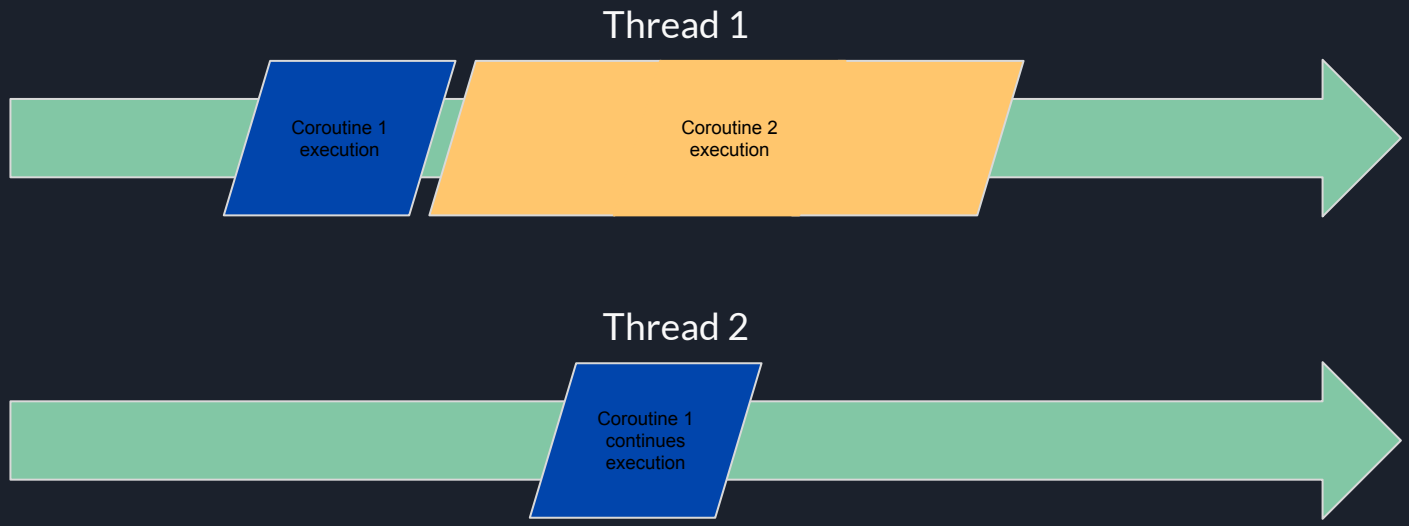
Coroutines are lightweight threads.
By lightweight, it means that creating coroutines doesn't allocate new threads.
Instead, they use predefined thread pools and smart scheduling for the purpose of which task to execute next and which tasks later.

The official documentation

What are coroutines?



What are coroutines?



Creating coroutines

Coroutines1.kt

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() = runBlocking {
    launch {
        delay(1000)
        println("world")
    }
    println("hello")
}
```

Creating coroutines

Coroutines1.kt

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() = runBlocking {
    launch {
        delay(1000)
        println("world")
    }
    println("hello")
}
```

- Entrypoint to coroutines world
- Creates a new scope for coroutines execution (will be covered in next slides)

Creating coroutines

Coroutines1.kt

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() = runBlocking {
    launch {
        delay(1000)
        println("world")
    }
    println("hello")
}
```

- Creates a new coroutine
- Runs the coroutine concurrently

Creating coroutines

Coroutines1.kt

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() = runBlocking {
    launch {
        delay(1000)
        println("world")
    }
    println("hello")
}
```

- Suspends function execution for a specific time
- Allows other coroutines to run and use the underlying thread for their code

Creating coroutines - exercise 1

1. What are names of threads running the coroutine?
2. Launch 1 000 coroutines, what are the names now?
3. Launch 20 000 coroutines, what is the memory and CPU consumption?
4. Launch 20 000 standard threads with `Thread.sleep()`, what is the memory and CPU consumption?

Creating coroutines - structured concurrency

Coroutines2.kt

```
fun main() = runBlocking { // CoroutineScope
    val job1 = launch {
        delay(500)
        println("world")

        val job2 = launch {
            delay(1000)
            println("from coroutines")
        }
        job2.invokeOnCompletion { println("job2 completed") }
    }
    job1.invokeOnCompletion { println("job1 completed") }
    println("hello")
}
```

- Outer scope can only be completed when all inside coroutines are completed
- // todo about leaks

Creating coroutines - exercise 2

1. What is the output of the current implementation?
2. Extract job2 outside of job1. What is the output now?
3. Extract the body of a `launch` function. What new keyword did you notice?

Creating coroutines - suspending - tangible example

Coroutines3.kt

```
fun main() {
    val sequence = buildList {
        println("one")
        add(1)

        println("two")
        add(2)

        println("three")
        add(3)
    }

    for (value in sequence) {
        println("The value $value")
    }
}
```

- What is the result of the above function?

Creating coroutines - suspending - tangible example

Coroutines3.kt

```
fun main() {
    val sequence = buildList {
        println("one")
        add(1)

        println("two")
        add(2)

        println("three")
        add(3)
    }

    for (value in sequence) {
        println("The value $value")
    }
}
```

- What is the result of the above function?
- What would be the result if `add()` was a suspending function?

Creating coroutines - suspending - tangible example

Coroutines3.kt

```
fun main() {
    val sequence = buildList { // coroutine equivalent: sequence { …
}
        println("one")
        add(1) // coroutine equivalent: yield(...)

        println("two")
        add(2)

        println("three")
        add(3)
    }

    for (value in sequence) {
        println("The value $value")
    }
}
```

- What is the result of the above function?
- What would be the result if `add()` was a suspending function?

Creating coroutines - exercise 3

1. Change the implementation of `Coroutines3.kt` to use suspending functions. What is the result?

Creating coroutines - suspending - tangible example

- Suspend keyword instructs the compiler to convert the function to be asynchronous
- Suspend keyword instructs the compiler we are in coroutines scope so other suspending functions can be called

Coroutines4.kt

```
class Coroutines4 {
    suspend fun getName(name: String): String {
        return name
    }
}
```

Thread management - dispatchers

- Are part of the context
- Determine what thread or threads the corresponding coroutine uses for its execution
- Available dispatchers:
  - *Dispatchers.Default*
  - *Dispatchers.Unconfined*
  - *newSingleThreadContext("MyOwnThread")*
  - *newFixedThreadPoolContext(4, "test")*
  - And a few more

```
val context = Dispatchers.Unconfined
launch(context) {
  ...
}
```

Dispatchers - exercise 1

1. Open `Coroutines5.kt` and add code that runs defined task 10x in coroutines
2. Run the same code using:
   a. *`Dispatchers.Unconfined`*
   b. *`newSingleThreadContext("MyOwnThread")`*
   c. *`newFixedThreadPoolContext(4, "test")`*

How the application behaves depending on the context?

Coroutines5.kt

```
fun main() {
    suspend fun task(taskId: Int) {
        println("start task $taskId in Thread ${Thread.currentThread().name}")
        delay(100)
        println("end task $taskId in Thread ${Thread.currentThread().name}")
    }

    // code goes here

}
```

```
val context = Dispatchers.Unconfined
launch(context) {
    ...
}
```

Dispatchers - exercise 2

1. Notice newFixedThreadPoolContext(...) is marked as an obsolete API. Redesign the solution to use API recommended in documentation:
   `Executors.newFixedThreadPool( 4).asCoroutineDispatcher()`

Launching blocking code asynchronously

```
fun <T> CoroutineScope.async(
    context: CoroutineContext =
EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T>
```

Example

```
val databaseResultDeferred: Deferred<Result> =  async {
    blockingRepository.findById(1)
}
val dbResult: Result = databaseResultDeferred.await()
```

Launching blocking code asynchronously - exercise 1

1. Complete `coroutines6.kt` *todos*.
   Create two asynchronous calls to a function that waits 2s blocking and then prints “Hello” and “world”.

```
fun <T> CoroutineScope.async(
    context: CoroutineContext =
EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T>
```

Example

```
val databaseResultDeferred: Deferred<Result> = async {
    blockingRepository.findById(1)
}
val dbResult: Result = databaseResultDeferred.await()
```

Launching blocking code asynchronously - exercise 2

1. How much time does it take to print the whole script? Why is it so slow?
2. Modify the script so it works faster.

```
fun <T> CoroutineScope.async(
    context: CoroutineContext =
EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T>
```

Example

```
val databaseResultDeferred: Deferred<Result> =  async {
    blockingRepository.findById(1)
}
val dbResult: Result = databaseResultDeferred.await()
```

Flows - Returning multiple values asynchronously

- Using the `List<>` result type, means we can only return all the values at once
- To represent the stream of values that are being computed asynchronously, we can use a `Flow<>`
- Implement many commonly known higher order functions: `filter`, `map`, `reduce` etc.

Coroutines7.kt

```
fun calculatePrices(): Flow<Int> = flow { // flow builder
    for (i in 1..5) {
        delay(100) // pretend we are doing some db call to get a price
        emit(i) // emit next value
    }
}

fun main() = runBlocking {
    // Launch a concurrent coroutine to check if the main thread is blocked
    launch {
        for (k in 1..5) {
            println("I'm not blocked $k")
            delay(100)
        }
    }
    // Collect the flow
    calculatePrices().collect { value -> println(value) }
}
```

Flows - exercise 1

1. Open `Coroutines7.kt` and run the application.
2. Modify the application to print only even values
3. transform is a higher order function for transforming one flow into another.
   The argument is result of the previous flow and the result is another flow.
   Emit additional value using the transform function that returns the original and doubled values. Example result:

```
The original price is 2
Doubled price is 4
The original price is 4
Doubled price is 8
```

Spring integration

- Spring provides support for coroutines
- Remember about using reactive versions of libraries (r2dbc, webflux etc.)
- There are many helpers for reactor defined in package `org.jetbrains.kotlinx:kotlinx-coroutines-reactor`

Spring exercise 1

1. Run the spring application and test endpoints using `requests.http`
2. Open `CustomerController.kt`
3. Modify `getCustomer` endpoint to be a suspended function. Use `awaitSingle()` extension function on the repository to convert `Mono` to suspended function.
   Keep the delay in place.
4. Modify `getCustomers` endpoint to be a suspended function. Find an extension function to convert it to a required return type.
5. Pretend save(customer) function is blocking. Modify the endpoint to be a suspending function. To pretend it, use:
   ```
   customerRepository.save(customer).block()
   ```

Spring exercise 2

1. Open `CustomerOneAndAllController.kt`
2. Modify `getCustomer` and `createCustomer` endpoints to be a suspended function.
   Use `awaitSingle()` extension function on the repository to convert `Mono` to suspended function.
   Keep the delay in place.

Read more

- Coroutines debugging requires special tooling, learn more here: https://kotlinlang.org/docs/debug-coroutines-with-idea.html
- Channels: https://kotlinlang.org/docs/channels.html
- Exception handling: https://medium.com/mindful-engineering/exception-handling-in-kotlin-coroutines-fd08e622360e
- Shared mutable state: https://kotlinlang.org/docs/shared-mutable-state-and-concurrency.html
- Good articles about difference between sequence and flow:
  - https://medium.com/mobile-app-development-publication/kotlin-flow-a-much-better-version-of-sequence-d2555ba9eb94
  - https://medium.com/mobile-app-development-publication/use-sequence-instead-of-kotlin-flow-when-ad577316ce51