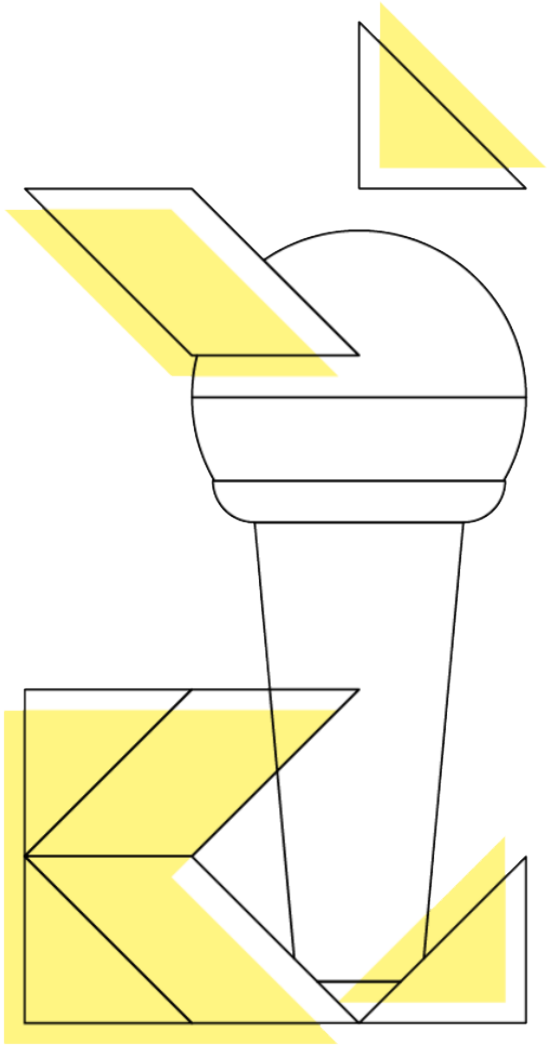
KOTLIN / Everywhere
Turkey KUG

WELCOME

Coroutines

SPEAKER NAME
Mehmet Ali SICAK
CS Teacher, MEB



What is Coroutines?

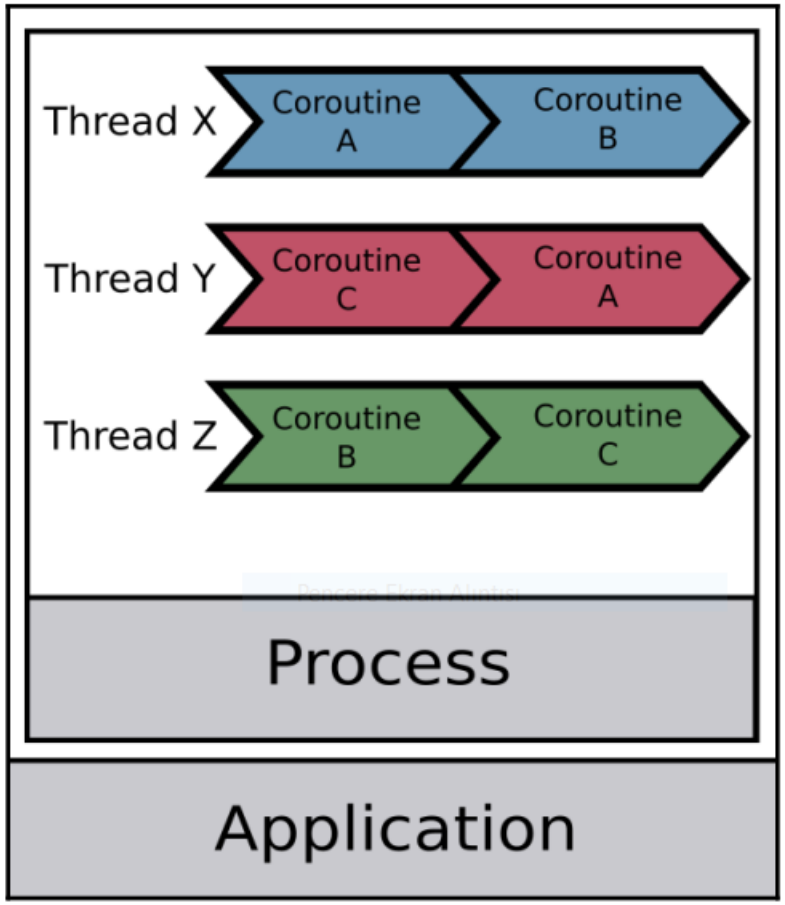
Kotlin 1.1 introduced coroutines

Coroutines are light-weight threads.

Library

kotlinx.coroutines is a rich library
for coroutines developed by JetBrains.

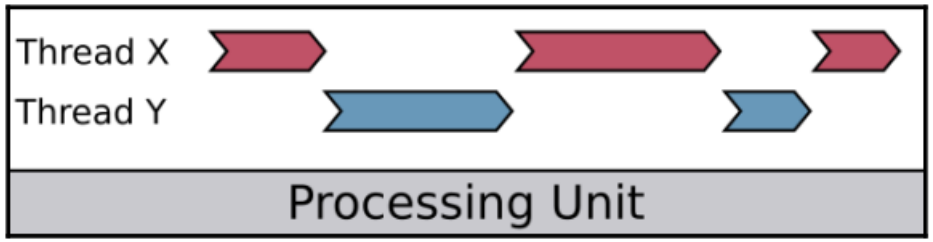
Process, Thread and Coroutine



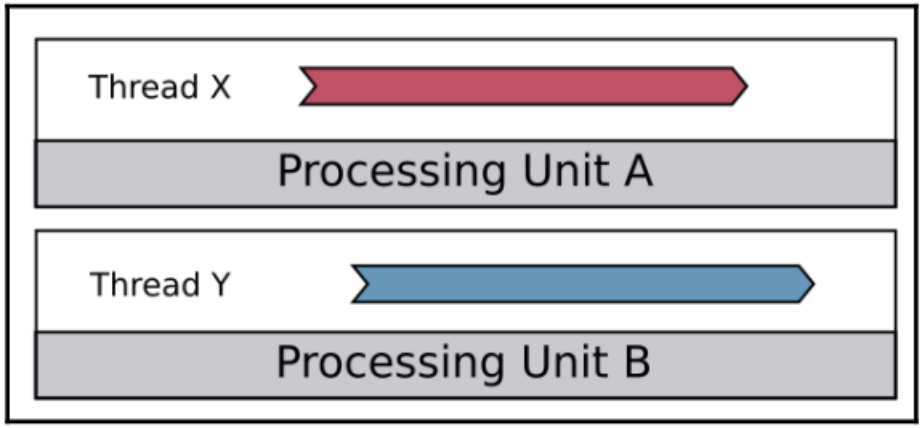
An application is composed of one or more processes and that each process has one or more threads.

Miguel Angel Castiblanco Torres, Learning Concurrency in Kotlin (London, Packt Publishing, 2018), 14.

Concurrency is not parallelism



Concurrency

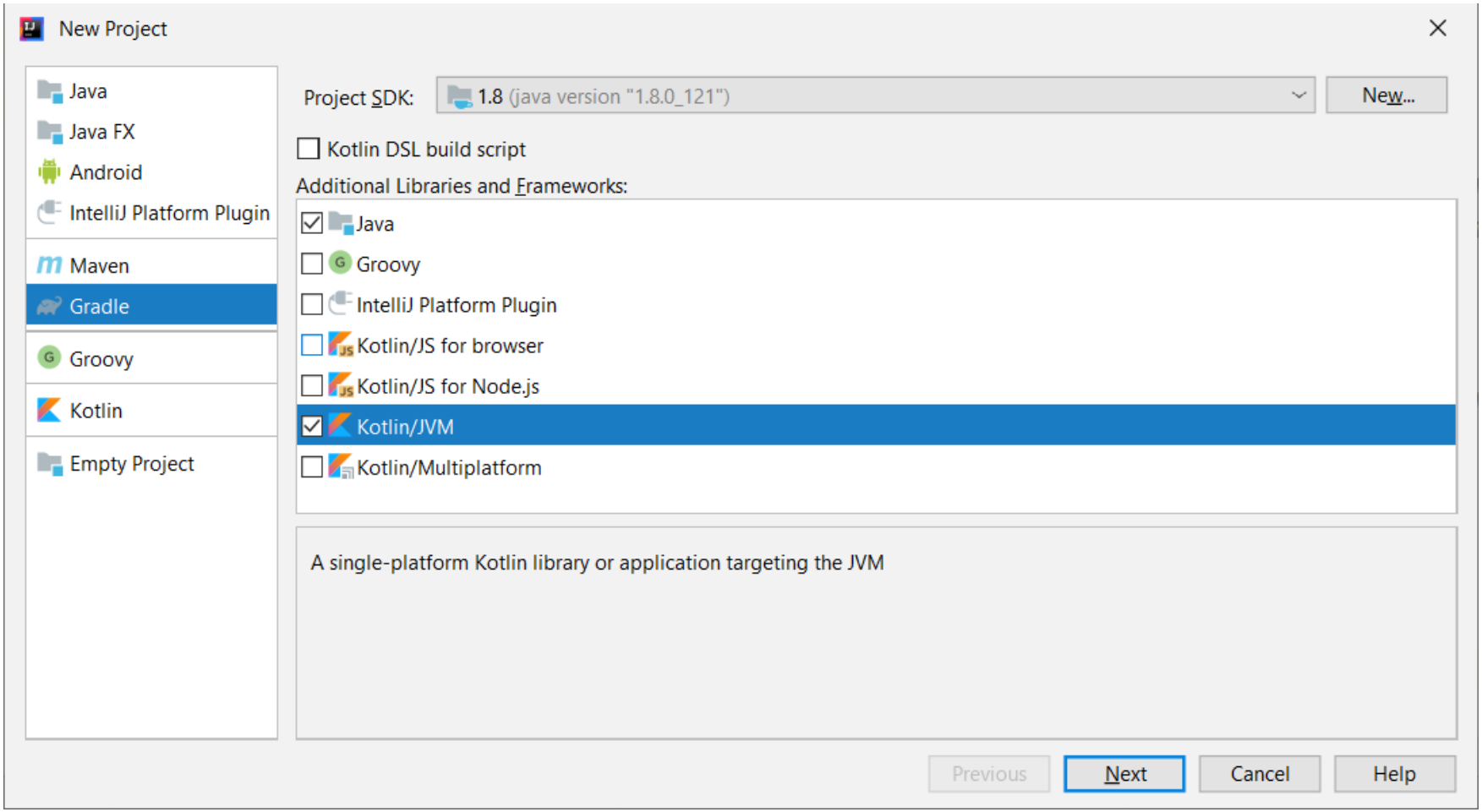


Parallelism

Miguel Angel Castiblanco Torres, Learning Concurrency in Kotlin (London, Packt Publishing, 2018), 19.

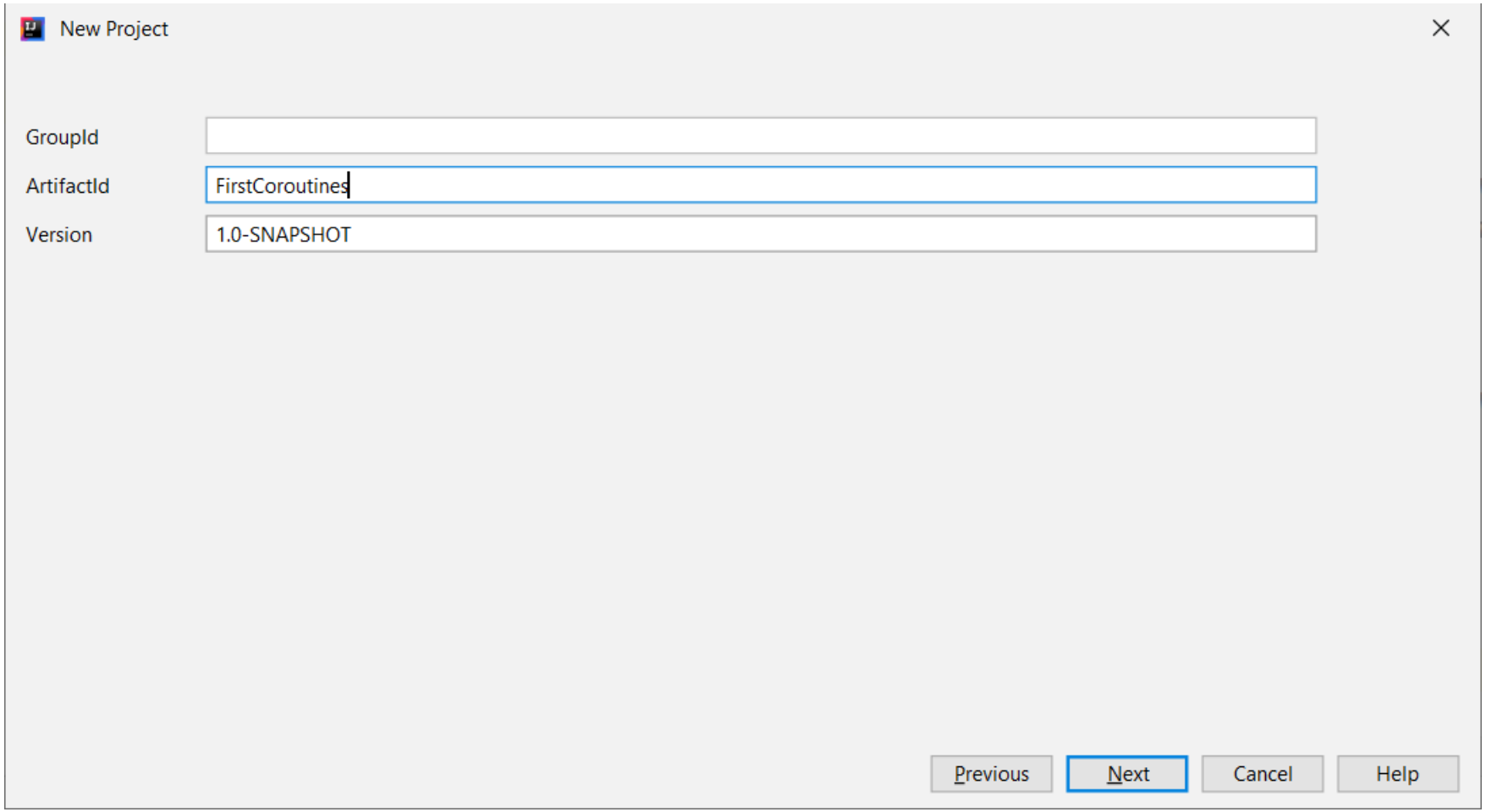
Your first coroutine with Kotlin (Setting up a project)

In IntelliJ IDEA go to File -> New > Project

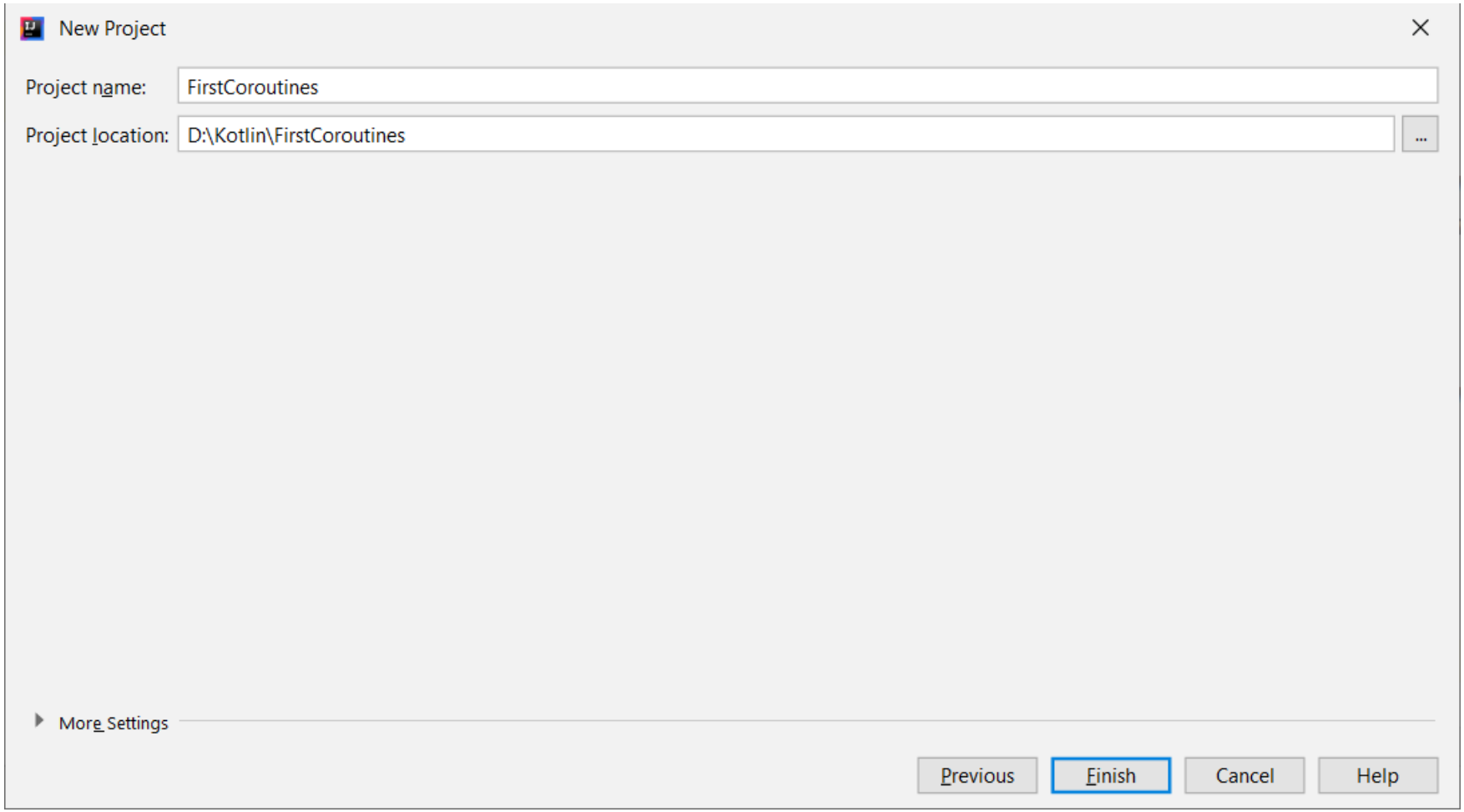


https://kotlinlang.org/docs/tutorials/coroutines/coroutines-basic-jvm.html

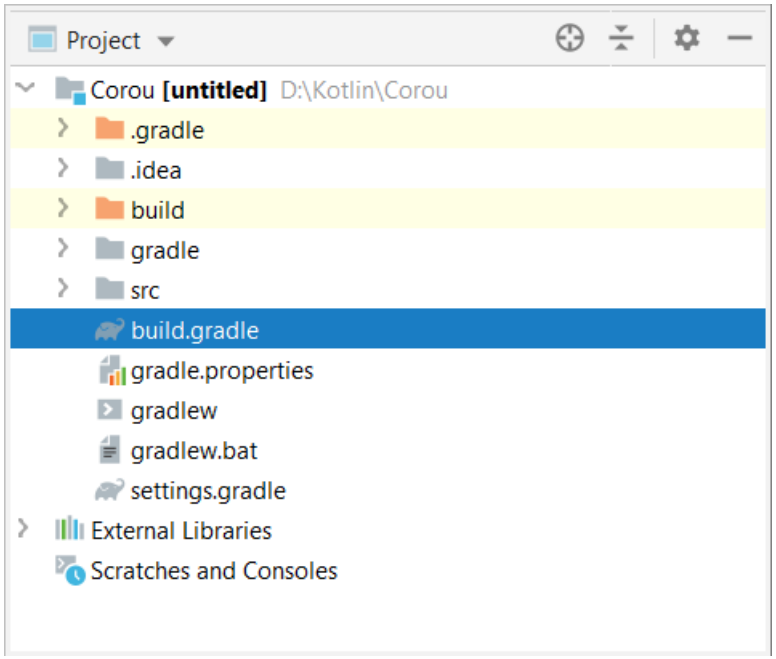
Your first coroutine with Kotlin (Setting up a project)



Your first coroutine with Kotlin (Setting up a project)



Your first coroutine with Kotlin (Setting up a project)



Let's add its recent version to our dependencies

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.1.1"
}
```

My first coroutine

```
import kotlinx.coroutines.*

fun main(args: Array<String>) {
    GlobalScope.launch { // launch a new coroutine in background and continue
        delay( timeMillis: 2000L) // non-blocking delay for 2 second (default time unit is ms)
        println("I am your the first coroutine!") // print after delay
    }
    println("Hello,") // main thread continues while coroutine is delayed
    Thread.sleep( millis: 4000L) // block main thread for 4 seconds to keep JVM alive
}
```

Output

Hello,

Output

Hello,
I am your the first coroutine!

Output

Hello,
I am your the first coroutine!

Process finished with exit code 0

My first coroutine - Output

```
Run: Unnamed
"C:\Program Files\Java\jdk1.8.0_121\b
Hello,
```

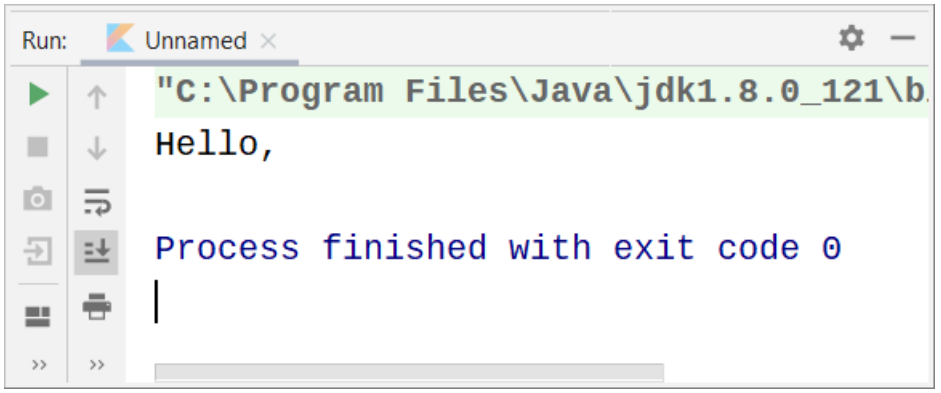
```
Run: Unnamed
"C:\Program Files\Java\jdk1.8.0_121\b
Hello,
I am your the first coroutine!
```

```
Run: Unnamed
"C:\Program Files\Java\jdk1.8.0_121\b
Hello,
I am your the first coroutine!

Process finished with exit code 0
```

What happens to output?

```
import kotlinx.coroutines.*

fun main(args: Array<String>) {
    GlobalScope.launch { // launch a new coroutine in background and continue
        delay( timeMillis: 4001L) // non-blocking delay for 4001 ms (default time unit is ms)
        println("I am your the first coroutine!") // print after delay
    }
    println("Hello,") // main thread continues while coroutine is delayed
    Thread.sleep( millis: 4000L) // block main thread for 4 seconds to keep JVM alive
}
```

Output is

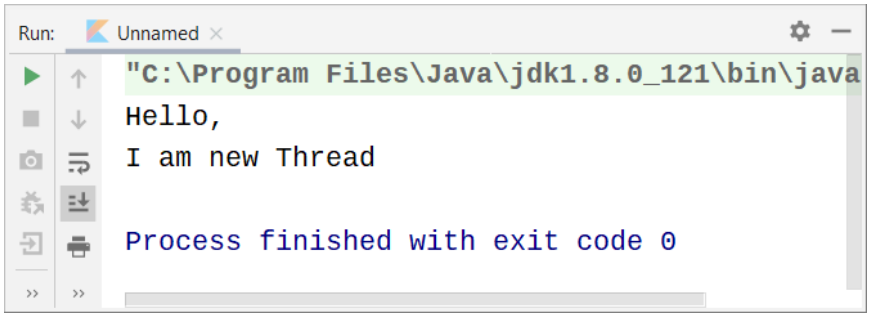


GlobolScope: the lifetime of the new coroutine is limited only by the lifetime of the whole application.

Rearrange the code

You can achieve the same result replacing **GlobalScope.launch { ... }** with **thread { ... }** and **delay(...)** with **Thread.sleep(...)**

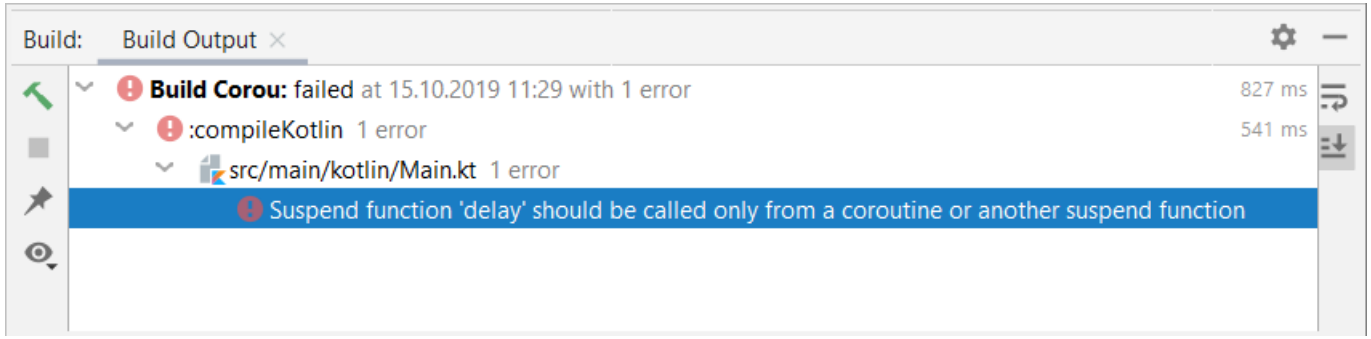
```
import kotlinx.coroutines.*
import kotlin.concurrent.thread

fun main(args: Array<String>) {
    thread {
        Thread.sleep( millis: 2000)
        println("I am new Thread")
    }

    println("Hello,")
    Thread.sleep( millis: 4000L)
}
```



Rearrange the code

```
import kotlinx.coroutines.*
import kotlin.concurrent.thread

fun main(args: Array<String>) {
    thread {
        delay( timeMillis: 2000)
        println("I am new Thread")
    }

    println("Hello,")
    Thread.sleep( millis: 4000L)
}
```

If you start by replacing **GlobalScope.launch** by **thread**, the compiler produces the following error



That is because ***delay*** is a special ***suspending function*** that does not block a thread, but suspends coroutine and it can be only used ***from a coroutine.***

Bridging blocking and non-blocking worlds

```
fun main(args: Array<String>) {

    GlobalScope.launch { // launch a new coroutine in background and continue
        delay( timeMillis: 1000L)
        println("World!")
    }
    println("Hello,") // main thread continues here immediately
    runBlocking {     // but this expression blocks the main thread
        delay( timeMillis: 2000L)  // ... while we delay for 2 seconds to keep JVM alive
    }
}
```

Output
Hello,

Hello,
World!

Output

Hello,
World!

Output

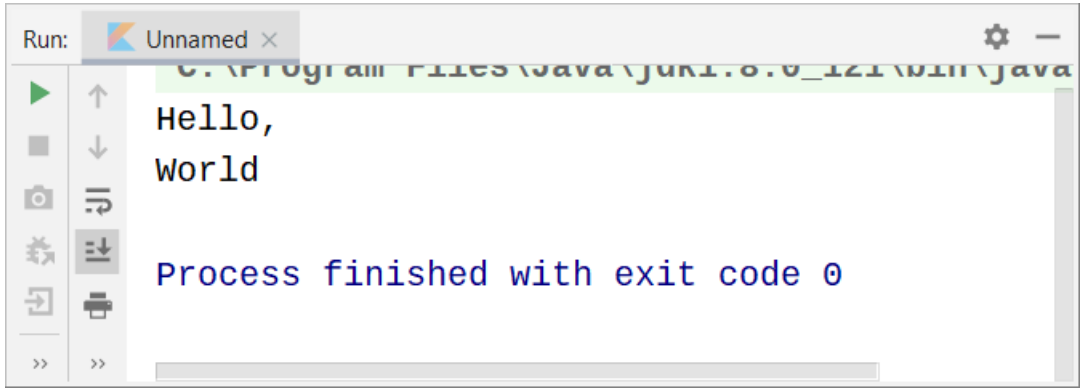
Process finished with exit code 0

The main thread invoking runBlocking blocks until the coroutine inside runBlocking completes.

Bridging blocking and non-blocking worlds

This example can be also rewritten in a more idiomatic way

```
import kotlinx.coroutines.*

fun main(args: Array<String>)  = runBlocking<Unit> { // start main coroutine
    GlobalScope.launch { // launch a new coroutine in background and continue
        delay( timeMillis: 1000L)
        println("World")
    }
    println("Hello,") // main coroutine continues here immediately
    delay( timeMillis: 2000L)      // delaying for 2 seconds to keep JVM alive
}
```



Extract function refactoring

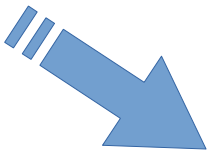
```
fun main(args: Array<String>) = runBlocking { this: CoroutineScope
    launch { this: CoroutineScope
        for(k in 1..3){
            doWorld()
        }
    }
    println("Hello,")
    println("It's done!")
}

// this is your first suspending function
suspend fun doWorld() {
    delay( timeMillis: 1000L)
    println("World!")
}
```

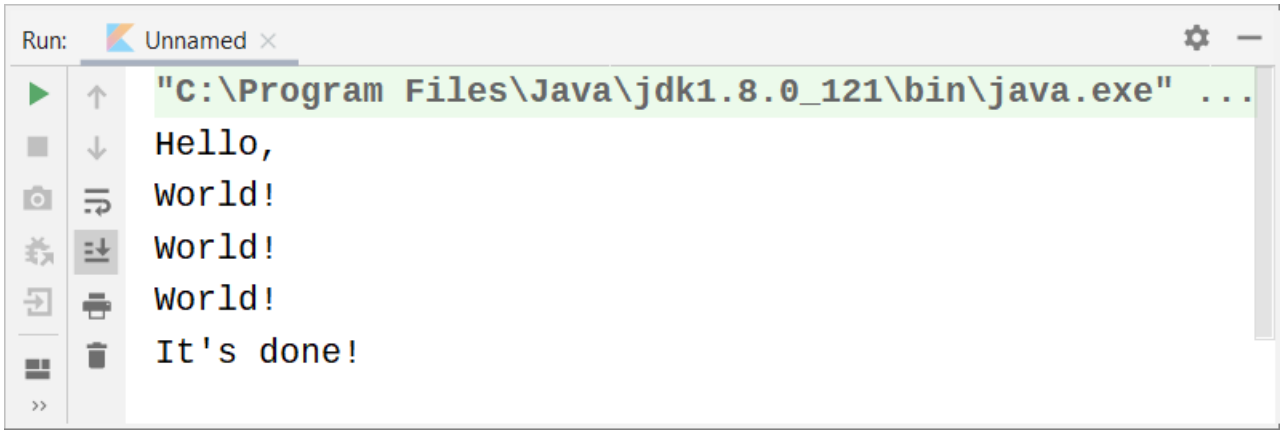
That is your first suspending function.

```
Run:  Unnamed
"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...
Hello,
It's done!
World!
World!
World!
```

Extract function refactoring

```
fun main(args: Array<String>) = runBlocking { this: CoroutineScope
    launch { this: CoroutineScope
        for(k in 1..3){
            doWorld()
        }
    }
    println("Hello,")
    delay( timeMillis: 4000)
    println("It's done!")
}

// this is your first suspending function
suspend fun doWorld() {
    delay( timeMillis: 1000L)
    println("World!")
}
```

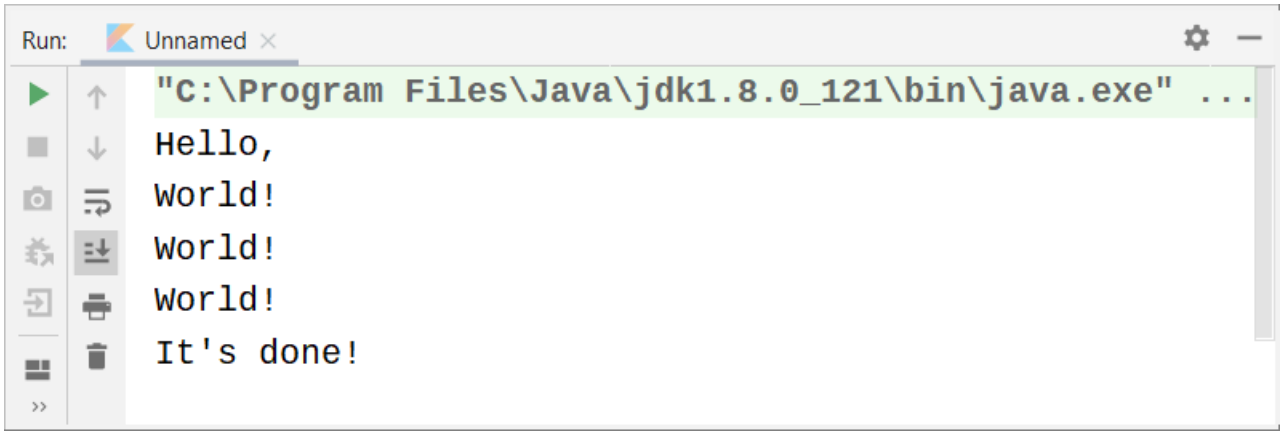




Waiting for a job

```
fun main(args: Array<String>) = runBlocking { this: CoroutineScope
    val job =launch { this: CoroutineScope
        for(k in 1..3){
            doWorld()
        }
    }
    println("Hello,")
    job.join()
    println("It's done!")
}

// this is your first suspending function
suspend fun doWorld() {
    delay( timeMillis: 1000L)
    println("World!")
}
```

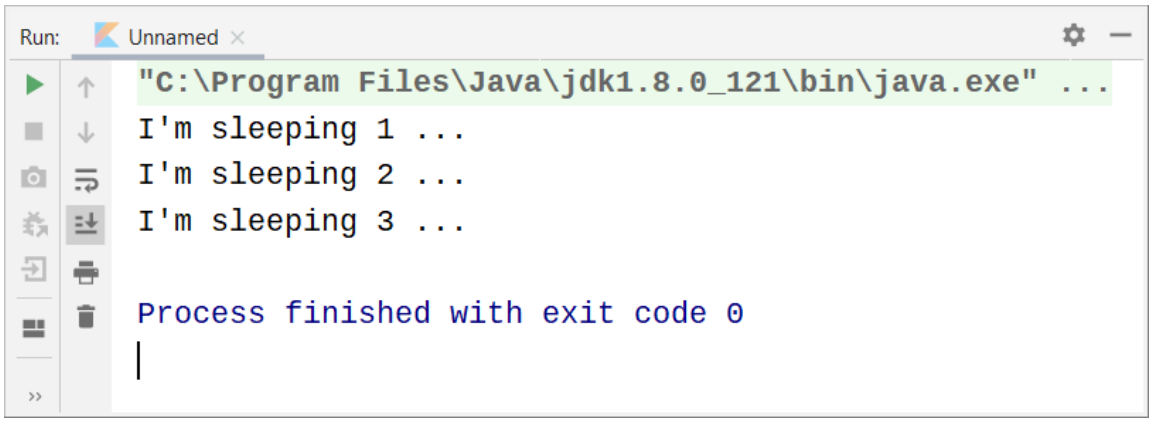
Delaying for a time while another coroutine is working is not a good approach.



Global Coroutines

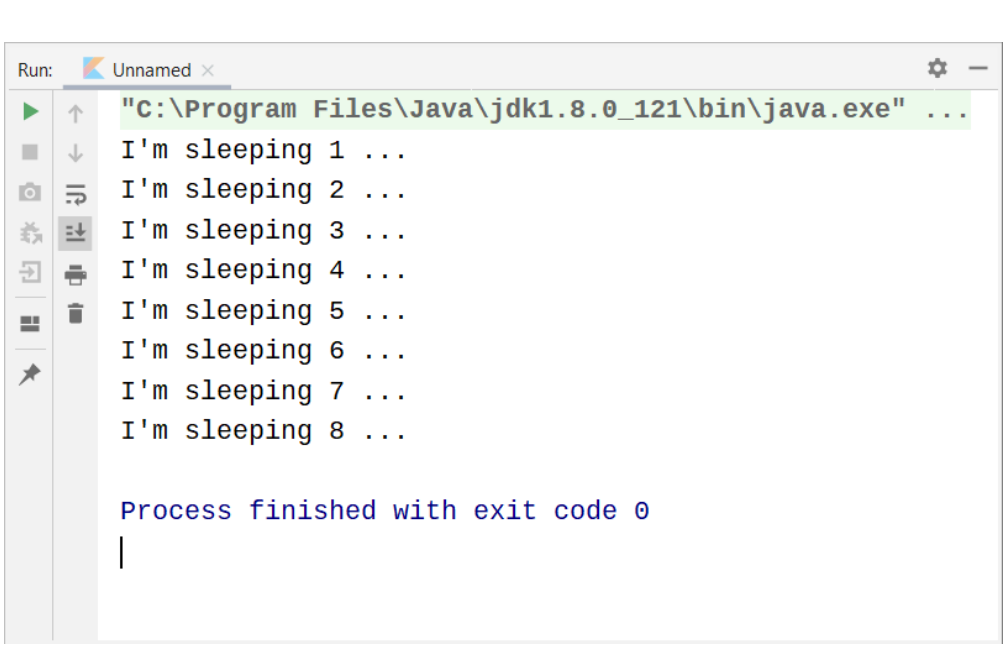
```
fun main(args: Array<String>) = runBlocking { this: CoroutineScope
    GlobalScope.launch { this: CoroutineScope
        for(i  in 1..8) {
            println("I'm sleeping $i ...")
            delay( timeMillis: 500L)
        }
    }
    delay( timeMillis: 1200L) // just quit after delay
}
```

Global coroutines are like daemon threads



Global Coroutines

```
fun main(args: Array<String>) = runBlocking { this: CoroutineScope
    launch { this: CoroutineScope
        for(i  in 1..8) {
            println("I'm sleeping $i ...")
            delay( timeMillis: 500L)
        }
    }
    delay( timeMillis: 1200L) // just quit after delay
}
```



Useful Links

- https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html

HAPPY HOUR

THANK YOU