

**KOTLIN /
Everywhere**
Turkey KUG

WELCOME

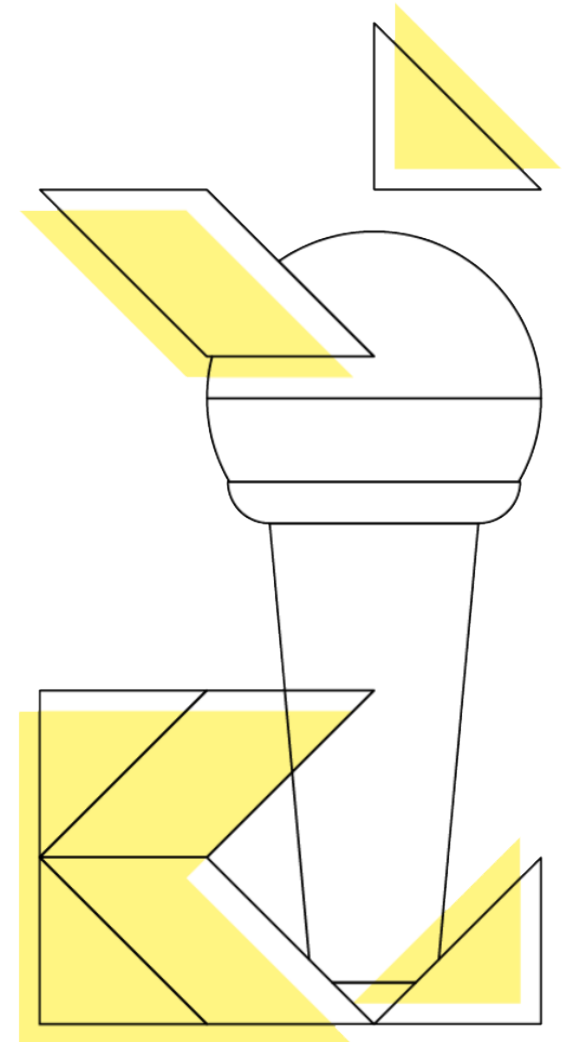


Coroutines

SPEAKER NAME

Mehmet Ali SICAK

CS Teacher, MEB



What is Coroutines?

**Don't block
Keep moving**

A Tasks — Threads

Kotlin 1.1 introduced coroutines

Coroutines are light-weight threads.

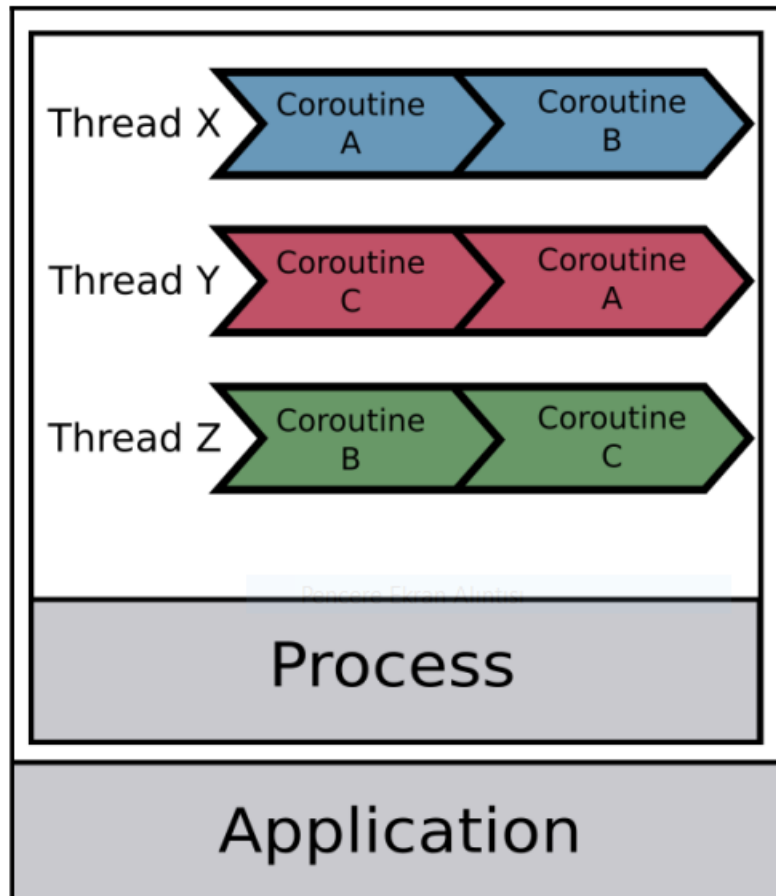


Library

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

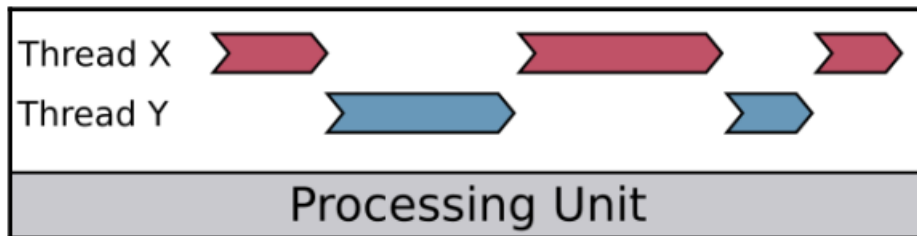


Process, Thread and Coroutine

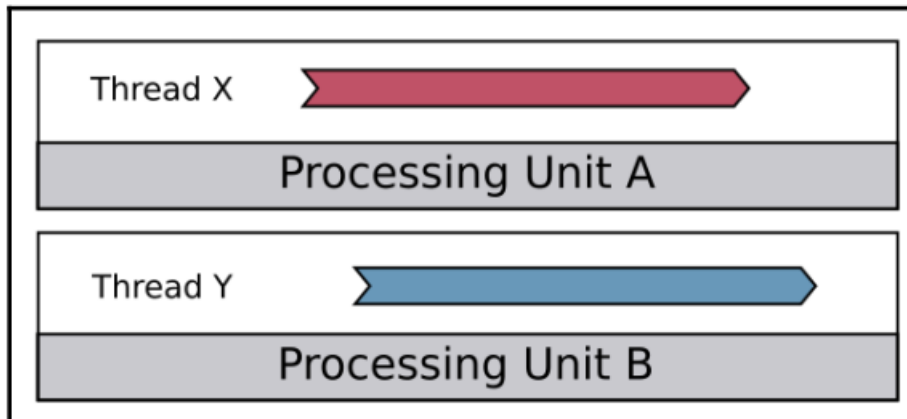


An application is composed of one or more processes and that each process has one or more threads.

Concurrency is not parallelism



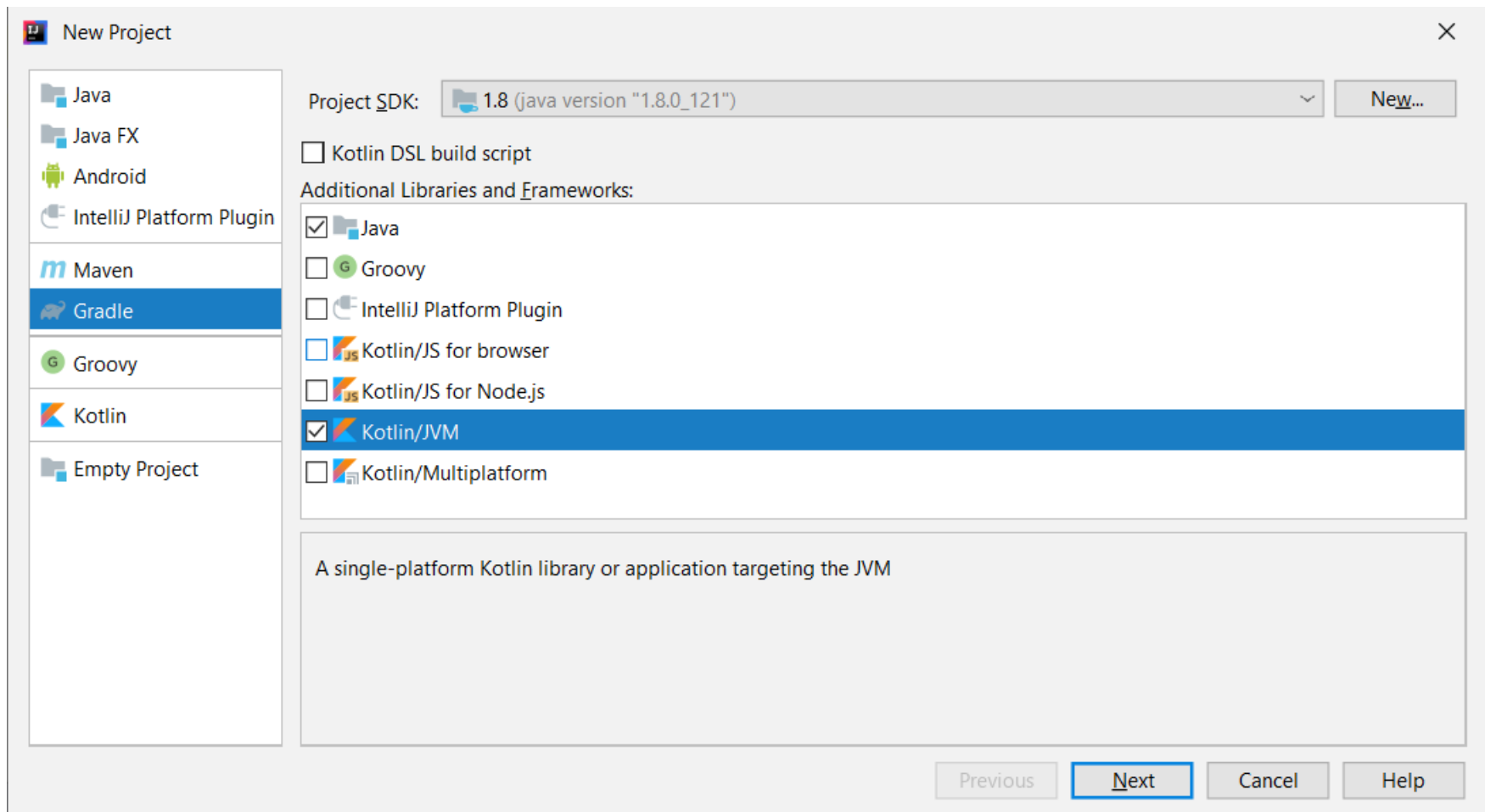
Concurrency



Parallelism

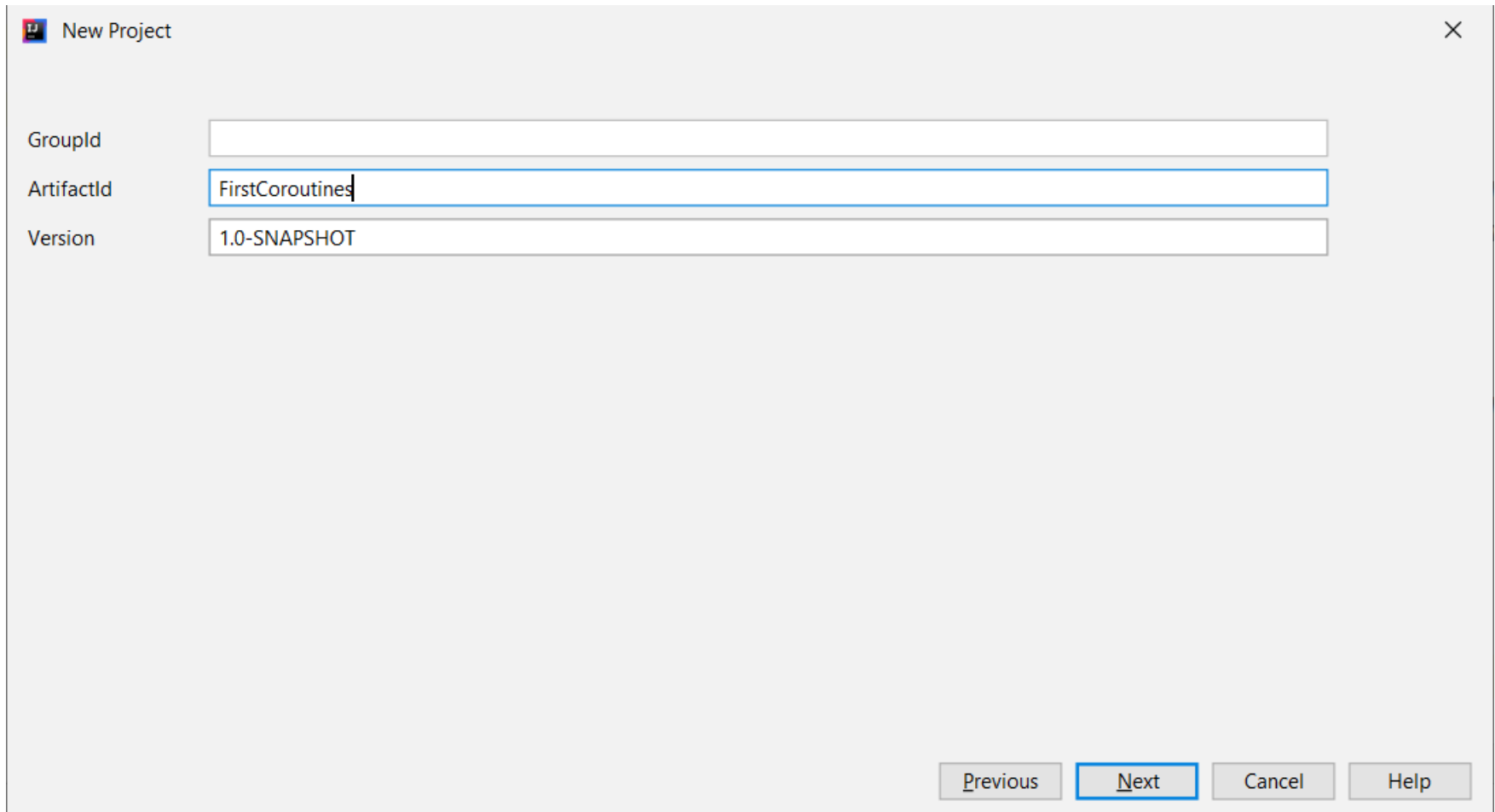
Your first coroutine with Kotlin (Setting up a project)

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)



The image shows the 'New Project' dialog box in IntelliJ IDEA. The dialog has a title bar with the IntelliJ logo and the text 'New Project'. It contains three input fields: 'GroupId' (empty), 'ArtifactId' (containing 'FirstCoroutines'), and 'Version' (containing '1.0-SNAPSHOT'). At the bottom, there are four buttons: 'Previous', 'Next' (highlighted with a blue border), 'Cancel', and 'Help'.

New Project

GroupId

ArtifactId

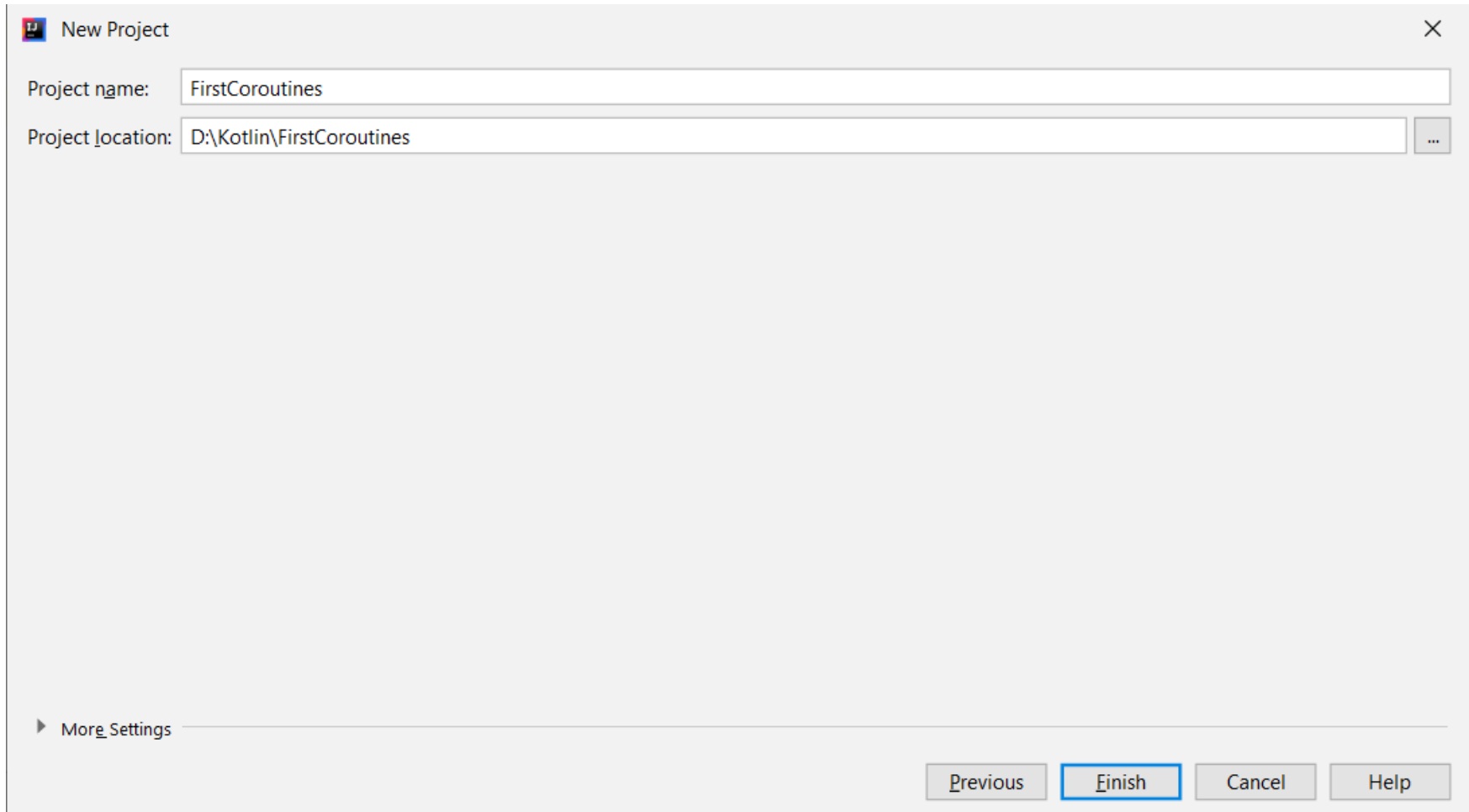
Version

FirstCoroutines

1.0-SNAPSHOT

Previous Next Cancel Help

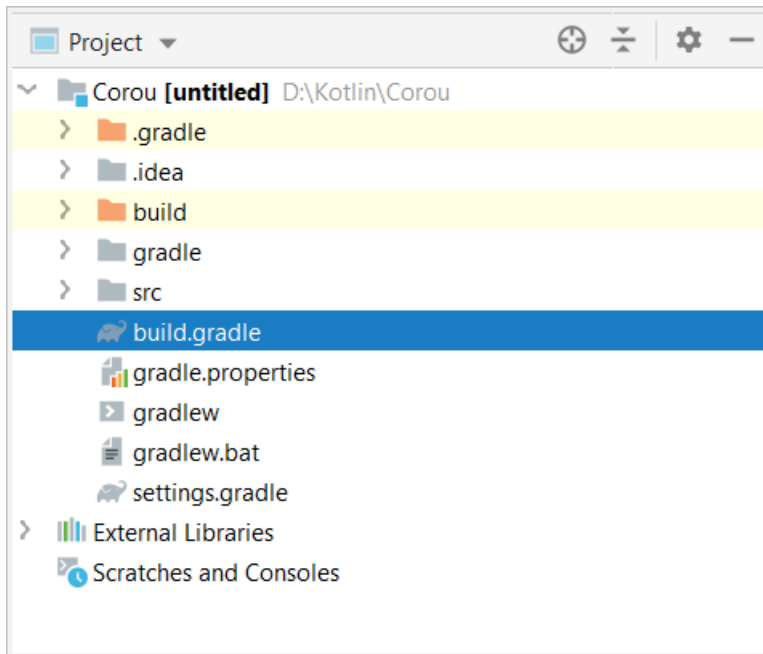
Your first coroutine with Kotlin (Setting up a project)



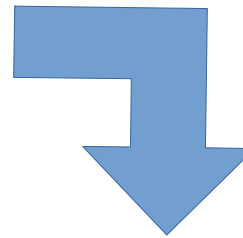
The screenshot shows a 'New Project' dialog box with the following fields and buttons:

- Project name:** FirstCoroutines
- Project location:** D:\Kotlin\FirstCoroutines
- Buttons:** Previous, Finish (highlighted with a blue border), Cancel, Help
- More Settings:** A link to expand more settings.

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!

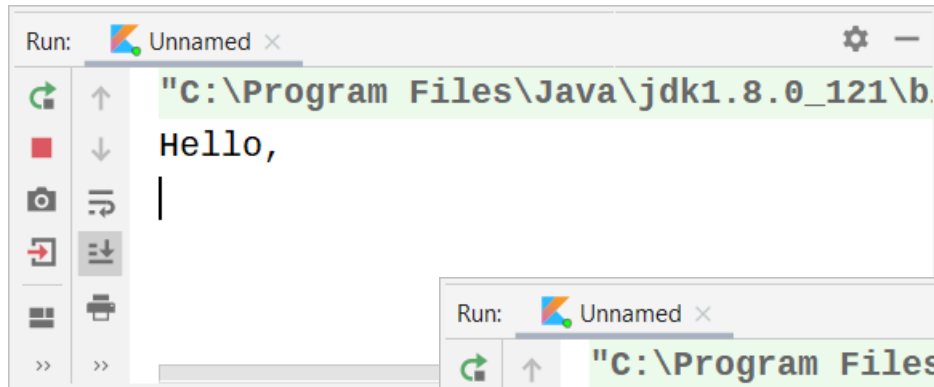


Output

Hello,
I am your the first coroutine!

Process finished with exit code 0

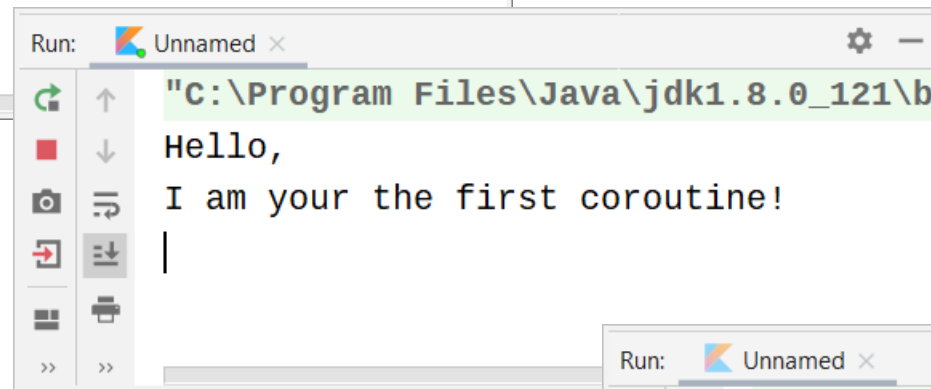
My first coroutine - Output



Run: Unnamed x

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

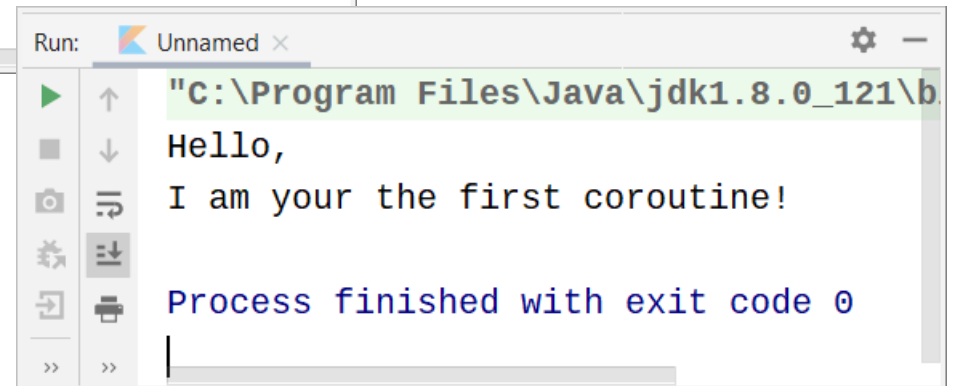
This is the first console window. It shows the Java path and the word "Hello," followed by a cursor. A blue arrow points from this window to the second one.



Run: Unnamed x

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

This is the second console window. It shows the same Java path and "Hello," but now includes the line "I am your the first coroutine!" followed by a cursor. A blue arrow points from this window to the third one.



Run: Unnamed x

```
"C:\Program Files\Java\jdk1.8.0_121\b  
Hello,  
I am your the first coroutine!  
  
Process finished with exit code 0  
|
```

This is the third console window. It shows the same Java path and "Hello," and "I am your the first coroutine!". It also includes the line "Process finished with exit code 0" in blue text, indicating the program has completed successfully.

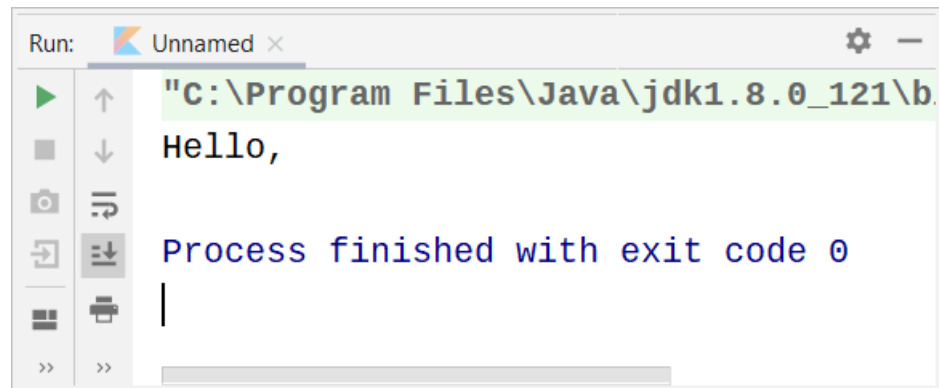
What happens to output?

```
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



GlobalScope: 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)
}
```



```
Run: Unnamed x
"C:\Program Files\Java\jdk1.8.0_121\bin\java"
Hello,
I am new Thread
Process finished with exit code 0
```

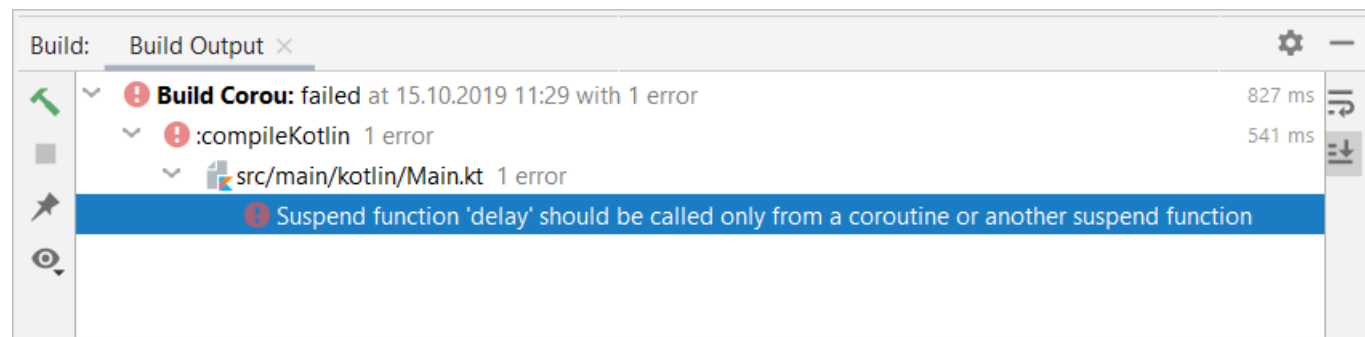
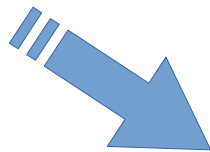
Rearrange the code

```
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

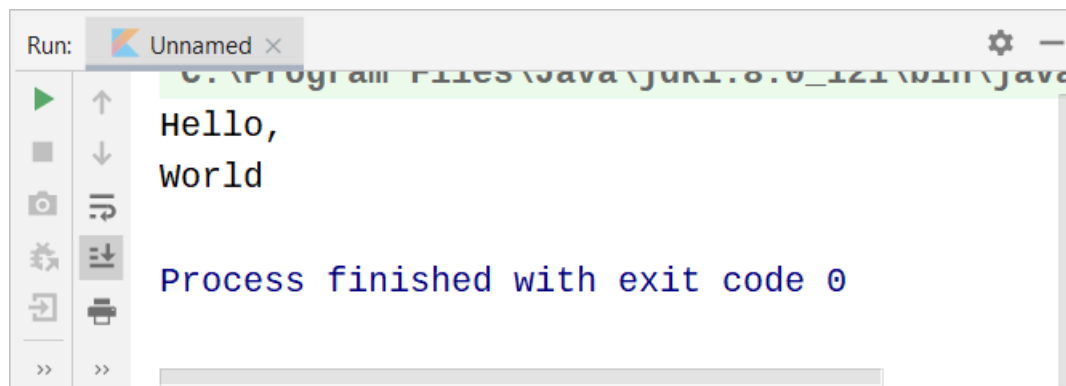
The main thread invoking `runBlocking` blocks until the coroutine inside `runBlocking` completes.

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
}
```



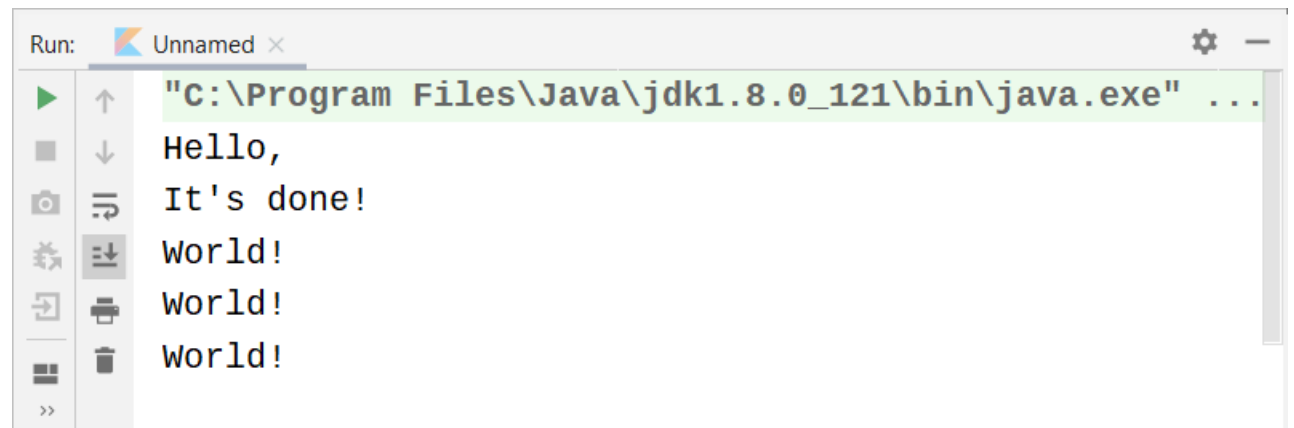
Extract function refactoring

```
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.



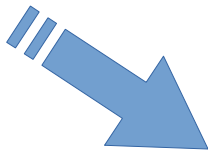
```
Run: C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...  
Hello,  
It's done!  
World!  
World!  
World!
```

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!")
}
```



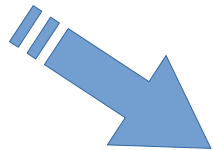
```
Run: Unnamed x
"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...
Hello,
World!
World!
World!
World!
It's done!
```

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!")
}
```

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

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

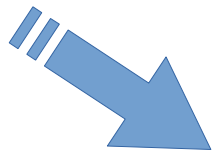


```
Run: Unnamed x
"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...
Hello,
World!
World!
World!
World!
It's done!
```

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
}
```

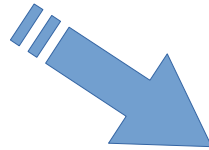
Global coroutines
are like daemon
threads



```
Run: Unnamed x
"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
Process finished with exit code 0
```

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
}
```



```
Run: Unnamed x
"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
I'm sleeping 4 ...
I'm sleeping 5 ...
I'm sleeping 6 ...
I'm sleeping 7 ...
I'm sleeping 8 ...

Process finished with exit code 0
|
```

Useful Links

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

HAPPY HOUR

THANK YOU

