Concurrency

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction 2
- Motivation 3
- Concurrency 6
- Processes & Threads 7

Coroutines 11
- What is a coroutine? 12
- Using Coroutines 15
- Coroutine Builders 18
- Dispatchers 26
- Structured Concurrency 29
- Coroutine Scope 30
- Debugging 33

Bibliography 34

Introduction

Motivation

We are accustomed to writing programs that execute linearly. We sometimes refer to this as synchronous execution, where we expect the CPU to wait for one instruction to complete before proceeding to the next. In this example, each function call is `blocking` or preventing further execution until it completes and returns a value.

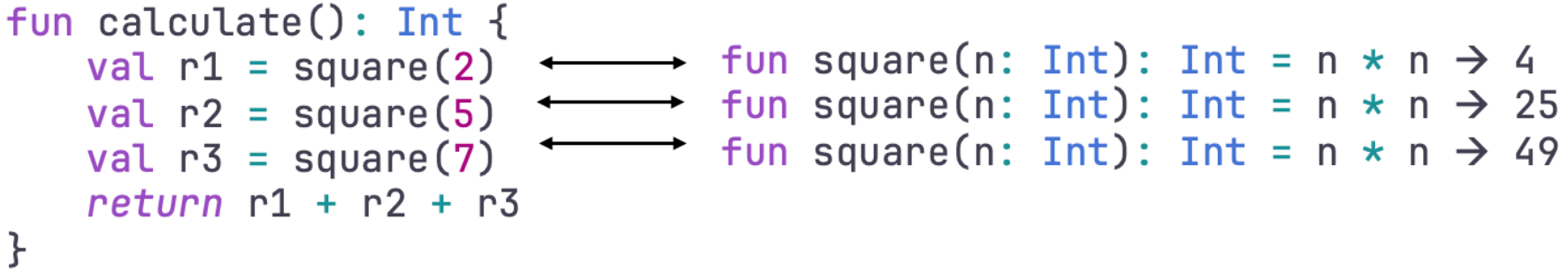


Figure 1: Each call to the square() function needs to return its result before the next invocation. This is synchronous execution, since we rely on executing instructions strictly in-order.

Motivation

Synchronous executions works fine for many tasks. However, as soon as you start working anything that takes time to process e.g., a database or network, this model falls over. In those situations, we would prefer asynchronous execution, where we can juggle multiple tasks [1].

In this example, the UI won't load until the transactions are loaded, even though we don't need the transaction data for the UI! It would be amazing if we could have the UI display while the transactions continue to load separately i.e. if one task didn't block the other [2].

```
@Composable
fun loadCustomer(id: custID, txTable: Table, custTable: Table) {
  val customer = custTable.fetch(id) // very fast
  val transactions = transactionsTable.fetch(customer) // very slow
  Column {
    CustomerView(customer)
  }
}
```

Motivation

You can already see this behaviour in standard GUI applications, where an application might “simultaneously”:

- Process a mouse-click to resize the application window,
- Draw some output to the screen,
- Read data from a database, and
- Fetch data from the web.

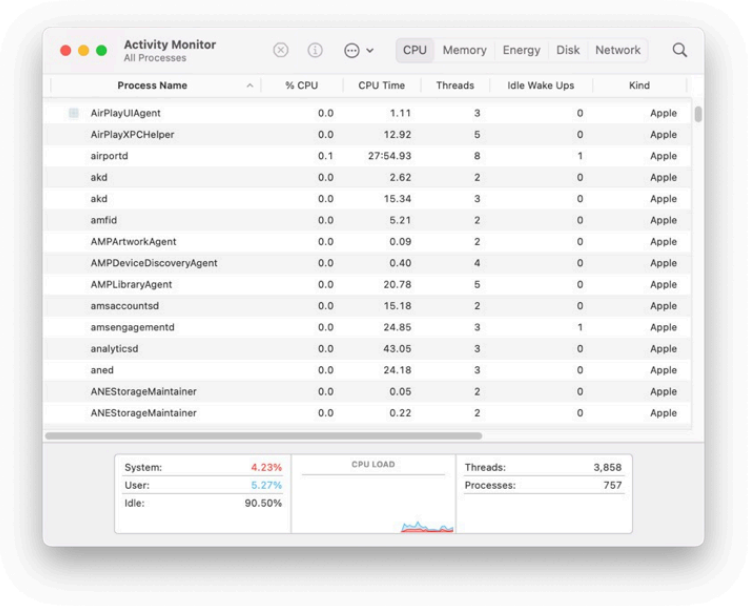
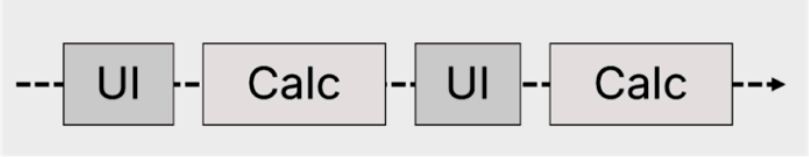


Figure 2: My computer, as I’m creating this slide, with 757 running processes. I don’t know what most of these are…

Concurrency

Concurrency is a general term that is used to mean that multiple tasks are **being worked on** simultaneously.

This does not suggest that tasks are being executed at the same time, only that we can alternate between them. Concurrency is about structuring processing in a way that lets us manage it efficiently [3].

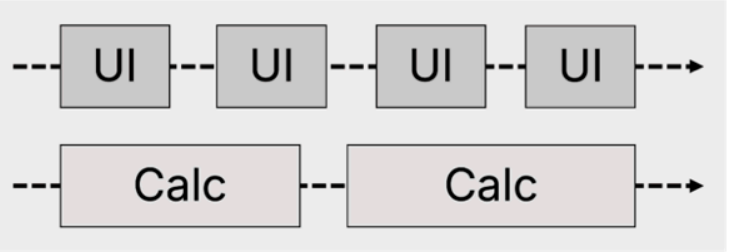


By switching back and forth between multiple tasks as required (like redrawing the user interface and performing parts of a long-running calculation), an application leverages concurrency.

Figure 3: Concurrent execution.

Parallelism means performing or **executing multiple tasks simultaneously**..

Parallel computations can use multi-core hardware effectively i.e. spread tasks across cores.



In this example, long-running calculations happen in the background while the UI is being rendered. This is parallelism, since operations happen simultaneously.

Figure 4: Parallel execution.

Processes & Threads

A process is a running instance of a program that is managed by the operating system.

- A thread is a context within which the CPU can execute instructions.
- Each program has one `main` thread that is created when your program launches.

Instructions typically run on the `main` thread.

- If your program requires additional threads, you (the developer) need to write code to create and manage them.
- Developer created threads are referred to as worker-threads (or sometimes background threads since they are doing background processing).

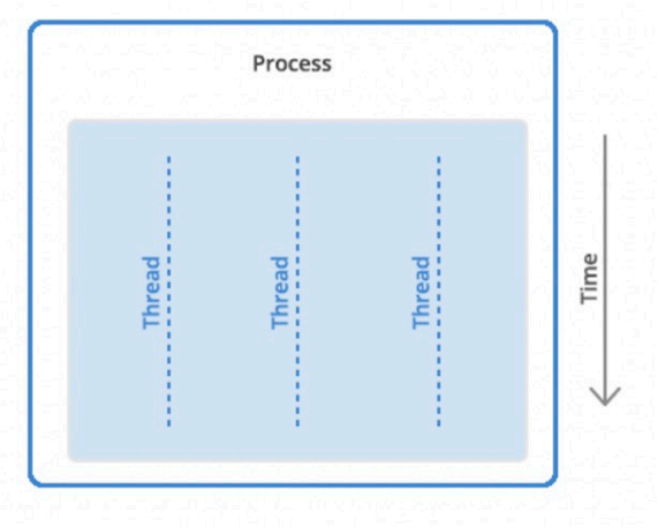


Figure 5: All instructions in a program are processed by one or more threads. Most programs only have one thread. Credit: backblaze.com

Processes & Threads

Creating additional threads provides significant benefits but adds complexity.

We can split computation across threads (one primary thread, and one or more background threads).

- e.g. one thread can wait for the blocking operation to complete, while the other threads continue processing.

This has the potential to increase performance, if we can split up work in a way that's efficient and lets us exploit this structure.

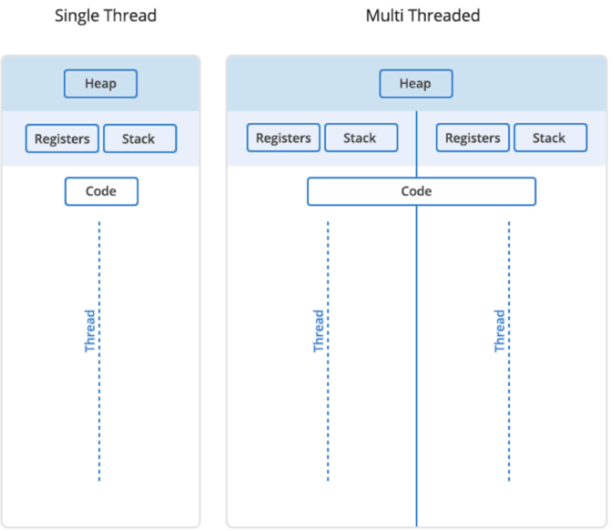


Figure 6: Each thread has its own registers and stack, but threads share heap memory. Credit: backblaze.com

Processes & Threads

Managing Threads

Kotlin has a function for managing threads. This is a fairly “standard” way to split out tasks and have them run asynchronously [4].

```
// interface
fun thread(
  start: Boolean = true,
  isDaemon: Boolean = false,
  contextClassLoader: ClassLoader? = null,
  name: String? = null,
  priority: Int = -1,
  block: () → Unit
): Thread

// creating a thread and launching it
// the body is the code to execute in that thread
thread(start = true) {
  println("${Thread.currentThread()} has run.")
}
```

Processes & Threads

Can we just use threads for everything?

It's not recommended. There are disadvantages to directly managing threads.

Threads are a poor abstraction for anything complex.

- Threads are "expensive" to start/stop and consume lots of memory.
- Launching or context switching between threads takes significant time.
- Worker threads may not always available in the number we require.
- A blocking operation on a thread blocks that thread.
  - e.g., while you want for a DB call to return, the thread is "hung".
- We have limited ability to control when/how threads run, so we risk race conditions.

Threads also change the program flow since they force you to think in terms of "blocks of code" for each thread.

- Threads make program execution difficult to follow.
- You can't just read code sequentially to understand the program's logic.

Coroutines

What is a coroutine?

Kotlin's approach to working with asynchronous code is to use coroutines.

A coroutine is a suspendable computation: a section of code that can suspend execution at some point and then resume processing later.

Why is this useful?

- Suspending code allows the OS to execute something else while waiting. This facilitates cooperative multitasking.
- This abstraction allows us to describe how our code should be executed, without worrying about the allocation of work to threads.
- Coroutines directly support concurrency.

What is a coroutine?

Coroutines provide a greatly improved abstraction:

- They are very lightweight and require far fewer resources than a thread.
- Coroutines can suspend without blocking resources i.e. if coroutine is suspended on a thread, that thread can still be used for something else.
- A coroutine is executed by a thread, but it is not tied to that specific thread. It may suspend its execution in one thread and resume in a different one.

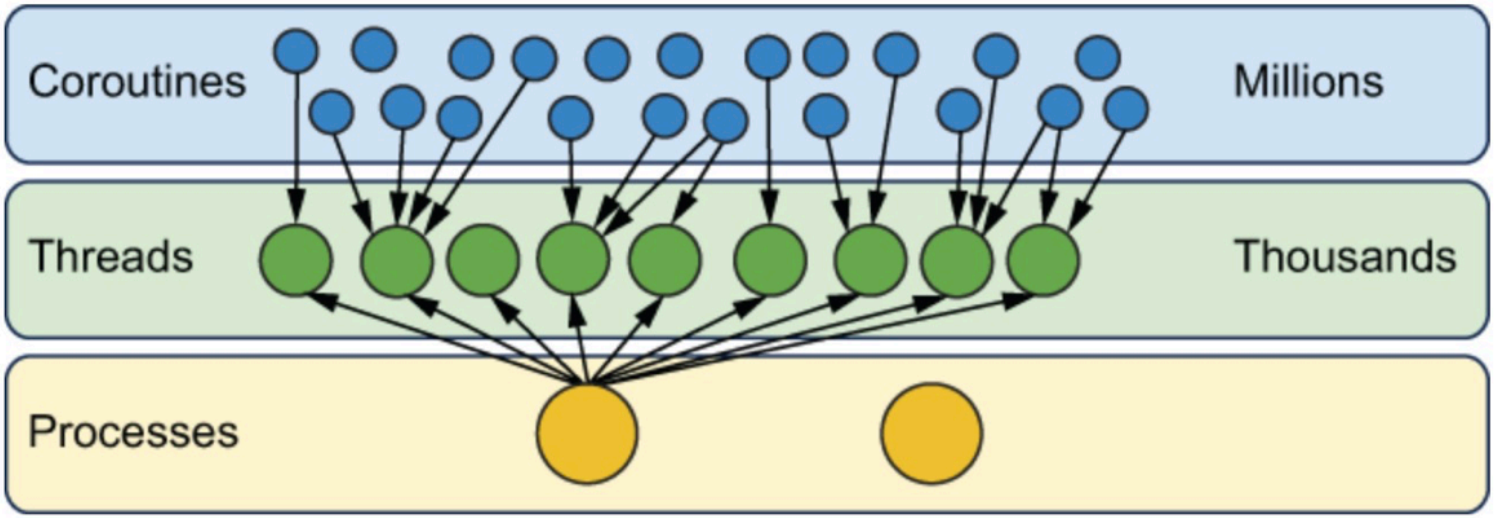


Figure 7: Aigner et al. Kotlin in Action (2024).

What is a coroutine?

Kotlin provides the `kotlinx.coroutines` library. You will need to add the dependency to your module's `module.yaml` file and then import the library in source code.

module.yaml

```
dependencies:
  - org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.1
```

source-code.kt

```
import kotlinx.coroutines.*
```

Using Coroutines

To run something asynchronously, you need two things to be present:

1. A coroutine which acts as a “runner” for asynchronous code.
   - We’ll use coroutine builder functions to create coroutines.
   - There are different coroutine builders that produce coroutines with different behaviours.

2. A block of code to run asynchronously.
   - Typically a coroutine or some other “special function”.

> **Tip**
>
> Basically, you setup a coroutine with some code to run, and it will execute the code based on how that specific coroutine works.

Using Coroutines

Suspending Functions

Suspending functions are regular functions that can be suspended by a coroutine.

A suspending function looks like a regular function with the “suspend” keyword.

Why are they useful?

- They serve as ”suspension points” for the coroutine, where it can suspend execution.
- A function that is suspended frees up the thread for other uses.

See Kotlin Play demo

```
fun main() {
  runBlocking {
    // suspending functions
    doOne()
    doTwo()
  }
}

suspend fun doOne() {
  delay(1000.milliseconds)
  println("doOne is done")
}

suspend fun doTwo() {
  delay(1000.milliseconds)
  println("doTwo is done")
}
```

Using Coroutines

```
suspend fun login(credentials: Credentials): UserID
suspend fun loadUserData(userID: UserID): UserData
fun showData(data: UserData)

suspend fun showUserInfo(credentials: Credentials) {
  val userID = login(credentials) // non-blocking
  val userData = loadUserData(userID) // non-blocking
  showData(userData)
}
```

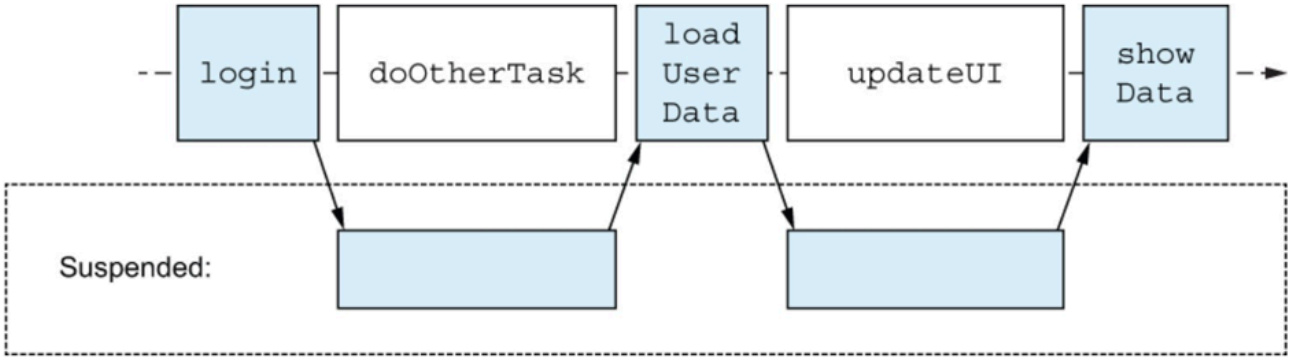


Figure 8: Each function “yields” to the OS, which can use the thread for other activities.

Coroutine Builders

A coroutine is generated by a coroutine-builder which creates and executes that coroutine.

runBlocking

- bridges normal and asynchronous code.
- blocks its thread until its job is complete.

launch

- executes code asynchronously.
- can only be run from an existing coroutine.
- returns a “job” that can be used to track progress.

async

- executes code asynchronously.
- can only be run from an existing coroutine.
- returns a “deferred” object, that you can wait on for a return value.

Coroutine Builders

runBlocking

runBlocking is a special coroutine builder that bridges the world of regular functions to the asynchronous code that will run.

- This program only proceeds past the runBlocking call when all the code in the lambda has returned.
- runBlocking is necessary because you cannot mix synchronous and asynchronous code!

```
fun main() {
  runBlocking {
    // asynchronous functions
    doSomething()
    doSomethingElse()
  }
  // synchronous code
  regularCode()
}
```

Coroutine Builders

launch

```
fun main() {
  log("The program launches")
  GlobalScope.launch {
    log("The first coroutine starts and is ready to be suspended")
    delay(500.milliseconds)
    log("The first coroutine is resumed")
  }

  GlobalScope.launch {
    log("The second coroutine starts and is ready to be suspended")
    delay(500.milliseconds)
    log("The second coroutine is resumed")
  }
  log("The program completes")
}

// 0 [main] The program launches
// 43 [main] The program completes
// Where's the rest of the output??
```

Coroutine Builders

```
fun main() {
  log("The program launches")
    runBlocking {
      log("runBlocking launches")
      launch { /* ... */ }
  // ...
}
// 0 [main] The program launches
// 48 [main @coroutine#1] runBlocking launches
// 50 [main @coroutine#1] runBlocking pauses
// 51 [main @coroutine#2] The first coroutine starts and is ready to
be suspended
// 58 [main @coroutine#3] The second coroutine starts and is ready
to be suspended
// 562 [main @coroutine#2] The first coroutine is resumed
// 562 [main @coroutine#3] The second coroutine is resumed
// 562 [main] The program completes
```

`runBlocking` will pause and wait for its children to complete

See Kotlin Play demo

Coroutine Builders

`launch` also provides some degree of job control.

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.

- e.g., wait for completion of the child coroutine and then print “Done”.

```
val job = launch { // keep a reference to the job
  delay(1000L)
  println("World!")
}

print("Hello")
job.join() // wait until child coroutine completes
println("Done")

// Hello World!
```

See Kotlin Play demo

Coroutine Builders

We can also cancel jobs. This is used frequently in applications to cancel unecessary operations.

```
val job = launch {
  repeat(1000) { println("job: I'm sleeping $it ..."); delay(500L) }
}

delay(1300L)  // delay a bit
println("main: I'm tired of waiting!")
job.cancel()  // cancels the job
job.join()    // waits for job's completion
println("main: Now I can quit.")

// job: I'm sleeping 0 ...
// job: I'm sleeping 1 ...
// job: I'm sleeping 2 ...
// main: I'm tired of waiting!
// main: Now I can quit.
```

See Kotlin Play demo

Coroutine Builders

async

Conceptually, async very similar to launch. It starts a separate coroutine that works concurrently with all the other coroutines.

However, instead of returning a job, async returns a Deferred object.

This is a lightweight 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.

`async` is useful when you want to run multiple independent tasks and have then return when they are ready.

Coroutine Builders

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

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

val time = measureTimeMillis {
  val one = async { doOne() }
  val two = async { doTwo() }
  println("The answer is ${one.await() + two.await()}") // wait
}
println("Completed in $time ms")

// The answer is 42
// Completed in 1017 ms
```

See Kotlin Play demo

Dispatchers

When you create a coroutine, you can optionally designate a dispatcher, which specifies which thread to use.

- If you don't specify it, the coroutine will inherit the parent's dispatcher.

Why use this?

- You shouldn't perform blocking IO operations on the default thread pool!
- Kotlin provides pre-allocated threads for blocking operations, use those.

```
launch (Dispatchers.Default) {
  // do some work using that dispatcher
}
```

Options include:

- `Dispatchers.Default` - Pool meant for computation (threads = # cores)
- `Dispatchers.IO` - Pool meant for blocking IO operations (threads = 64+)
- `Dispatchers.Main` - User Interface thread (threads = 1)

Dispatchers

You can switch dispatchers if needed. This is rarely needed.

When does it matter?

- Blocking IO operations should be done on `Dispatchers.IO` (pool of threads).
- User interface operations should be done on `Dispatchers.Main` (UI thread).

```
runBlocking(Dispatchers.Default) {
  launch (Dispatchers.IO) {
    // blocking operation
    withContext (Dispatchers.Main) {
      // UI operation
    }
  }
}
```

Dispatchers

When a coroutine builder is used without parameters, it inherits the context (and thus dispatcher) from the parent coroutine.

```
runBlocking(Dispatchers.Default) {
  launch {  // will inherit Dispatchers.Default from runBlocking
    delay (5000.milliseconds)
  }
}
```

The `withContext()` function is also commonly used to execute a suspending function using a particular dispatcher.

```
withContext(Dispatchers.IO) { // do something here }
```

Structured Concurrency

In a real application, you will be launching a lot of coroutines.

Structured concurrency is the ability to track and manage the hierarchy of coroutines in your application. This is useful to manage related coroutines.

- e.g., imagine that when a user switches screens, you launch multiple coroutines to load data from different sources. If they navigate back and cancel the screen load, you should be able to cancel the coroutines together.

Structured concurrency ensures that coroutines are never lost and do not leak.

- An outer scope cannot complete until all its children coroutines complete.
- A child coroutine throwing an exception will cause other coroutines in the same scope to stop executing as well.

Coroutine Scope

Each coroutine belongs to a coroutine scope. When you create a new coroutine e.g., launch or async, it will automatically become a child of that coroutine.

```
fun main() {
  runBlocking { // 1
    launch { // 2
      delay(1.seconds)
      launch { // 4
        delay(250.milliseconds)
        log("Grandchild done")”
      }
      log("Child 1 done!")
    }
    launch { // 3
      delay(500.milliseconds)
      log("Child 2 done!")
    }
    log("Parent done!")
  }
}
```

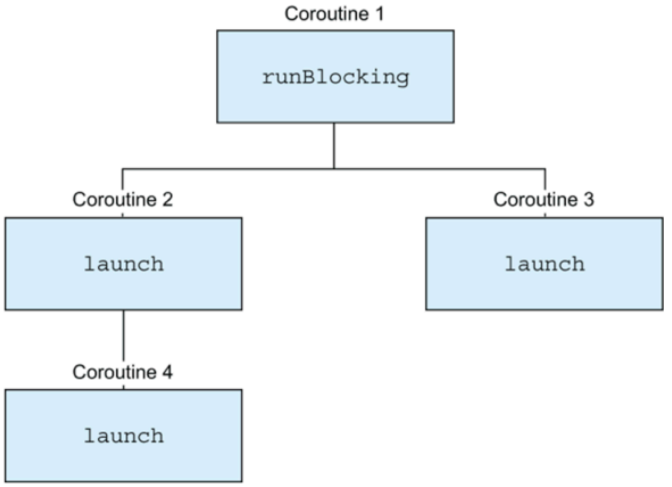


Figure 9: Nested coroutines follow the principles of structured concurrency.

```
> Task :CoroutineScopeKt.main()
40 [main] Parent done!
555 [main] Child 2 done!
1055 [main] Child 1 done!
1308 [main] Grandchild done
```

Coroutine Scope

Coroutine builders create scope, but you can also create it manually.

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

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

A coroutine scope can be defined inside of any suspending function. Here we use it to launch 2 concurrent coroutines.

```
Hello
World 1
World 2
Done
```

See Kotlin Play demo

Coroutine Scope

When you cancel a coroutine, it's children are also cancelled.

```
fun main() = runBlocking {
  val job = launch {
    launch {
      launch {
        launch {
          log("I'm started")
          delay(500.milliseconds)
          log("I'm done!")
        }
      }
    }
  }
  delay(200.milliseconds)
  job.cancel() // everything is cancelled
}
```

Debugging

Add this line to the `VM options` section of your Run Configuration in IntelliJ IDEA, to include coroutine debug information in the output.

```
-Dkotlinx.coroutines.debug
```

```
> Task :LaunchKt.main()
35 [main @coroutine#1] The
first, parent, coroutine starts
41 [main @coroutine#1] The first
coroutine has launched two more
coroutines
42 [main @coroutine#2] The
second coroutine starts and is
ready to be suspended
45 [main @coroutine#3] The third
coroutine can run in the
meantime
```

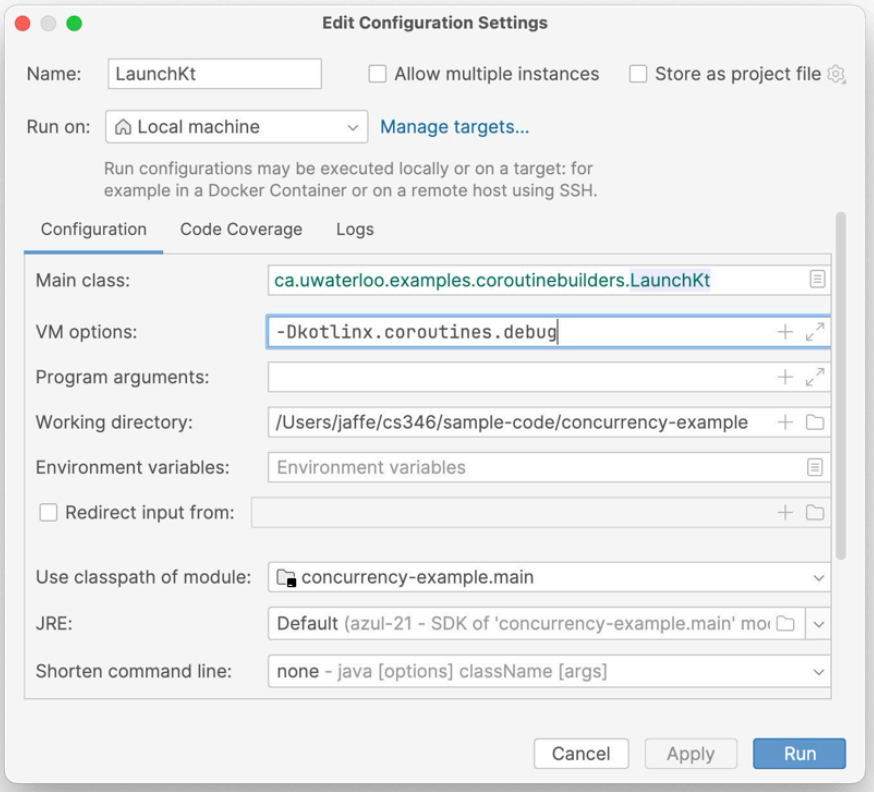


Figure 10: Run Configuration Window.

Bibliography

[1] T. Hoare, “Communicating Sequential Processess,” *Communications of the ACM*, vol. 21, no. 8, 1978.

[2] S. Aigner, R. Elizarov, S. Isakova, and D. Jemerov, *Kotlin in Action*, Second. New York, USA: Manning Publications, 2024. [Online]. Available: https://www.manning.com/books/kotlin-in-action-second-edition

[3] R. Pike, *Concurrency Is Not Parallelism*, (2012). [Online Video]. Available: https://vimeo.com/49718712

[4] JetBrains Inc., “Kotlin Documentation.” [Online]. Available: https://kotlinlang.org/