

Concurrency

CS 346 Application
Development

Synchronous execution is “normal”

```
fun calculate(): Int {  
    val r1 = square(2)  $\longleftrightarrow$  fun square(n: Int): Int = n * n  $\rightarrow$  4  
    val r2 = square(5)  $\longleftrightarrow$  fun square(n: Int): Int = n * n  $\rightarrow$  25  
    val r3 = square(7)  $\longleftrightarrow$  fun square(n: Int): Int = n * n  $\rightarrow$  49  
    return r1 + r2 + r3  
}
```


Each call to the square() function needs to return its result before the next invocation. This is synchronous, since operations need to be performed in-order.

In practice, we typically program using a **synchronous execution model**, where we expect the CPU to wait for one instruction to complete before proceeding to the next. This works fine for simple programs, but it doesn't scale *at all*.

- e.g., imagine a web server that had to finish one query before handling the next one... not useful!

We want “asynchronous execution”

```
@Composable
fun loadCustomerRecord(id: customerID, transactionsTable: Table, customerTable: Table) {
    val customer = customerTable.fetch(id) // very fast operation
    val transactions = transactionsTable.fetch(customer) // very slow operation
    Column {
        Row {
            CustomerRow(customer)
            items(transactions) { item ->
                TransactionRow(item)
            }
        }
    }
}
```



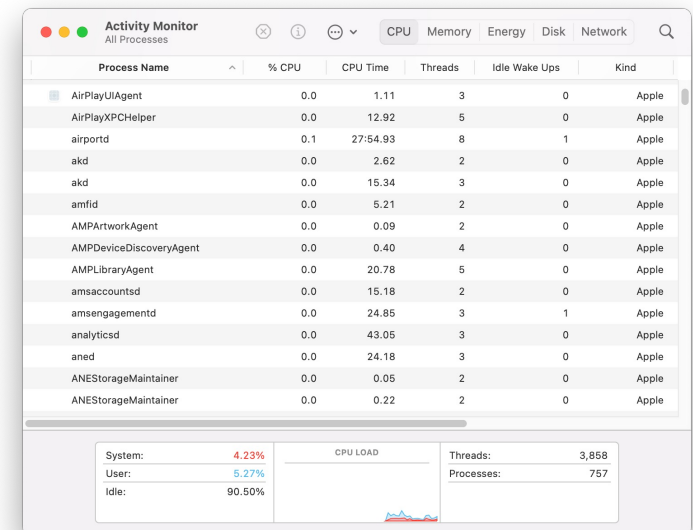
If `customer` data returns first, we want to display it immediately and not have to wait for `transactions` to return.

We want an **asynchronous execution model** where multiple tasks can be handled concurrently.

e.g., an application might need to make 3-4 database requests. You wouldn't want to have to wait for the first to complete before proceeding to the second (which would be synchronous). Ideally, you launch them all and they return as they complete.

Asynchronous execution is everywhere

- Your computer already does this!
- You probably have multiple applications running, each with many smaller tasks.
 - The operating system schedules time for each task, and alternates between them.
 - Often these tasks need to share resources, which the OS also needs to coordinate e.g., file, screen access.
- A single application might “simultaneously”:
 - Respond to a mouse-click to resize a window,
 - Draw the results to the screen, while
 - Reading data from a database, and
 - Fetching more data from the web.
- *We need a programming model that supports asynchronous execution.*

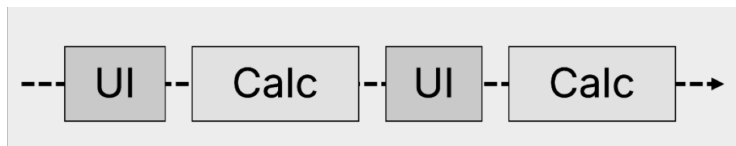


My computer, as I'm creating this slide, with 757 running processes. I don't know what most of these are...

Concept: Concurrency

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

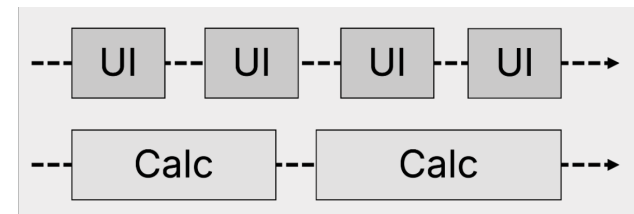
This does not suggest that tasks are being executed at the same time, only that we can *alternate between them easily*.



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.

Parallelism means *executing* or performing multiple tasks simultaneously.

Parallel computations can use multi-core hardware effectively, often making them more efficient (i.e. one task per core).



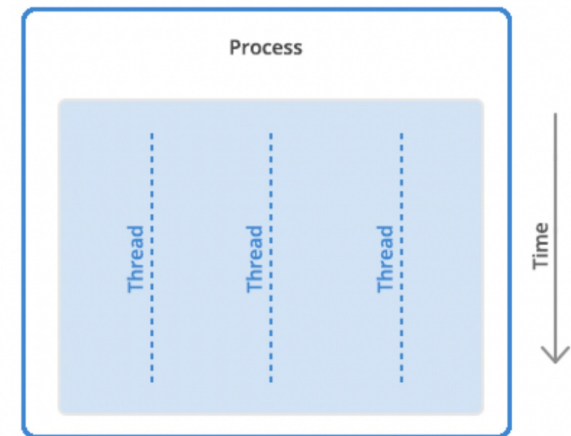
In this example, long-running calculations happen in the background while the UI is being rendered. This is parallelism, since operations happen simultaneously.₅

Threads

It always comes back to OS threads...

Concept: Threads

- Each program that you run is a **process** 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).

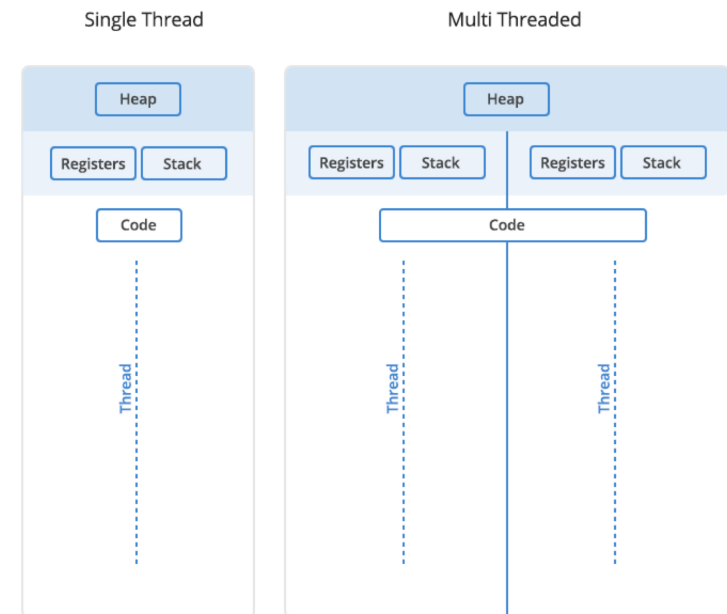


All instructions in a program are processed by one or more threads. Most programs only have one thread.

<https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/>

Threads for Background Processing

- 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
 - i.e. concurrency and/or parallelism



Make sure that threads don't compete for resources!

Managing a Thread

```
fun thread(  
    start: Boolean = true,  
    isDaemon: Boolean = false,  
    contextClassLoader: ClassLoader? = null,  
    name: String? = null,  
    priority: Int = -1,  
    block: () -> Unit  
): Thread  
  
thread(start = true) { // code to run in the lambda  
    println("${Thread.currentThread()} has run.")  
}
```

<https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.concurrent/thread.html>

Should we use threads?

- A worker thread can allow us to execute work in parallel. This is huge.
- However, there are disadvantages.
 - Threads are “expensive” to start/stop and consume lots of memory.
 - Launching or context switching between OS threads takes significant time.
 - Worker threads may not always be available in the number we require.
 - A blocking operation prevents any further work on the thread.
 - e.g., while you wait for a DB call to return, the thread is “hung” and unusable.
 - Threads are a poor, low-level abstraction.
 - We have limited ability to control when/how the OS runs threads, so we risk race conditions if we are not careful.
 - Your program execution model can be difficult to follow! You can’t just read code sequentially to understand the program’s logic.

Coroutines

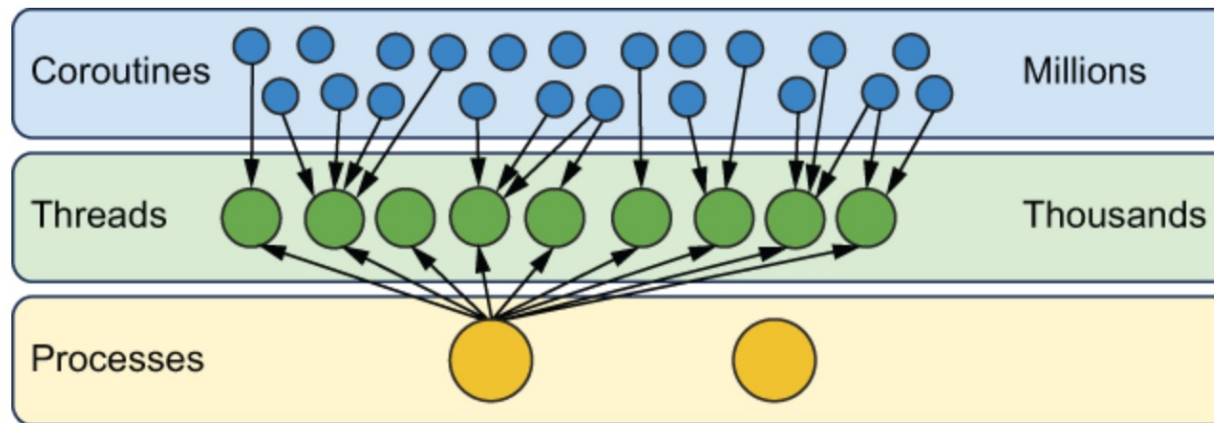
How to manage concurrency in Kotlin.

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 later.
 - Suspending a block of code allows the OS to execute something else while waiting.
- This abstraction allows us to describe how our code should be executed, without worry about the explicit allocation of work to threads.
- We achieve concurrency, plus parallelism under certain conditions.

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.



Aigner et al. **Kotlin in Action** (2024).

Importing Coroutine Libraries

Kotlin provides the [kotlinx.coroutines](https://kotlinlang.org/docs/coroutines-guide.html) library with high-level coroutine-enabled primitives. You will need to add the dependency to your `build.gradle.kts` file and then import the library.

```
// build.gradle.kts
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2")

// code
import kotlinx.coroutines.*
```

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

Structure of a coroutine

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

1. **A coroutine**, which is 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.**
 - Coroutines
 - “Special functions”

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

1. Creating a coroutine

A coroutine is generated by a **coroutine-builder**. The coroutine provides a context in which asynchronous code can run.

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

- It creates a coroutine, which executes the code passed to it.
- 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()  
    }  
    regularCode()  
}
```


2. Providing asynchronous code

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

- 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.
- A suspending function looks like a regular function with the "suspending" keyword.

GitLab:

[sample-code > coroutines-lecture > coroutinebuilders](#)

```
fun main() {  
    runBlocking {  
        // suspending functions  
        doSomething()  
        doSomethingElse()  
    }  
}  
  
suspend fun doSomething() {  
    delay(1000.milliseconds)  
    println("doSomething is done")  
}  
  
suspend fun doSomethingElse() {  
    delay(1000.milliseconds)  
    println("doSomethingElse is done")  
}
```

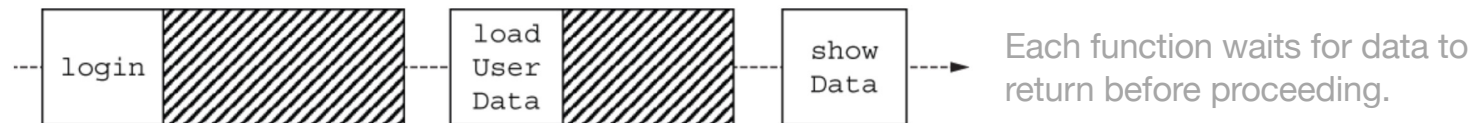
Issues with Regular Functions

Regular functions block their thread when waiting.

- In this example, the program “hangs” while waiting for each function call to return.
- This code can run in a coroutine but cannot be managed and blocks the thread.

```
fun login(credentials: Credentials): UserID // blocking function, takes time
fun loadUserData(userID: UserID): UserData // blocking function, takes time
fun showData(data: UserData) // #1

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



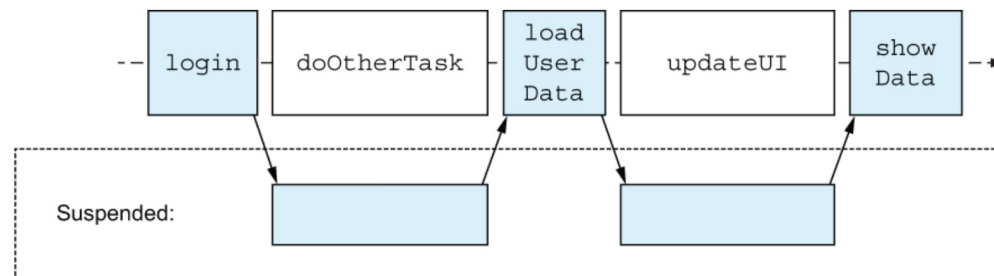
Suspending Functions

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

- They act as "suspension points" for the coroutine.

```
suspend fun login(credentials: Credentials): UserID // suspending, so no blocking
suspend fun loadUserData(userID: UserID): UserData // suspending, so no blocking
fun showData(data: UserData)
```

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



This changes the execution to be asynchronous. The OS can perform other actions while our functions suspend.

Coroutine Builders

Creating and executing coroutines.

Coroutine Builders

- **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.

launch

“Fire and forget” coroutine builder since it doesn’t return a value.

```
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 is the output from the second and third coroutines?

<https://pl.kotl.in/SE0hz4S4h>

```

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

```

runBlocking wraps everything in a parent coroutine, which will pause and wait for its children to complete before proceeding.

```

// 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

```

Managing a 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:

```
val job = launch { // launch a new coroutine and keep a reference
    delay(1000L)
    println("World!")
}
println("Hello")
job.join() // wait until child coroutine completes
println("Done")
```

<https://pl.kotl.in/t3nXouzWf>

Cancelling a job

```
val job = launch {  
    repeat(1000) { i ->  
        println("job: I'm sleeping $i ...")  
        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.")
```

```
// output  
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.
```

<https://pl.kotl.in/aCPfkC5Hm>

async (1/2)

- Conceptually, [async](#) is just like [launch](#). It starts a separate coroutine which is a light-weight thread that works concurrently with all the other coroutines. The differences:
 - `async` returns a [Deferred](#) — 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 them return when they are ready.

async (2/2)

```
suspend fun doSomethingUsefulOne(): Int {  
    delay(1000L) // pretend we are doing something useful here  
    return 13  
}  
  
suspend fun doSomethingUsefulTwo(): Int {  
    delay(1000L) // pretend we are doing something useful here, too  
    return 29  
}  
  
val time = measureTimeMillis {  
    val one = async { doSomethingUsefulOne() }  
    val two = async { doSomethingUsefulTwo() }  
    println("The answer is ${one.await() + two.await()}")  
}  
println("Completed in $time ms")  
  
// The answer is 42  
// Completed in 1017 ms
```

<https://pl.kotl.in/OXdh6hEup>

Debugging Tip!

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
```

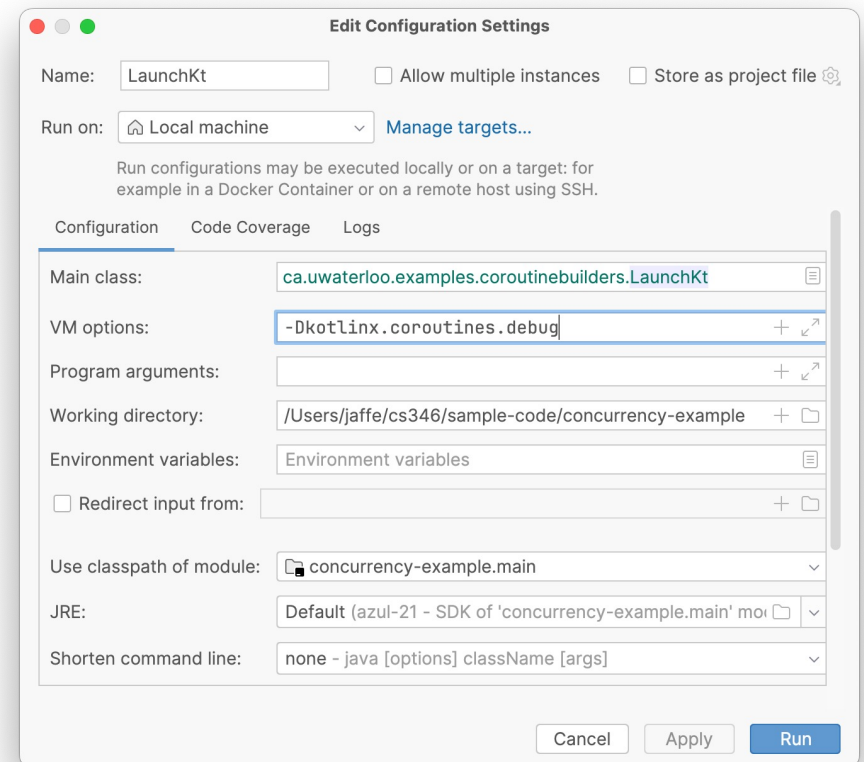
```
> Task :LaunchKt.main()
```

```
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
```



Dispatchers

Specifying which thread your coroutine will use.

Dispatchers

When you create a coroutine, you can optionally designate a **coroutine dispatcher**, which determines which thread will be used.

- 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!

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

Options include:

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

Mixing dispatchers

For most small computations, it won't matter which dispatcher use you.

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) {  
        // fetch from a database, which blocks normally  
        withContext (Dispatchers.Main) {  
            // update the UI directly within the same coroutine  
        }  
    }  
}
```

withContext

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

Providing safety with coroutine scope.

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.

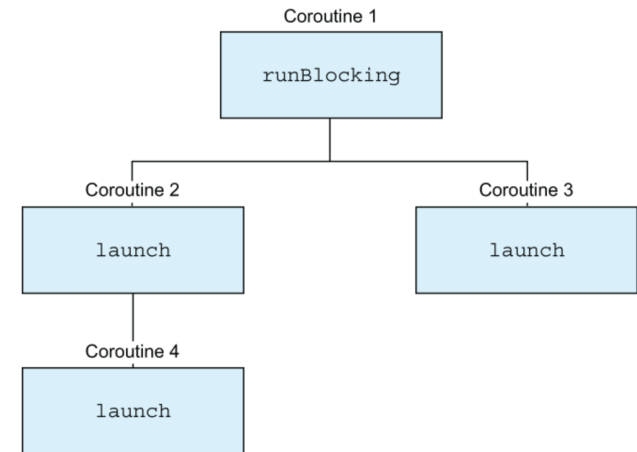
Structured concurrency ensures that coroutines are never lost and do not leak.

- 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!")  
    }  
}
```



Output

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

`runBlocking` won't complete until its children are done, thanks to structured concurrency.

Coroutine Scope Builder

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

```
// Executes doWorld followed by "Done"
fun main() = runBlocking {
    doWorld()
    println("Done")
}

// Concurrently executes both sections
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.

NOTE the ordering!

Output

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

<https://pl.kotl.in/PIZRdoh02>

Scope provides safety

```
suspend fun onMessage(msg: Message) = coroutineScope {  
    val ids: List<Int> = msg.getIds()  
  
    ids.forEach { id ->  
        // launch is called on the coroutineScope.  
        launch { restService.post(id) }  
    }  
}
```

See [“The reason to avoid GlobalScope”](#).

We’ve added a top-level coroutine scope.

If anything crashes, then ALL of the coroutines are cancelled.

Scope allows cancellation

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()  
}
```

Kotlin Flows

Leveraging coroutines to support a new programming model!

Streams of values

A suspending function can execute code asynchronously i.e. it can suspend/resume.

- However, a suspending function otherwise behaves *like any other function*.
- Return values are only delivered after the function completes.

In this example, we build a list of values and then return it once the function completes.

- We cannot deliver the first value early but must wait for the function to be “done”.
- *What if we wanted to deliver values as they were produced instead?*

```
suspend fun createValues(): List<Int> {  
    return buildList {  
        add(1)  
        delay(1.seconds)  
        add(2)  
        delay(1.seconds)  
        add(3)  
        delay(1.seconds)  
    }  
}  
  
fun main() = runBlocking {  
    val list = createValues()  
    list.forEach {  
        log(it)  
    }  
}  
  
// output  
// 3099 [main @coroutine#1] 1  
// 3107 [main @coroutine#1] 2  
// 3107 [main @coroutine#1] 3
```


Kotlin Flows

Reactive programming is a declarative programming paradigm that is based on the idea of asynchronous event processing and data streams.

- As a model, it supports passing processing data asynchronously and delivering it over time.

Kotlin flows are a coroutine-based abstraction that supports reactive programming.

- Think of it as a “pipeline” architecture with a producer and consumer.
 - One function emits values, and another function consumes them.
- Like RxJava, but for Kotlin. Leverages coroutines nicely.

Example: Flow

```
fun createValues(): Flow<Int> {  
    return flow {  
        emit(1)  
        delay(1000.milliseconds)  
        emit(2)  
        delay(1000.milliseconds)  
        emit(3)  
        delay(1000.milliseconds)  
    }  
}  
  
fun main() = runBlocking {  
    val myFlowOfValues = createValues()  
    myFlowOfValues.collect { log(it) }  
}
```



Keywords

- “flow” is a coroutine builder that creates a Flow coroutine.
- “emit” places items into the Flow.
- “collect” on the returned value.

```
// output  
// published as they are emitted  
29 [main] 1  
1040 [main] 2  
2045 [main] 3
```

Types of Flows

- **Cold flows** represent data streams that only start emitting items when the items will be consumed and processed by a collector. If “nobody” is listening, then values won’t be emitted.
- **Hot flows** produce items independently of whether anyone is collecting them i.e., they broadcast “blindly”, and items will be “lost” if no one is listening.

Cold Flows

- The `flow` keyword creates a cold flow **emitter**.
- Normally you define an associated **collector** for each flow.
 - The collector captures emitted values using the `collect` function.
 - The lambda that you pass to the collector is executed every time a value is emitted.
 - It only starts emitting when the collector starts collecting.

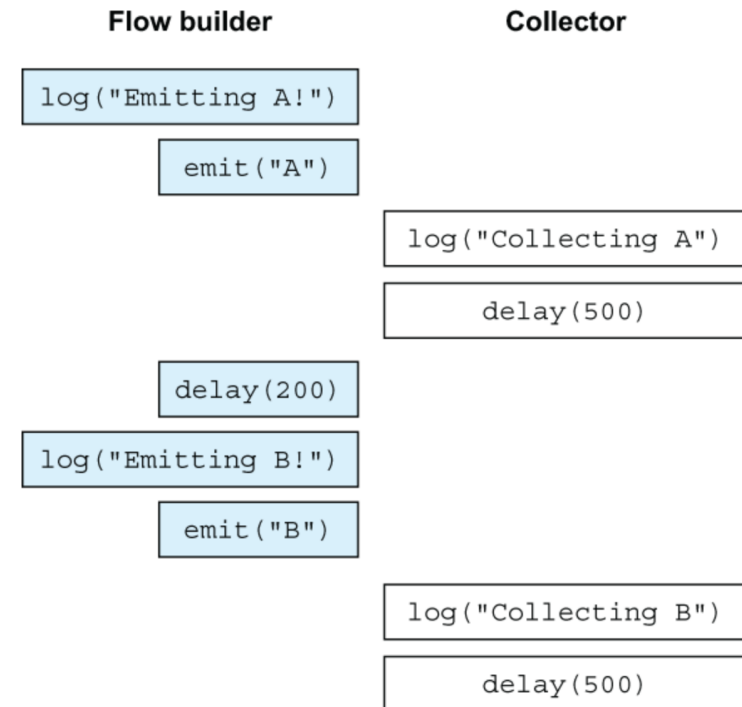
```

val letters = flow {
    log("Emitting A!")
    emit("A")
    delay(200.milliseconds)
    log("Emitting B!")
    emit("B")
}

fun main() = runBlocking {
    letters.collect {
        log("Collecting $it")
        delay(500.milliseconds)
    }
}

// output
18 [main] Emitting A!
25 [main] Collecting A
741 [main] Emitting B!
742 [main] Collecting B

```



Cold flow example from [Kotlin in Action, 2nd Ed.](#)
 The flow won't start until the collector starts to collect the emitted values.

Infinite Flow

Flows can be “infinite”, meaning that they won’t terminate.

- In this case it’s a cold flow, so if you stopped collecting it would stop emitting.

```
val counterFlow = flow {  
    var x = 0  
    while (true) {  
        emit(x++)  
        delay(200.milliseconds)  
    }  
}  
  
launch {  
    log("Starting infinite flow")  
    counterFlow.collect {  
        log("> collected $it")  
    }  
}
```

```
// output  
31 [main] > collected 0  
236 [main] > collected 1  
438 [main] > collected 2  
643 [main] > collected 3  
845 [main] > collected 4  
1046 [main] > collected 5  
1251 [main] > collected 6
```

Multiple Collectors

If you attempt to collect a flow multiple times, the flow will be executed for every collection e.g., the second collection suspends until the first completes.

```
fun main() = runBlocking {  
    letters.collect {  
        log("(1) Collecting $it") // 1  
        delay(500.milliseconds)  
    }  
    letters.collect {  
        log("(2) Collecting $it") // 2  
        delay(500.milliseconds)  
    }  
}  
  
val letters = flow {  
    log("Emitting A!")  
    emit("A")  
    delay(200.milliseconds)  
    log("Emitting B!")  
    emit("B")  
}  
}
```

```
// output  
0 [main] Emitting A!  
8 [main] (1) Collecting A  
720 [main] Emitting B!  
720 [main] (1) Collecting B  
1229 [main] Emitting A!  
1230 [main] (2) Collecting A  
1937 [main] Emitting B!  
1937 [main] (2) Collecting B
```

Cancelling Collectors

If you cancel the coroutine of the collector, you stop the collection of the flow at the next cancellation point.

```
fun main() = runBlocking {  
    val collector = launch {  
        counterFlow.collect {  
            println(it)  
        }  
    }  
    delay(5.seconds)  
    collector.cancel()  
}  
  
val counterFlow = flow {  
    var x = 0  
    while (true) {  
        emit(x++)  
        delay(200.milliseconds)  
    }  
}
```

// output halts at 24
0
1
2
3
...
24

Hot Flows

Hot flows share emitted items across multiple collectors, called **subscribers**.

- They are suitable when you are emitting events or state changes in your system that happen independently or aren't tied to a specific collector.
- Emissions happen even with no subscribers present

Kotlin coroutines come with two hot flow implementations out of the box:

- **Shared flows**, which are used for broadcasting values, and
- **State flows**, for the special case of communicating state

A shared flow broadcasts values

Shared flows operate in a broadcast fashion. All subscribers will receive the same values.

```
fun main() = runBlocking {
    RadioStation().beginBroadcasting(this)
}

class RadioStation {
    private val _messageFlow = MutableSharedFlow<Int>()
    val messageFlow = _messageFlow.asSharedFlow()

    fun beginBroadcasting(scope: CoroutineScope) {
        scope.launch {
            while(true) {
                delay(500.milliseconds)
                val number = Random.nextInt(0..10)
                log("Emitting $number!")
                _messageFlow.emit(number)
            }
        }
    }
}
```



Hide the mutableFlow behind a RO getter

```
// output
554 [main] Emitting 0!
1061 [main] Emitting 0!
1562 [main] Emitting 8!
2066 [main] Emitting 10!
2571 [main] Emitting 10!
```

Subscriptions

```
fun main(): kotlin.Unit = runBlocking {  
    val radioStation = RadioStation()  
    radioStation.beginBroadcasting(this)  
    delay(600.milliseconds)  
  
    launch {  
        radioStation.messageFlow.collect {  
            log("A collecting $it!")  
        }  
    }  
  
    launch {  
        radioStation.messageFlow.collect {  
            log("B collecting $it!")  
        }  
    }  
}
```

```
// output  
559 [main] Emitting 0!  
1066 [main] Emitting 9!  
1070 [main] A collecting 9!  
1071 [main] B collecting 9!  
1574 [main] Emitting 0!  
1574 [main] A collecting 0!  
1575 [main] B collecting 0!  
2080 [main] Emitting 0!  
2081 [main] A collecting 0!  
2081 [main] B collecting 0!
```

A state flow broadcasts a variable's state

A **state flow** is a special type of shared flow that makes it easy to track changes to a variable.

```
fun main() {  
    val vc = ViewCounter()  
    vc.increment()  
    vc.increment()  
    vc.increment()  
    println(vc.counter.value)  
}
```

// output
3

```
class ViewCounter {  
    private val _counter = MutableStateFlow(0)  
    val counter = _counter.asStateFlow()  
  
    fun increment() {  
        _counter.update { it + 1 }  
    }  
}
```



initial value



`update` makes the operation atomic, so even if multiple coroutines are trying to update at the same time, no data is lost.

Choosing a flow?

Cold flow	Hot flow
Inert by default (triggered by the collector)	Active by default
Has a collector	Has multiple subscribers
Collector gets all emissions	Subscribers get emissions from the start of subscription
Potentially completes	Doesn't complete
Emissions happen from a single coroutine (unless <code>channelFlow</code> is used).	Emissions can happen from arbitrary coroutines.

Often both will work. You should consider the behavior of the flow and if it matches your data monitoring requirements.

Reference

- Aigner et al. 2024. [Kotlin in Action](#), 2nd edition.
- Google. 2025. [Kotlin Flows on Android](#).
- Leeds. 2025. [Kotlin: An Illustrated Guide](#).
- Otta. 2025. [What is reactive programming?](#)