Concurrency

CS 346 Application Development

Synchronous execution

```
fun calculate(): Int {
    val r1 = square(2)
    val r2 = square(5)
    val r3 = square(7)
    return r1 + r2 + r3
}
```

```
fun square(n: Int): Int = n * n → 4
fun square(n: Int): Int = n * n → 25
fun square(n: Int): Int = n * n → 49
```

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

Programs normally adhere to a **synchronous execution model**, 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* any further execution until it returns a value.

Asynchronous execution

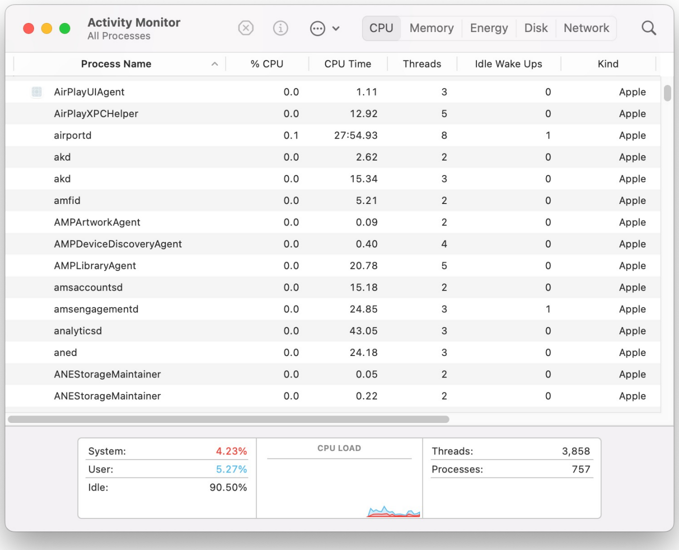
```
@Composable
fun loadCustomerRecord(id: customerID, transactionsTable: Table, customerTable: Table) {
    val customer = customerTable.fetch(id) // very fast operation
    val transactions = transactionsTable.fetch(customer) // very slow operation
    Column {
        Row {
            CustomerRow(customer)
            items(transactions) { item ->
                TransactionRow(item)
            }
        }
    }
    // how can we return data instead of waiting for “transactions” to complete?
}
```

We want *asynchronous execution*, where we can juggle multiple tasks.

- If `customer` data returns first, we want to display it and not have to wait for `transactions` to return.
- We could re-order our operations to achieve this e.g., fetch and display the customer first, but it would require major restructuring.

Hardware and OS support for this

- Your OS already runs things asynchronously!
- 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.
- 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.



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	Kind
AirPlayUIAgent	0.0	1.11	3	0	Apple
AirPlayXPCHelper	0.0	12.92	5	0	Apple
airportd	0.1	27:54.93	8	1	Apple
akd	0.0	2.62	2	0	Apple
akd	0.0	15.34	3	0	Apple
amfid	0.0	5.21	2	0	Apple
AMPArtworkAgent	0.0	0.09	2	0	Apple
AMPDeviceDiscoveryAgent	0.0	0.40	4	0	Apple
AMPLibraryAgent	0.0	20.78	5	0	Apple
amsaccountsd	0.0	15.18	2	0	Apple
amsengagementd	0.0	24.85	3	1	Apple
analyticsd	0.0	43.05	3	0	Apple
aned	0.0	24.18	3	0	Apple
ANEStorageMaintainer	0.0	0.05	2	0	Apple
ANEStorageMaintainer	0.0	0.22	2	0	Apple

System:	4.23%
User:	5.27%
Idle:	90.50%

Threads:	3,858
Processes:	757

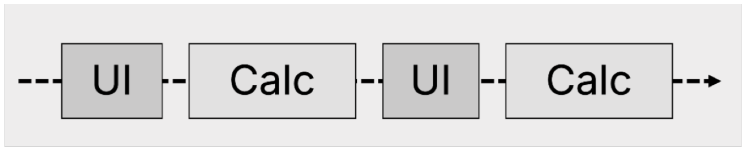
My computer, as I'm creating this slide, with 757 running processes. I don't know what most of these are…

We need a programming model that supports this within a single program.

Concept: Concurrency

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

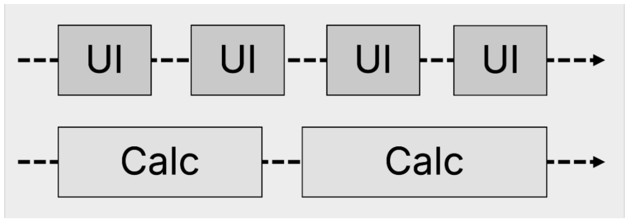
This does not suggest that tasks are being executed at the same time, only that we can *alternate between them easily*.



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).



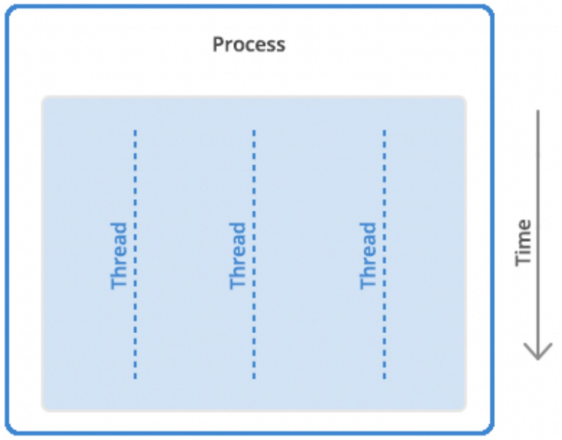
In this example, long-running calculations happen in the background while the UI is being rendered. This is parallelism, since operations happen simultaneously.

Threads

It always comes back to OS threads…

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).

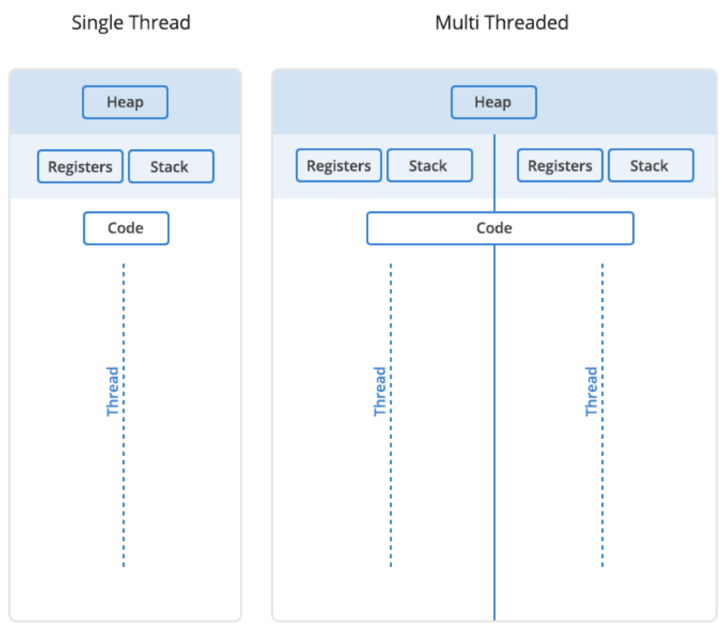


All instructions in a program are processed by one or more threads. Most programs only have one thread.

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

```kotlin
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

Can we manually use threads?

There are disadvantages to using threads directly:

- 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 available in the number we require.
- A blocking operation blocks a thread, preventing any further work.
  - e.g., while you want for a DB call to return, the thread is “hung”.
- Threads are a poor, low-level abstraction.
  - We have limited ability to control when/how threads run, so we risk race conditions.
- Threads make program execution difficult to follow!
  - You can’t just read code sequentially to understand the program’s logic.

Coroutines

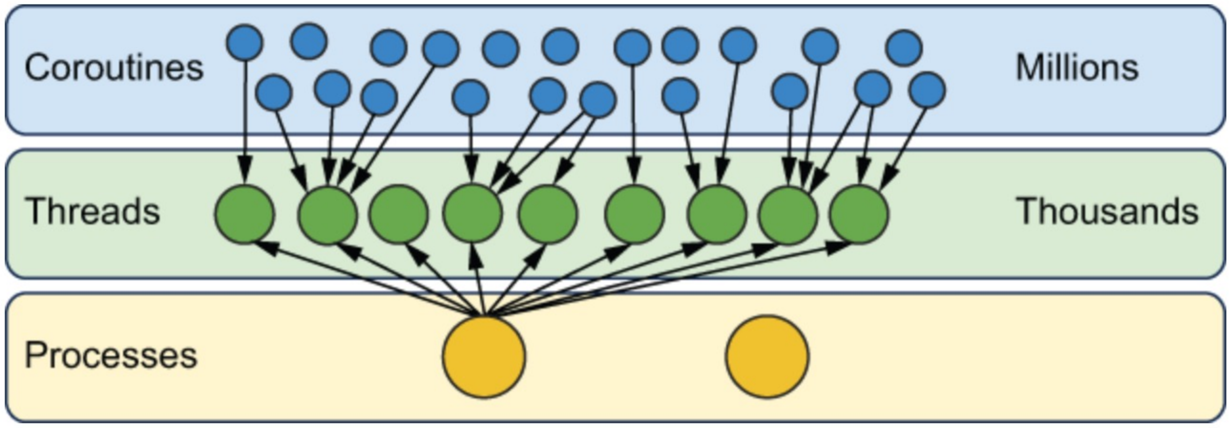
Kotlin’s concurrency model.

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 worrying about the 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 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 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.
   - 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**.

`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 “suspend” keyword.

lectures > coroutines > coroutinebuilders

https://pl.kotl.in/z85qR8Rj3

```
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")
}
```

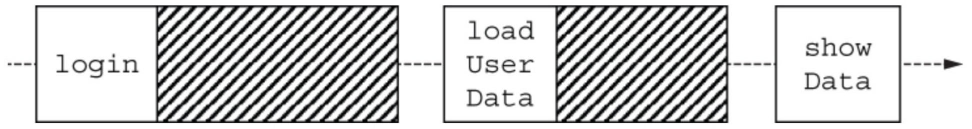
Issues with Regular Functions

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)
}
```



Each function waits, blocking the thread

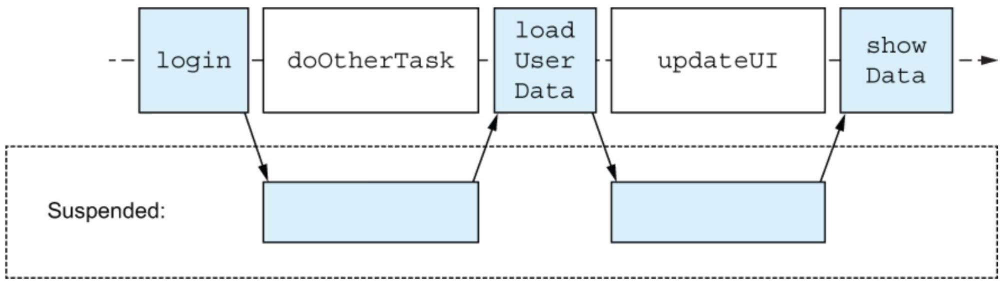
Suspending Functions

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)
}
```



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

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 coroutines?

https://pl.kotl.in/SEOhz4S4h

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

```
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")
}


// 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
```

https://pl.kotl.in/w_99PwHPE

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:

```kotlin
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 then 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()}").  ← pauses here
}
println("Completed in $time ms")


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

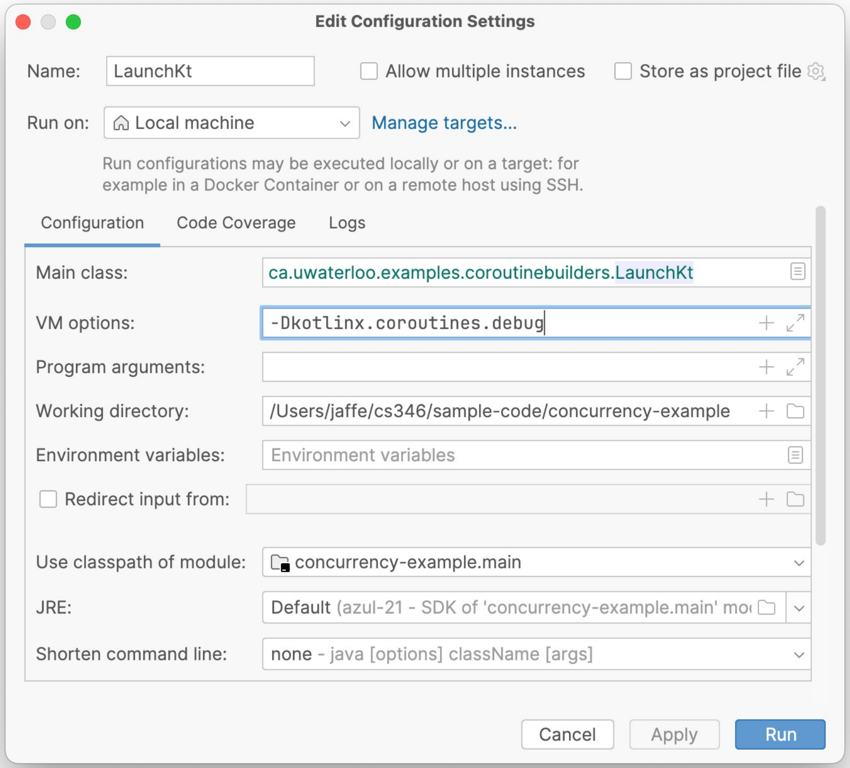
https://pl.kotl.in/OXdh6hEup

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
```



Edit Configuration Settings

Name: LaunchKt
☐ Allow multiple instances
☐ Store as project file

Run on: Local machine
Manage targets...

Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.

Main class: ca.uwaterloo.examples.coroutinebuilders.LaunchKt

VM options: -Dkotlinx.coroutines.debug

Program arguments:

Working directory: /Users/jaffe/cs346/sample-code/concurrency-example

Environment variables: Environment variables

☐ Redirect input from:

Use classpath of module: concurrency-example.main

JRE: Default (azul-21 - SDK of 'concurrency-example.main' mo

Shorten command line: none - java [options] className [args]

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.

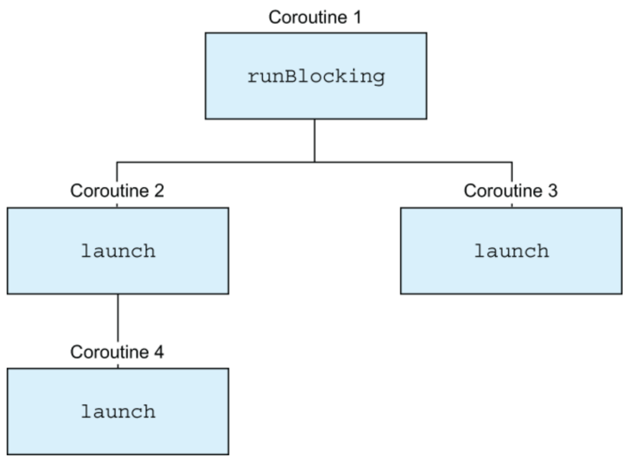
Structured concurrency ensures that coroutines are never lost and do not leak.

- 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()
  }
```