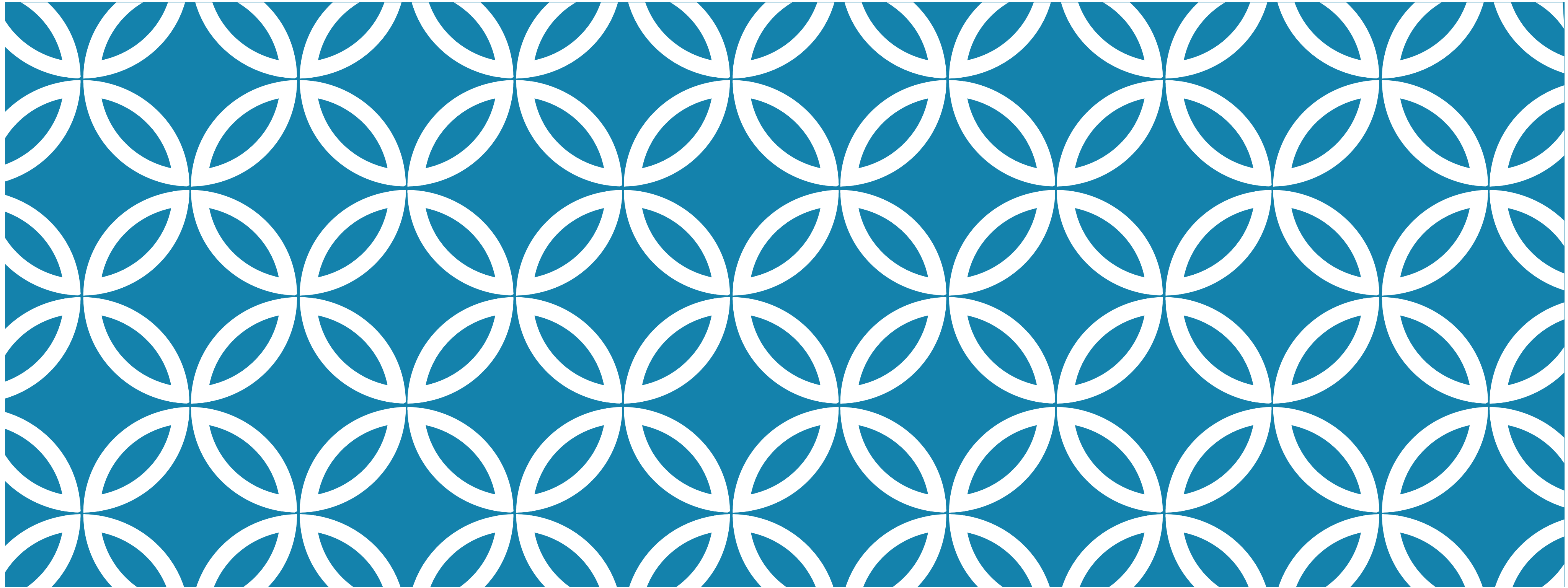


CONCURRENCY

CS 346: Application Development

INTRODUCTION

Modern computers are multi-tasking devices.

You probably have:

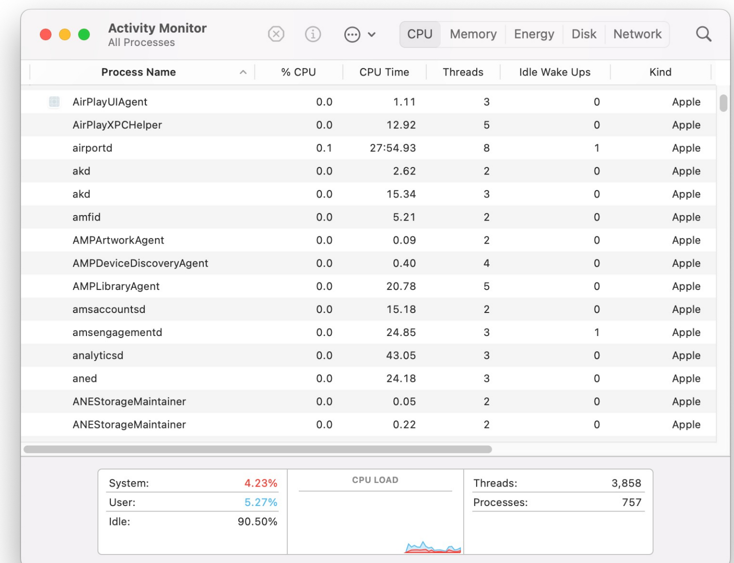
- Multiple processes running,
- Multiple tasks being performed by each task.

Your computer cannot run all these simultaneously!

The operating system *carefully* schedules time for each process, and alternates between them.

This can happen *within* a process as well. Your application might need to "simultaneously":

- Read data from a database,
- Draw to the screen,
- Respond to a mouse-click.

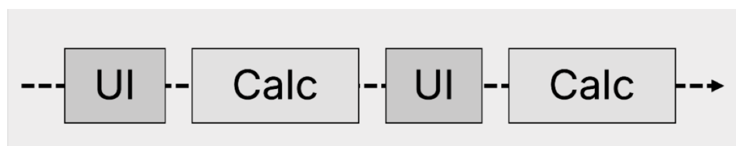


My computer, as I'm creating this slide, with *757* running processes. I don't know what most of these are…

CONCEPTS: CONCURRENCY

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

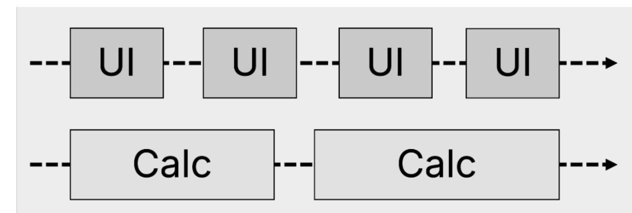
This does not suggest that they are executed at the same time, only that we can alternate between them easily.



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.



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

DEMO: PROGRAM EXECUTION

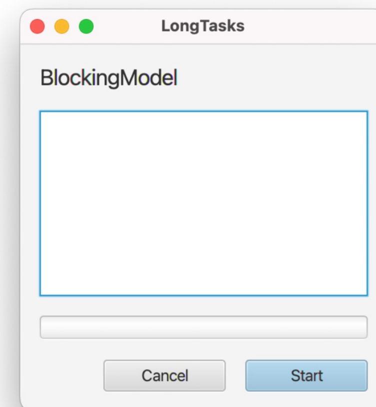
The normal mode of execution for any program, in any programming language, is *blocking*, sequential execution.

- Your program runs one instruction at a time.
- Any pauses in execution result in the entire application stalling.

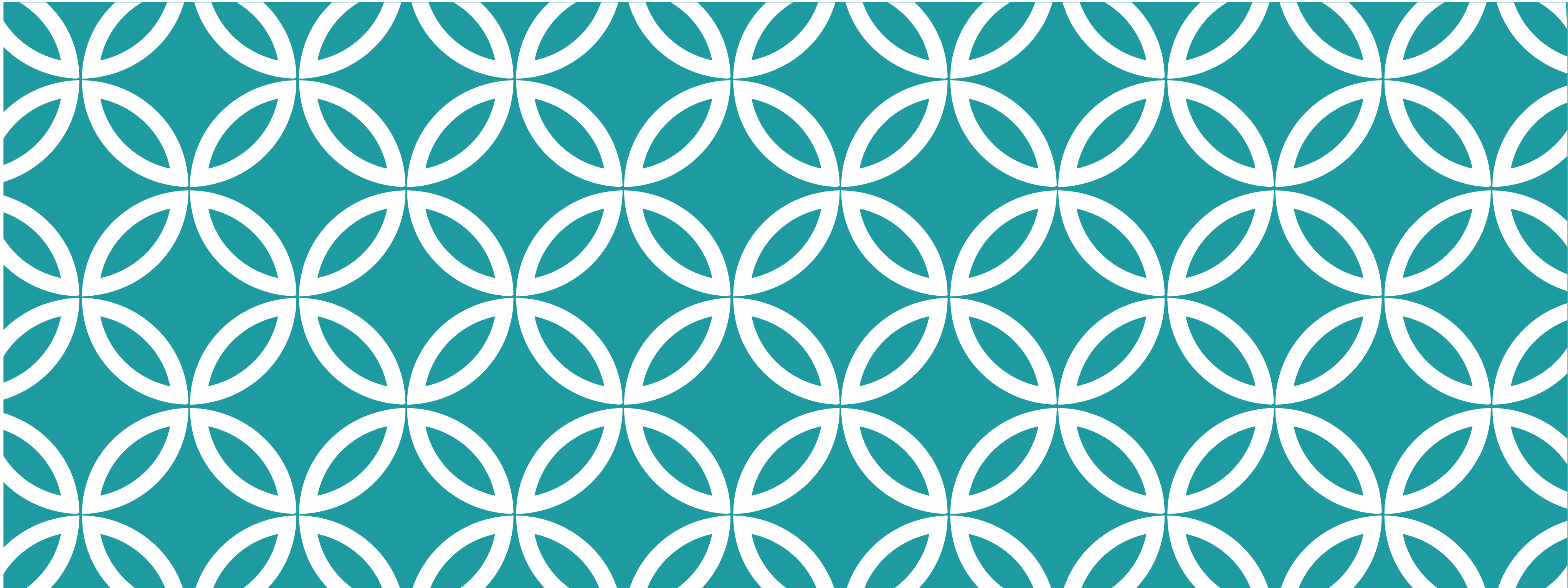
Let's consider three different execution models.

- **Blocking**: runs the entire computation at-once (as above).
- **Concurrent**: alternates between computation and display tasks.
- **Parallel**: runs computation and display tasks simultaneously.

Demo: what effect does the execution model have?



This application calculates prime numbers and displays them on the screen.



CONCURRENCY > THREADS

CS 346: Application Development

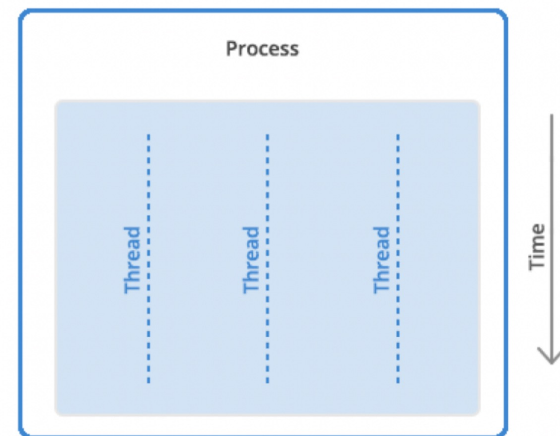
CONCEPT: THREADS

A typical way to achieve concurrency is via threads. A **thread** is a context within which the CPU can execute instructions.

- A program always has a **main thread**. By default, this executes all instructions in order.
- A program can have additional **worker threads**.

Threads are managed by the operating system; libraries provide a convenient abstraction.

As a developer, you explicitly create and manage threads and assign work.



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

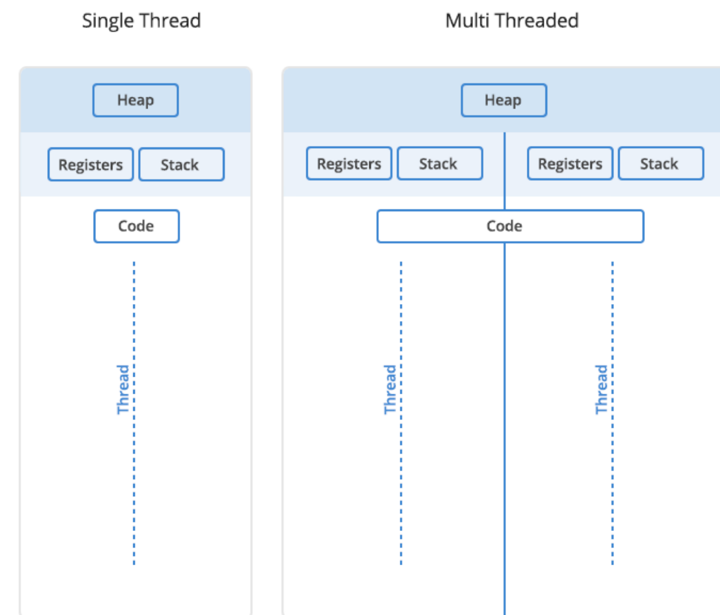
We can split computation across threads (one **primary** thread, and one or more **background** threads).

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 and have it done in parallel!

Concurrency + Parallelism

… *but be very very careful.*



Make sure that threads don’t compete for resources!

MANAGING A THREAD (JAVA)

```
val t = object : Thread() {
    override fun run() {
    // define the task here
    // it will run to completion on this thread
  }
}


t.start() // launch the thread, it will run to completion
t.stop()  // we can also stop it manually
```

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

MANAGING A THREAD (KOTLIN)

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


// provide arguments where you want a non-default value
thread(start = true) {
  // the thread will end when this block completes
  println("${Thread.currentThread()} has run.")
}
```

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

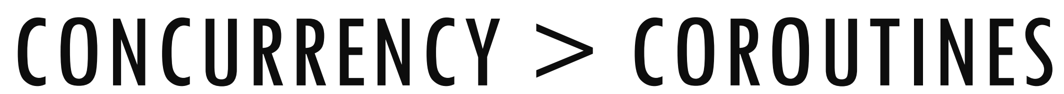
THREADS: PROS/CONS

Pro

- A worker thread can allow us to execute work in parallel.

Con

- Worker threads may not always available (in the number we require).
- We are required to explicitly control threads. It's easy to encounter race conditions with multiple threads in-play.
- We have limited ability to control when/how the OS runs threads.
- Shared state is difficult and error-prone to manage. You need to take great care to ensure that two threads are not accessing the same resources at the same time - see synchronization primitives.



CS 346: Application Development

INTRODUCTION

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

A **coroutine** is a *suspendable computation*: a function that can suspend its execution at some point, and then resume later.

Coroutines can be thought of as *light-weight threads*, with some advantages.

- Coroutines are lightweight and avoid the overhead (memory + performance) of creating and destroying threads.
- A coroutine is not tied to any thread. It may suspend its execution in one thread and resume in a different one.
- Coroutines can suspend without blocking resources.

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:1.6.0")
```

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

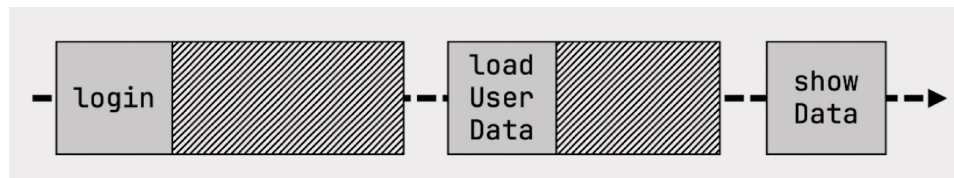
https://kotlinlang.org/docs/coroutines-guide.html

SUSPENDING FUNCTIONS (1/2)

In this example, the login and loadUserData are blocking functions.
We have to run them in order (and halt while waiting).

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

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



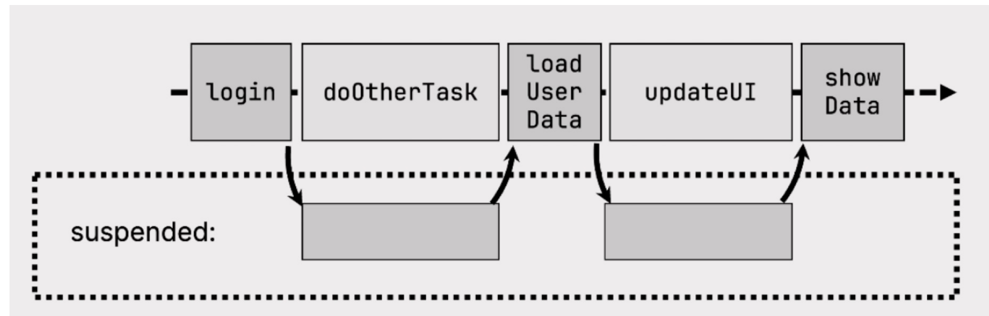
Each function waits for data to return before proceeding.

SUSPENDING FUNCTIONS (2/2)

Suspending functions can suspend themselves while waiting!

```
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)
    val userData = loadUserData(userID)
    showData(userData)
}
```



IMPORTANT: Suspending functions can only be called from other suspending functions or a coroutine.

The thread is then free to do other work while functions are suspended. e.g. no UI freezes.

COROUTINE BUILDERS (1/2)

A **coroutine** is an instance of a suspendable computation.

- A coroutine is a unit of execution that can be run concurrently or in parallel - much like a thread.

They manage suspending functions that they contain. Think of them as providing context to a suspending function.

To create a coroutine, you use a coroutine builder: `runBlocking`, `launch` or `async`.

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

suspend fun doWorld() {
    delay(1000L)
    println("World!")
}

// Hello
// World
```

COROUTINE BUILDERS (2/2)

runBlocking is a *coroutine builder* that bridges the non-coroutine world of a regular fun main() and the code with coroutines inside { }.

launch is also a *coroutine builder*. It launches a new coroutine concurrently with the rest of the code, which continues to work independently.

async is a coroutine builder, which returns a deferred object: a lightweight non-blocking *future* that represents a promise to provide a result later.

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

suspend fun doWorld() {
    delay(1000L)
    println("World!")
}

// Hello
// World
```

LAUNCH

Where is the output from the second and third coroutines?

```
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
```

https://pl.kotl.in/SE0hz4S4h

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

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:

- launch returns a Job and does not carry any resulting value
- 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)

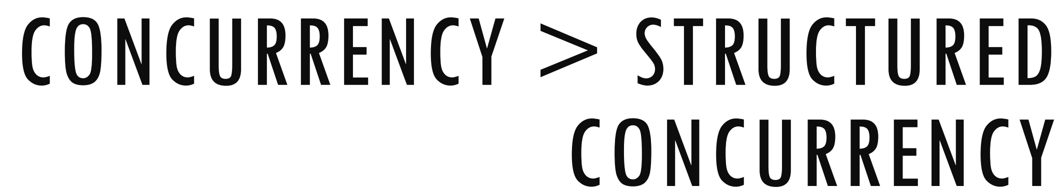
```kotlin
suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

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



CONCURRENCY > STRUCTURED CONCURRENCY

CS 346: Application Development

STRUCTURED CONCURRENCY

Coroutines follow a principle of **structured concurrency** which means that new coroutines can be only launched in a specific CoroutineScope.

- e.g., runBlocking establishes the corresponding scope and that is why the previous example waits until everything completes before exiting the program.

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

Structured concurrency ensures that coroutines are not 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 BUILDERS

In addition to the coroutine scope provided by different builders, it is possible to declare your own scope using the coroutineScope builder. It creates a coroutine scope and does not complete until all launched children complete.

runBlocking and coroutineScope builders may look similar because they both wait for their body and all its children to complete. The main difference is that

- runBlocking method *blocks* the current thread for waiting,
- coroutineScope suspends, releasing the underlying thread for other usages.

SCOPE BUILDER

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

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

// Concurrently executes both sections
suspend fun doWorld() = coroutineScope { // this: CoroutineScope
    launch {                             // coroutine 1
        delay(2000L)
        println("World 2")
    }
    launch {                             // coroutine 2
        delay(1000L)
        println("World 1")
    }
    println("Hello")
}
```

```
// output
Hello
World 1
World 2
Done
```

https://pl.kotl.in/PIZRdoh02

JOBS (1/2)

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

JOBS – CANCELLING (2/2)

```
coroutineScope {
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