

CS 398: Application Development

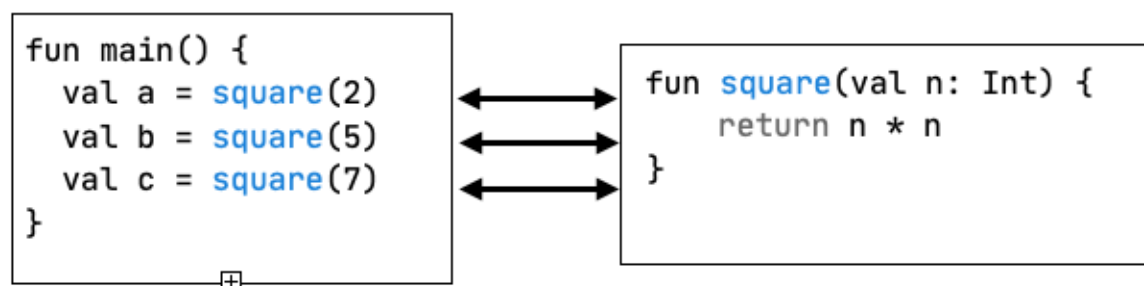
Asynchronous programming

Blocking; Threads; Callbacks; Promises; Coroutines.

Motivation

Programs typically consists of functions that we call in order to perform some operations. One underlying assumption is that functions will run to completion before returning control to the calling function.

Once called, a subroutine will **block** your program from executing any further until it completes.

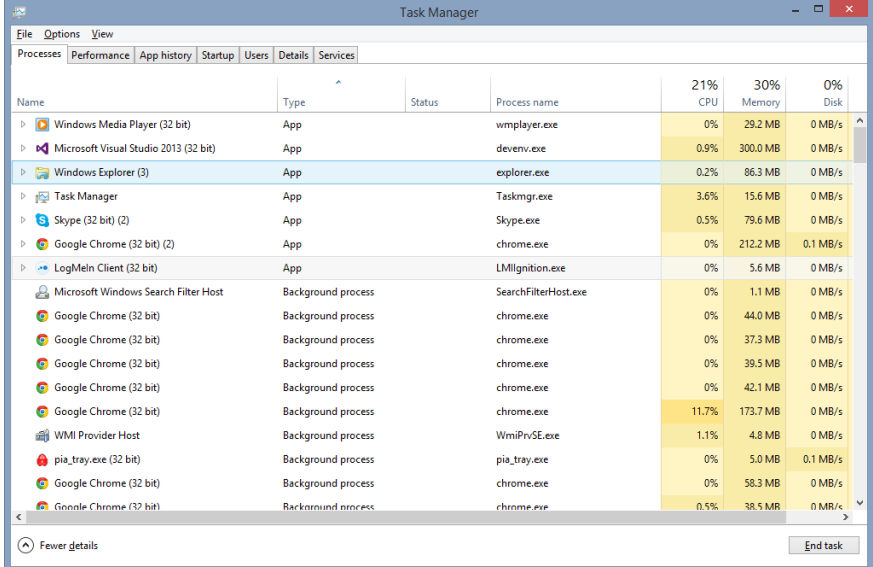


Each call to the `square()` function returns the result of that function running to completion. The calling function has to wait for the function call to return before the next call is executed.

Programs often have work to do that takes time to complete: reading a file, writing to a database, making a request over a network. We call these **blocking operations** since they essentially halt the program from running any further until the operations have completed.

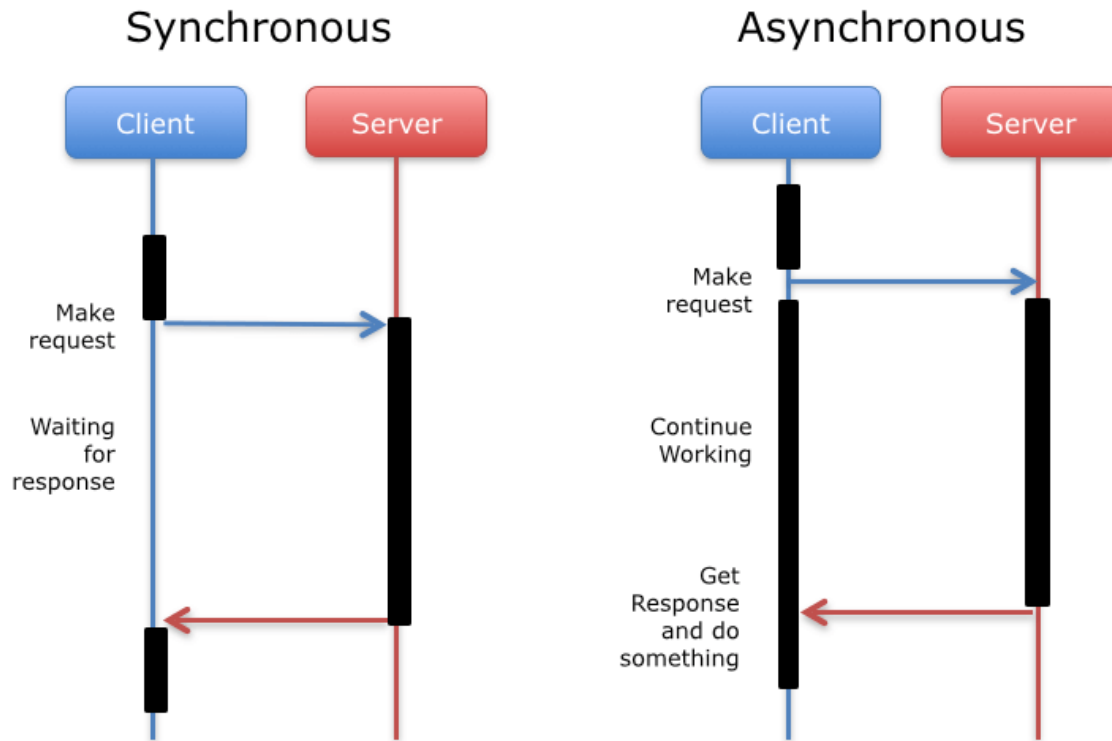
The solution is to design software so that long-running tasks can be run in the background, or **asynchronously**.

The Task Manager on Windows will show you all of the processes that are running simultaneously. It's normal for your computer to do a lot of background processing!



The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. The window title is 'Task Manager'. The menu bar includes 'File', 'Options', and 'View'. The tabs at the top are 'Processes', 'Performance', 'App history', 'Startup', 'Users', 'Details', and 'Services'. The main area displays a table of running processes with columns for Name, Type, Status, Process name, CPU, Memory, and Disk. The CPU column is expanded to show a total of 21% usage. The processes listed include various applications like Windows Media Player, Visual Studio, Explorer, Task Manager, Skype, Chrome, and LogMeIn, as well as several background processes like Search Filter Host and WMI Provider Host.

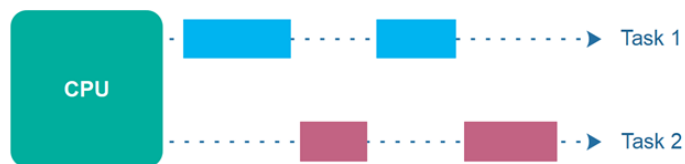
Name	Type	Status	Process name	21% CPU	30% Memory	0% Disk
Windows Media Player (32 bit)	App		wmplayer.exe	0%	29.2 MB	0 MB/s
Microsoft Visual Studio 2013 (32 bit)	App		devenv.exe	0.9%	300.0 MB	0 MB/s
Windows Explorer (3)	App		explorer.exe	0.2%	86.3 MB	0 MB/s
Task Manager	App		Taskmgr.exe	3.6%	15.6 MB	0 MB/s
Skype (32 bit) (2)	App		Skype.exe	0.5%	79.6 MB	0 MB/s
Google Chrome (32 bit) (2)	App		chrome.exe	0%	212.2 MB	0.1 MB/s
LogMeIn Client (32 bit)	App		LMIgnition.exe	0%	5.6 MB	0 MB/s
Microsoft Windows Search Filter Host	Background process		SearchFilterHost.exe	0%	1.1 MB	0 MB/s
Google Chrome (32 bit)	Background process		chrome.exe	0%	44.0 MB	0 MB/s
Google Chrome (32 bit)	Background process		chrome.exe	0%	37.3 MB	0 MB/s
Google Chrome (32 bit)	Background process		chrome.exe	0%	39.5 MB	0 MB/s
Google Chrome (32 bit)	Background process		chrome.exe	0%	42.1 MB	0 MB/s
Google Chrome (32 bit)	Background process		chrome.exe	11.7%	173.7 MB	0 MB/s
WMI Provider Host	Background process		WmiPrvSE.exe	1.1%	4.8 MB	0 MB/s
pia_tray.exe (32 bit)	Background process		pia_tray.exe	0%	5.0 MB	0.1 MB/s
Google Chrome (32 bit)	Background process		chrome.exe	0%	58.3 MB	0 MB/s
Google Chrome (32 bit)	Background process		chrome.exe	0.5%	38.5 MB	0 MB/s



Our goal is for our main program to continue executing while other work gets done in the background. In the example above, the client doesn't want to be blocked waiting for the server to return data.

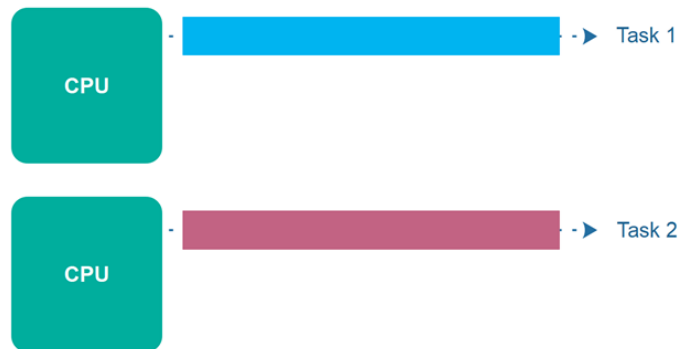
Source: <https://medium.com/i-learn-esp32-and-you-should-too/9-the-esp32-real-time-chart-591c0cbb03f>

We need to be clear about what we mean by running “in the background”. There is a difference between concurrent and parallel computation.



Concurrent tasks.

The processor can alternate between 2 tasks and make progress on both. No performance gain overall.



Parallel tasks.

The processor can have 2 tasks running simultaneously, usually by splitting computation across threads (and having multiple processors or cores on the system).

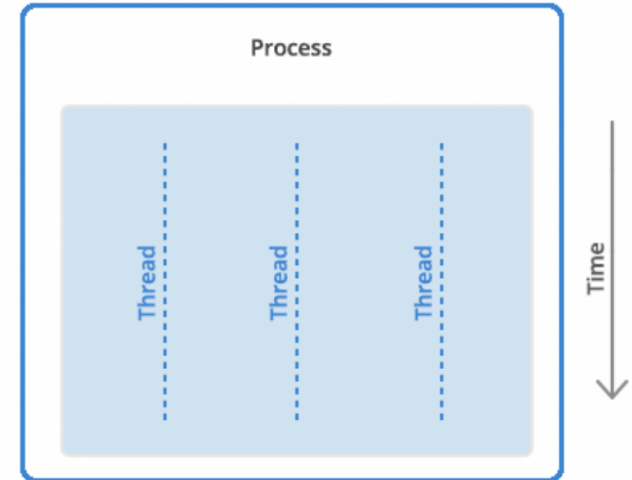
Do accomplish either, we need to understand threads.

A **thread** is a “stream” of execution, that handles executing instructions within a program.

Every program has a *main thread*, and may *optionally* have additional threads for background processing.

Every thread has its own instructions that it executes, and the processor is responsible for allocating time for each thread to run.

Asynchronous programming involves programming models that can leverage multiple threads.



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

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

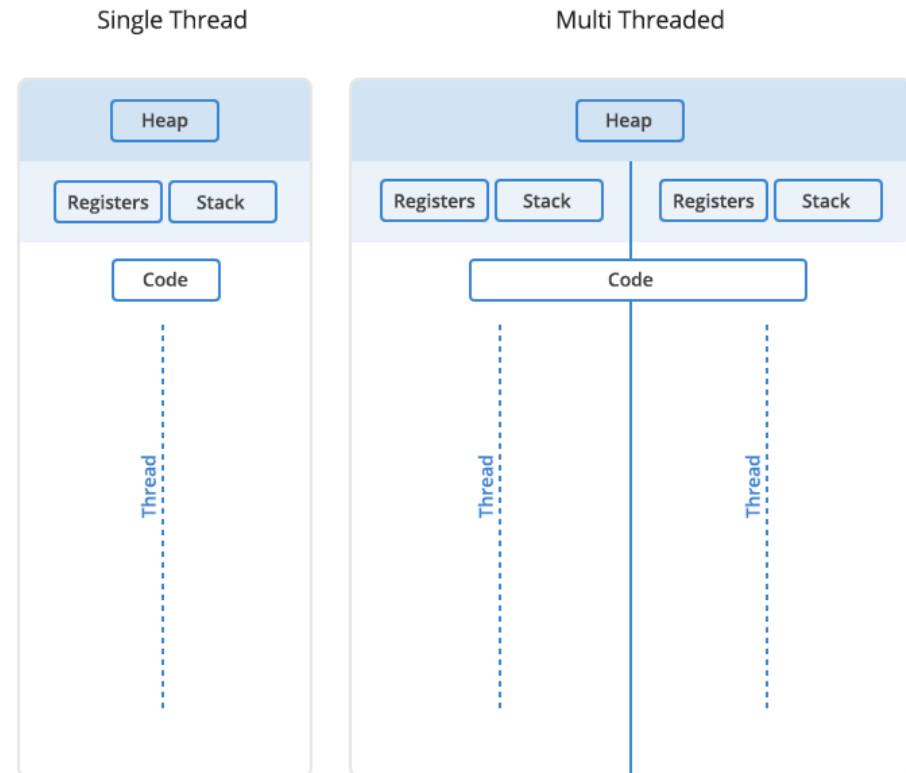
Strategies

How do we accomplish this?

Multi-threading

Multi-threading is the idea of manually splitting up computation across threads. These would result in one **primary thread**, and one or more **background threads** all running at the same time.

- This can help solve the blocking problem, since one thread can wait for the blocking operation to complete, while the other threads continue processing.
- This also has the potential to increase performance, if we can split up work and have it done in parallel.



Threads share a heap. However, you want to take care to ensure that 2 threads aren't competing for something in-memory!

Managing Threads

Kotlin has native support for creating and managing threads. This is done by

- Creating a user thread (distinct from the main thread where you application code typically runs).
- Defining some task for it to perform.
- Starting the thread.
- Cleaning up when it completes.

In this way, threads can be used to provide asynchronous execution, typically running in parallel with the main thread (i.e. running "in the background" of the program).

Managing a thread

```
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
```

Alternate Syntax

```
fun thread(  
    start: Boolean = true,  
    isDaemon: Boolean = false,  
    contextClassLoader: ClassLoader? = null,  
    name: String? = null,  
    priority: Int = -1,  
    block: () -> Unit  
): Thread  
  
// a thread can be instantiated quite simply  
// pass in the arguments where you want a non-default value  
thread(start = true) {  
    // the thread will end when this block completes  
    println("${Thread.currentThread()} has run.")  
}
```

Threads: Pros/Cons

Background threads can solve the two issues that we identified above.

- a worker thread could wait for background computation to complete, while the main thread continues to process user commands or interaction.
- a worker thread can allow us to manage work in parallel or concurrently.

However, manually managing threads is challenging:

- It requires us to divide work and control the thread. It's easy to encounter **race conditions**.
- 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 (**contention**).
- User threads may not always be available, or may not be available in the number that we require.

Callbacks

Another solution is to use a callback function. Essentially, you provide the long-running function with a reference to a function and let it run on a thread in the background. When it completes, it calls the callback function.

```
fun postItem(item: Item) {
    preparePostAsync { token ->
        submitPostAsync(token, item) { post ->
            processPost(post)
        }
    }
}
```

```
fun preparePostAsync(callback: (Token) -> Unit) {
    // make request and return immediately, arrange callback to be invoked later
}
```

This is still not an ideal solution, and can be difficult to implement:

- Nested callbacks. Usually a function that is used as a callback, often needs its own callback.... complex.
- Error handling. The nesting model makes error handling and propagation of these more complicated.

Promises

Instead of blocking, a function can return a Promise - an object that we can reference immediately but which will be processed at a later time.

```
fun postItem(item: Item) {
    preparePostAsync()
        .thenCompose { token -> submitPostAsync(token, item) }
        .thenAccept { post -> processPost(post)}
}

fun preparePostAsync(): Promise<Token> {
    // make request and return a promise that is completed later
    return promise
}
```

Challenges

- Different programming model. The model moves from top-down imperative to compositional with chained calls.
- Different APIs. Need to learn a new API such as `thenCompose` or `thenAccept`, which can vary across platforms.
- Specific return type. Return type is a `Promise` which has to be introspected to determine the actual data.
- Error handling can be complicated. The propagation and chaining of errors aren't always straightforward.

Coroutines

The idiomatic “Kotlin Way” to handle this.

Coroutines

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 in time, and resume later on.

Coroutines can be thought of as light-weight threads, in the sense that they run a block of code in parallel with the rest of the code. However, they differ in some important ways:

- **A coroutine is not tied to any particular thread.** It may suspend its execution in one thread and resume in another one, taken from a pool of threads. This makes allocating and managing coroutines much more performant than managing threads.
- **A coroutine may remember state between calls.** This means that we can use coroutines to have cooperating functions, where control is passed back and forth between them.

Example: Lightweight Coroutines

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    repeat(100_000) { // launch a lot of coroutines
        launch {
            delay(5000L)
            print(".")
        }
    }
}

// output
..... (repeat 100,000 times)
```

Advantages of Coroutines

- The function signature remains exactly the same. The only difference is `suspend` being added to it. The return type however is the type we want to be returned.
- The code is still written as if we were writing synchronous code, top-down, without the need of any special syntax, beyond the use of a function called `launch` to kick off the coroutine.
- The programming model and APIs remain the same. We can continue to use loops, exception handling, etc. and there's no need to learn a complete set of new APIs.
- It is platform independent. Whether we're targeting JVM, JavaScript or any other platform, the code we write is the same. The compiler takes care of adapting it to each platform!

Setup

Kotlin provides the `kotlinx.coroutines` library with a number of high-level coroutine-enabled primitives. You will need to add the dependency to your `build.gradle` file, and then import the library.

```
// build.gradle
implementation 'org.jetbrains.kotlin:kotlinx-coroutines-core:1.6.0'

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

Your first coroutine

```
fun main() = runBlocking { // this: CoroutineScope
    launch {                // launch a new coroutine and continue
        delay(1000L)       // non-blocking delay for 1 second
        println("World!") // print after delay
    }
    println("Hello")       // main coroutine continues while previous delays
}
```

```
//output
```

```
Hello
```

```
World
```

> The section of code within the `launch {}` scope is delayed for 1 second. The program runs through the last line, prints "Hello" and then prints "World" after the delay.

How does it work?

The section of code within the `launch {}` scope is delayed for 1 second. The program runs through the last line, prints "Hello" and then prints "World" after the delay.

- [runBlocking](#) is a coroutine builder that bridges the non-coroutine world of a regular `fun main()` and the code with coroutines inside of the `runBlocking { ... }` curly braces. This is highlighted in an IDE by this: `CoroutineScope` hint right after the `runBlocking` opening curly brace.
- [launch](#) is also a *coroutine builder*. It launches a new coroutine concurrently with the rest of the code, which continues to work independently. That's why `Hello` has been printed first.
- [delay](#) is a special *suspending function*. It *suspends* the coroutine for a specific time. Suspending a coroutine does not *block* the underlying thread, but allows other coroutines to run and use the underlying thread for their code.

Suspending Functions

We can extract the block of code inside `launch {}` into a *suspending function*.

Suspending functions can be used inside coroutines just like regular functions, but their additional feature is that they can use other suspending functions (like `delay`) to *suspend* execution of a coroutine.

```
fun main() = runBlocking { // this: CoroutineScope
    launch { doWorld() } // call a suspending function using launch
    println("Hello")
}

// this is your first suspending function, indicated with suspend keyword
suspend fun doWorld() {
    delay(1000L) // delay is also a suspending function
    println("World!")
}

// output
Hello // same behaviour as before
World!
```

Structured Concurrency

Coroutines follow a principle of **structured concurrency** which means that new coroutines can be only launched in a specific [CoroutineScope](#) which delimits the lifetime of the coroutine.

The above example shows that [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 they are not lost and do not leak. An outer scope cannot complete until all its children coroutines complete.

Coroutine Builders: Launch

A **coroutine builder** is a function that *creates a new coroutine*. Most coroutine builders also start the coroutine immediately. The most common coroutine builder is `launch`, which takes a lambda argument representing the function to execute.

```
val ENDPOINT = "http://kotlin-book.bignerdranch.com/2e/flight"
fun fetchData(): String = URL(ENDPOINT).readText()
```

```
@OptIn(DelicateCoroutinesApi::class)
fun main() {
    println("Started")
    GlobalScope.launch {
        val data = fetchData()
        println(data)
    }
    println("Finished")
}
```

```
// output
Started
Finished
```

When we run this program, it completes immediately. After the `fetchData()` function is called, the program continues executing and completes.

This is the due to the `launch` builder.

This is a case where running the entire program asynchronously isn't *really* what we want.

Coroutine Builders: runBlocking

```
val ENDPOINT = "http://kotlin-book.bignerdranch.com/2e/flight"  
fun fetchData(): String = URL(ENDPOINT).readText()
```

```
@OptIn(DelicateCoroutinesApi::class)  
fun main() {  
    runBlocking {  
        println("Started")  
        launch {  
            val data = fetchData()  
            println(data)  
        }  
        println("Finished")  
    }  
}
```

```
// output  
Started  
Finished  
ZI9135,ALQ,FPC,Delayed,30
```

Using the `runBlocking` builder, `launch` is called asynchronously, and then execution continues to the end of the scope before pausing.

The ordering of data demonstrates how the data from `fetchData()` is returned *after* “Finished” is displayed.

Coroutine Builders: Scope Builder

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.

Coroutine Builders: Scope Builder

```
// Sequentially 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
```

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

```

import kotlinx.coroutines.*

suspend fun main() {
    val start = System.currentTimeMillis()
    coroutineScope {
        for (i in 1..10) {
            launch {
                delay(3000L - i * 300)
                log(start, "Countdown: $i")
            }
        }
        log(start, "Liftoff!")
    }
}

fun log(start: Long, msg: String) {
    println("$msg " + "(on ${Thread.currentThread().name}) " + "after ${((System.currentTimeMillis() - start)/1000F)}s")
}

```

```

> Countdown: 10 (on DefaultDispatcher-worker-1 @coroutine#10) after 0.216s
> Countdown: 9 (on DefaultDispatcher-worker-1 @coroutine#9) after 0.516s
> Countdown: 8 (on DefaultDispatcher-worker-1 @coroutine#8) after 0.814s
> Countdown: 7 (on DefaultDispatcher-worker-1 @coroutine#7) after 1.114s
> Countdown: 6 (on DefaultDispatcher-worker-1 @coroutine#6) after 1.414s
> Countdown: 5 (on DefaultDispatcher-worker-1 @coroutine#5) after 1.616s
> Countdown: 4 (on DefaultDispatcher-worker-1 @coroutine#4) after 1.821s
> Countdown: 3 (on DefaultDispatcher-worker-1 @coroutine#3) after 2.143s
> Countdown: 2 (on DefaultDispatcher-worker-1 @coroutine#2) after 2.365s
> Countdown: 1 (on DefaultDispatcher-worker-1 @coroutine#1) after 2.659s
> Liftoff!

```

<https://kotlinlang.org>

public repo: coroutines/countdown

Managing Coroutines

Jobs

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 to its Job
    delay(1000L)
    println("World!")
}
println("Hello")
job.join() // wait until child coroutine completes
println("Done")
```

public repo: [coroutines/job](#)

Cancellation

The [launch](#) function returns a [Job](#) that can be used to cancel the running coroutine:

```
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.
```

public repo: [coroutines/job](#)

Composing Suspending Functions

Sequential (Default)

What do we do if we need 2 functions to be called sequentially? e.g. we want to execute `doSomethingUsefulOne` *and then* `doSomethingUsefulTwo`, and compute the sum of their results.

- Just use a normal sequential invocation, because the code in the coroutine, just like in the regular code, is *sequential* by default.
- In other words, we can just call the functions outside of a coroutine scope and they will execute like regular functions.

Sequential Example

```
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 = doSomethingUsefulOne()
    val two = doSomethingUsefulTwo()
    println("The answer is ${one + two}")
}
println("Completed in $time ms")

// output
The answer is 42
Completed in 2017 ms
```

Async for Promises

What if there are no dependencies between invocations of `doSomethingUsefulOne` and `doSomethingUsefulTwo` and we want to get the answer faster, by doing both *concurrently*?

- Use **async**, another builder.

Conceptually, [async](#) is just like [launch](#). It starts a separate coroutine which is a lightweight 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 Example

```
val time = measureTimeMillis {  
    val one = async { doSomethingUsefulOne() }  
    val two = async { doSomethingUsefulTwo() }  
    println("The answer is ${one.await() + two.await()}")  
}  
println("Completed in $time ms")
```

// output

The answer is 42

Completed in 1017 ms

Resources

- Andrew Bailey, David Greenhalgh & Josh Skeen. 2021. **Kotlin Programming: The Big Nerd Ranch Guide**. 2nd Edition. Pearson. ISBN 978-0136891055.
- Korhan Bircan. 2017. **Multithreading and Kotlin**. <https://medium.com/@korhanbircan/multithreading-and-kotlin-ac28eed57fea>
- Roman Elizarov. 2017. **Introduction to Coroutines**. KotlinConf. https://www.youtube.com/watch?v=_hfBv0a09Jc
- Google. 2022. **Kotlin Coroutines on Android**. <https://developer.android.com/kotlin/coroutines>
- Ryan Harrison. **Make HTTP Requests in Kotlin**. <https://ryanharrison.co.uk/2018/06/15/make-http-requests-kotlin.html>
- Soufiane Sakhi. 2019. **Kotlin Coroutines: An Introduction**. <https://simply-how.com/kotlin-coroutines-introduction>
- Upsana. 2022. **Using Java 11 HttpClient with Kotlin Coroutines**. <https://www.javacodemonk.com/using-java-11-httpclient-with-kotlin-coroutines-f0ca9111>