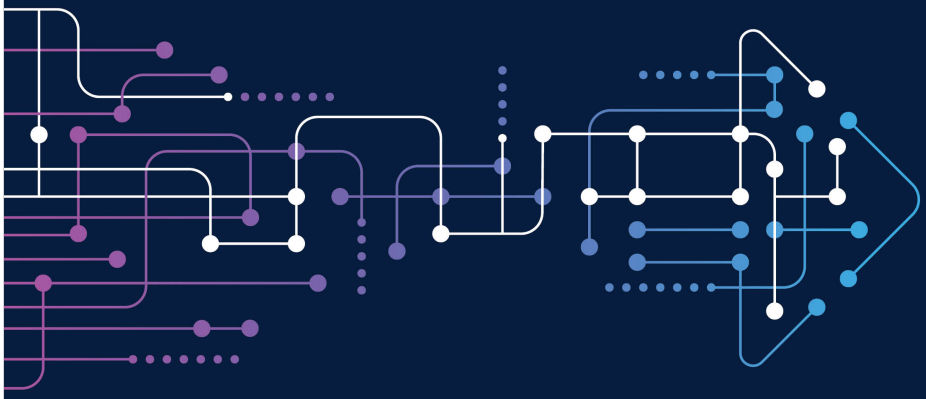
The Pragmatic Programmers

Kotlin Coroutine Confidence

Untangle Your Async, Ship Safety at Speed



Sam Cooper
edited by Michael Swaine

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Get Ready to Party

Imagine you're preparing for a party. First, you'd like to clean and dry your favorite outfit. Next, you need to bake and decorate the cake.

We can write out the tasks as a simple Kotlin program, with a named function for each task.

party/v1/src/main/kotlin/com/example/party/PartyPlanningApplication.kt

```
suspend fun doTasks() {
  doLaundry()
  bakeCake()
}

suspend fun main() {
  doTasks()
  println("All done!")
}
```

Both tasks are a good fit for asynchronous suspending functions, because they each involve some waiting. The clothes will sit in the washer without any intervention from you, and the cake will need some time to bake in the hot oven.

Let's implement the functions for the two tasks, so we can run the program.

We'll use the suspending delay() function to simulate the time you'd spend waiting for the real task to complete. To show what each task is doing, we'll also include some println() calls.

party/v1/src/main/kotlin/com/example/party/PartyPlanningApplication.kt

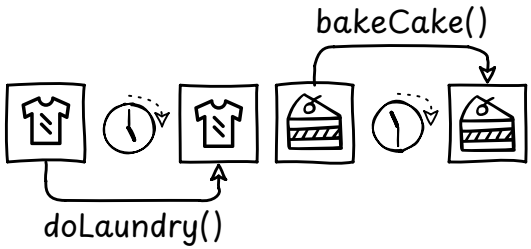
```
suspend fun doLaundry() {
  println("[Wash1] Putting the clothes in the washer")
  delay(1.seconds)
  println("[Wash2] Moving the clothes to the dryer")
  delay(1.seconds)
  println("[Wash3] Laundry's ready")
}

suspend fun bakeCake() {
  println("[Cake1] Putting the cake in the oven")
  delay(1.seconds)
  println("[Cake2] Waiting for the cake to cool")
  delay(1.seconds)
  println("[Cake3] Decorating the cake")
}
```

When you run the program, notice how it works through the instructions one by one, completing each function before moving on to the next. It won't start

work on the cake until the laundry is clean and dry. You can read through the output in exactly the same order as you read through the program's code.

```
[Wash1] Putting the clothes in the washer
[Wash2] Moving the clothes to the dryer
[Wash3] Laundry's ready
[Cake1] Putting the cake in the oven
[Cake2] Waiting for the cake to cool
[Cake3] Decorating the cake
All done!
```

As you learned in Chapter 2, Escape From Callback Hell, on page ?, a suspending function executes its contents in order, one line at a time, just like an ordinary function.



Unlike the callbacks they replace, suspending functions don't create their own background tasks. But is that really what we want? If you were actually doing these jobs around the house, you wouldn't separate them out sequentially. Instead, you'd multitask, working on both jobs concurrently.

Once the clothes are in the washer, there's no need to sit and wait for them. You can go and make a start on the cake recipe while the washing machine carries on doing its thing. Later, while you wait for the cake to cool, you can come back and move the clothes to the dryer.

By switching between the tasks, you'll finish the work quicker, without getting help from anyone else or working any harder.

Start Your Coroutines!

A suspending function frees up its thread while it's waiting—but that's only useful if there are other things for that thread to do during the gaps. This program just has one task, with its own dedicated thread, so our code doesn't yet have any opportunities for multitasking.

Let's fix that, and let our program know that it doesn't need to wait for the first task to complete before starting work on the second. All we need to do is launch() each function in its own coroutine.

To make new coroutines, we'll need a coroutine scope, just as we did in the last chapter. But this time, we're doing everything from within a suspending function that's perfectly capable of waiting for things. That gives us a whole new way to manage our background tasks. If we just use a suspension point to wait for all our coroutines to finish, none of them will ever outlive the function that created them, and there won't be any risk of task leaks. Clever, right?

That's exactly the job of the coroutineScope() function, which is part of the kotlinx.coroutines core library.

Be careful, though—there's also a capitalized CoroutineScope() constructor function. That's a more customizable version of the MainScope() function we've used before, and won't help us here.

Unlike MainScope() and CoroutineScope(), the lowercase coroutineScope() function doesn't create and return a CoroutineScope object. Instead, it takes your code as an argument, including any coroutines you want to launch, and runs everything in its own in-house coroutine scope. It's also a suspending function, and it'll wait for all the code and tasks inside its scope to finish before it returns.

Let's put it to use in our doTasks() function, and break our code into separate coroutines.

party/v2/src/main/kotlin/com/example/party/PartyPlanningApplication.kt

```
suspend fun doTasks() = coroutineScope {
  launch { doLaundry() }
  launch { bakeCake() }
}

suspend fun main() {
  doTasks()
  println("All done!")
}
```

Ready to Receive

The coroutineScope() function's code block can access its coroutine scope as an implicit receiver. In the last chapter, we had to write out coroutineScope.launch() with its explicit receiver in full. This time, we're free to just write launch() on its own. Nice!

You'll notice two things that are different when we run the program this time.

First, the lines of output are in a different order.

```
[Wash1] Putting the clothes in the washer
[Cake1] Putting the cake in the oven
```

```
[Wash2] Moving the clothes to the dryer
[Cake2] Waiting for the cake to cool
[Wash3] Laundry's ready
[Cake3] Decorating the cake
All done!
```

After the first task starts, the second task kicks off right away. Within each individual task, the normal rules of ordering still apply—so you'll never see the cake being decorated before it's baked, for example. But the program's free to switch back and forth between the two tasks, just like we wanted.

Out of Order

Whenever a program's output appears in a different order from the lines of code in the actual program, it's a clue that there's some concurrency going on.

Second, the program finished faster! We included two seconds of simulated delay in each task, so running them sequentially would have taken a total of four seconds. But this time, the program finished in closer to two seconds.

Fill in The Blanks

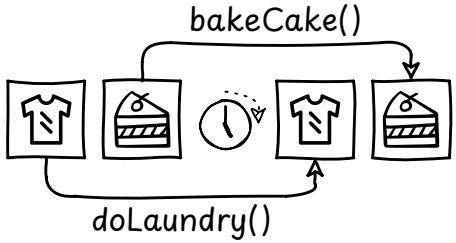
How did adding more code make our program go faster?

You learned right at the start of our journey that suspending functions are all about waiting. Each suspension point, whether it's a timed `delay()`, a network call, or some other outside event, leaves a gap where its thread is no longer busy running code. Using those gaps to make progress on other tasks is a big part of how coroutines help make our programs more efficient.

That means coroutines aren't just for user-interface code. As we'll learn in Chapter 7, Unlock Parallel Processing, on page ?, new threads don't come for free. The more we can make use of the threads we already have, the more we can save resources and improve performance, no matter what kind of work we're doing.

To provide extra things to keep our thread busy, all we need to do is launch more coroutines. Then, when one coroutine is held up, the program can simply pick one of the others to make progress on instead.

In our upgraded `doTasks()` function, we're creating two coroutines—one for each of our two tasks. When the `doLaundry()` task reaches its first call to `delay()`, that coroutine is going to be suspended for a while. It's during that waiting time that the program can switch its attention and make a start on the `bakeCake()` function in coroutine number two.



Each task still took the same amount of time, but we took advantage of the gaps in each task to make progress on the other at the same time.

This kind of concurrency is also called *cooperative multitasking*, because the tasks are cooperating with each other by giving up their thread at each suspension point. Since we're just making more efficient use of the resources we already have, cooperative multitasking lets us complete asynchronous tasks faster without any need for extra threads, processor cores or other resources. Who says there's no such thing as a free lunch?