



★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



Seven demos to understand coroutines: scope, context and Jobs



Tom Colvin · [Follow](#)

Published in ProAndroidDev · 6 min read · 2 days ago



68



Photo by [Antony Trivet](#) (Unsplash)

Understanding coroutines — *really* understanding them, not just learning patterns — comes from seeing what goes on under the hood.

My last post pulled apart common coroutine patterns in Android to see why they work. The more we dug, the more we saw there were three concepts woven like threads through everything: context, scopes and Job.

So it seems like those might be the gateway to understanding what coroutines *are* — and therefore key to using them properly.

With that in mind, let's have some fun pulling at those threads*...

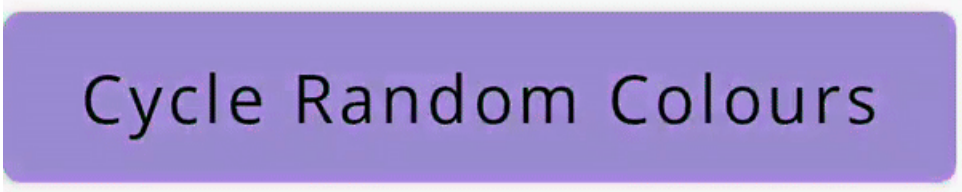
(* pun slightly intended)

Demo 1: Firing and forgetting coroutines

A coroutine *scope* is a box into which you place coroutines. It exists to restrict them.

You can see the value of that when you want to cancel coroutines — just destroy the box, and all the coroutines inside it get cancelled. This is brilliant, because it means you don't have to keep track of coroutines you create. Their lifecycle is managed by the “box”.

To give a concrete example, here's a button which cycles its background colour when clicked:

A rectangular button with rounded corners, filled with a solid purple color. The text "Cycle Random Colours" is centered on the button in a black, sans-serif font.

```

1  @Composable
2  fun RandomColourButton() {
3      val scope = rememberCoroutineScope()
4      var buttonColor by remember { mutableStateOf(Color.Red) }
5
6      Column {
7          Button(colors = ButtonDefaults.buttonColors(backgroundColor = buttonColor),
8              onClick = {
9                  scope.launch {
10                     while(true) {
11                         delay(500)
12                         buttonColor = Color(Random.nextInt(0xFF), Random.nextInt(0xFF),
13                     }
14                 }
15             }
16         ) {
17             Text("Cycle Random Colours")
18         }
19     }
20 }

```

MainScreen.kt hosted with ❤ by GitHub

[view raw](#)

We saw in my [last post](#) why it's safe to have the `while(true)` there, which would in any other context be horrific. But we skipped over something even more exciting!

...Which is this: See how we simply launched a coroutine on line 9, and forgot about it? We can do so perfectly safely because we launched it into a **scope**, in this case one that's obtained by `rememberCoroutineScope()`. That scope gets cancelled when the `RandomColourButton` is taken off the screen, and so our launched coroutine is also cancelled.

To any developer coming from an environment which only has *threads*, not coroutines, it's hard to overstate what a massive leap forward that is. It would be unthinkable to launch a thread and forget about it. You have to meticulously account for them all, on pain of weird bugs and resource leaks.

Demo 2: Cancel a launch()ed coroutine

When you launch a coroutine in a scope, as we did with `scope.launch {...}` in the code above, you get a `Job`. This `Job` represents the running coroutine.

You can cancel the coroutine (without affecting the parent scope, or any other coroutines in it) using `Job.cancel()`:

```

1  val job = scope.launch {
2      while(true) {
3          delay(500)
4          buttonColor = Color(Random.nextInt(0xFF), Random.nextInt(0xFF), Random.nextInt(0
5      }
6  }
7
8  scope.launch {
9      delay(2000)
10     job.cancel()
11 }

```

MainActivity.kt hosted with ❤ by GitHub

[view raw](#)

...and with the code above, the flashing button stops flashing after 2 seconds. We have saved the first coroutine's Job and we launch a second coroutine to cancel it after a delay.

Demo 3: Launching a coroutine inside a coroutine

A Job can also have children. An easy way to create a child Job is to simply use `launch {...}` within a coroutine.

When a Job is cancelled, all its children are cancelled too. Here's an example of that using our flashing button.

This time we've added another coroutine which increments a count:



Cycle Random Colours (count = 1)

The coroutine which counts up is launched as a child of the coroutine which picks the random colour:

```

1  @Composable
2  fun FlashingCountingButton() {
3      val scope = rememberCoroutineScope()
4      var buttonColor by remember { mutableStateOf(Color.Red) }
5      var count by remember { mutableStateOf(1) }
6
7      Column {
8          Button(colors = ButtonDefaults.buttonColors(backgroundColor = buttonColor),
9              onClick = {
10                 val job = scope.launch {
11                     // launch a separate coroutine, inside this one, to increase the cou
12                     launch {
13                         while(true) {
14                             delay(500)
15                             count ++
16                         }
17                     }
18
19                     //...and here we change the background colour
20                     while(true) {
21                         delay(500)
22                         buttonColor = Color(Random.nextInt(0xFF), Random.nextInt(0xFF),
23                     }
24                 }
25
26                 // cancel the above coroutine after 5 seconds
27                 scope.launch {
28                     delay(5_000)
29                     job.cancel()
30                 }
31             }
32         ) {
33             Text("Cycle Random Colours (count = $count)")
34         }
35     }
36 }

```

MainActivity.kt hosted with ❤ by GitHub

[view raw](#)

Notice how the count stops when the flashing stops. That's because the coroutine launched on line 12 is a child of the coroutine launched on line 10. When the parent is cancelled after a delay, it cancels the child, too.

In the world of coroutines, the laws governing this parent-child relationship (those same laws which ensure you can't lose track of coroutines), are called **structured concurrency**.

Demo 4: Launching a coroutine in a context, specifying a job

There are other ways to launch a coroutine into a Job. You can specify the Job in a call to launch or async:

```

launch(myJob) {
    ...
}

```

```
}
```

Obviously, if you do this it's up to you to carefully keep track of myJob, cancelling it when needed, so that this coroutine's lifecycle is properly bounded.

The reason this works is because the argument to `launch` is a `CoroutineContext`, of which the job is an element.

A **coroutine context** is just a collection of metadata about a coroutine. This includes its Job, name and dispatcher. See it as luggage tags attached to the coroutine, which explain to the coroutines library how to run it.

Demo 5: Launching a coroutine onto a different thread pool

One of those “luggage tags” — that is, one item of metadata attached to a coroutine — is the dispatcher. The coroutines library uses this to determine which thread or thread pool to run the coroutine on.

So using the same mechanism as above, you could launch a coroutine on the IO dispatchers thread pool:

```
launch(Dispatchers.IO) {  
    ...  
}
```

This works by creating a coroutine context which sets the dispatcher to `Dispatchers.IO`.

Demo 6: Other coroutine launching options (and combinations)

Or, you can combine elements of a coroutine context using the plus operator. Here's how to launch a coroutine called “boo” into a new Job on the IO dispatchers thread pool:

```
launch(Dispatchers.IO + Job() + CoroutineName("boo")) {  
    ...  
}
```

When you create a coroutine context in this way, **any elements you don't mention will be inherited from the parent's context**. So if you have `launch(Job())` in a coroutine being run on `Dispatchers.IO`, the launched child will also run on `Dispatchers.IO`. This is because its context's dispatcher will have been inherited from its parent.

Demo 7: Using a coroutine scope — and what happens when you do

So we've seen:

- Demo 1: A coroutine *scope* is kind of a lifetime-restricting container we can put coroutines into
- Demo 3: A coroutine *Job* is ... kind of a lifetime-restricting container we can put coroutines into.

Hmm. So what's the difference?

As it turns out, there is no difference. Because:

A coroutine scope is just a wrapper for a coroutine context, which holds a `Job`.

...Which you can see from the coroutines library code:

```
public interface CoroutineScope {  
    public val coroutineContext: CoroutineContext  
}
```

Why, then, have separate types and a whole different name for essentially the same thing? The great Roman Elizarov, former project lead for Kotlin language, gives the analogy of a ship which has many different names for “rope”. Why? Because they are named by *usage*. A halyard is a rope to pull up the main sail, a downhaul is one to pull it down, a sheet is used to trim a sail.

In the analogy, a coroutine scope is so named because its intended use is as a *particular kind* of coroutine context. It's one used to limit and control coroutines' lifespans.

Under the hood, that means a coroutine scope will have a `Job` attached to a specific lifecycle:

- `viewModelScope` has a Job which is cancelled when the view model is cleared
- `rememberCoroutineScope()` has a Job which is cancelled when the composable exits the composition
- `viewLifecycleOwner.lifecycleScope` has a Job which is cancelled when the lifecycle of the Android View ends

...and so on.

That takes us all the way back to demo 1, in which we saw it was safe to fire and forget a coroutine as it's placed inside a scope. We can now see that's because the scope has a Job bound to a relevant lifecycle.

To summarize...

The concepts of coroutine context, scope and Job underpin a large part of we do with coroutines. We've seen how context is like a list of metadata associated with a coroutine which provides information to observers and tells the dispatcher how to run it. We've seen how a Job represents a running coroutine, and can act as a parent to other coroutines. And from that we can understand a coroutine scope, which is a context with a parent Job.

I hope this has been helpful. As always, post any questions below!

Tom Colvin has been architecting software for two decades and is particularly partial to working with Android. He's co-founder of Apptaura, the mobile app specialists, and available on a consultancy basis.

Android

Android App Development

Kotlin

Coroutine

