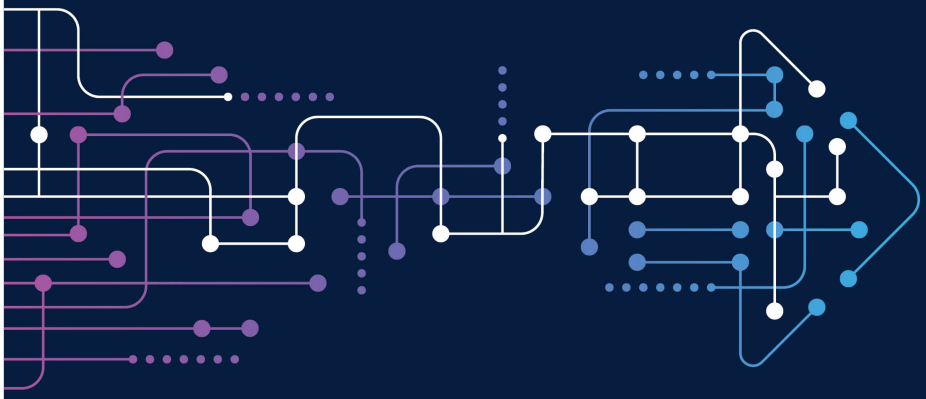
The Pragmatic Programmers

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.

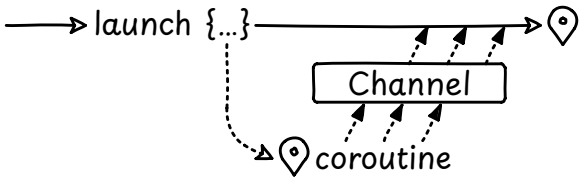
For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Create a Channel

Okay, let's rethink. We want our image producer to run in a new coroutine, but we need its output in the original coroutine where the flow is being collected.

If we were launching a coroutine to generate a single value, we'd solve that problem with the async() builder. It returns a Deferred result, letting the new coroutine pass a value back to its creator.

But an async() coroutine's await() function waits for the entire task to finish, before providing a single result. That's not what we need here. Our image producer is going to generate an ongoing stream of data, so we don't want to wait for its completion. Instead, we need a tool that will let us send a series of values back to the caller, while both tasks continue running.



What we're looking for is a Channel.

Let's make a simple program where we can see it in action. We'll need two coroutines—a sender and a receiver. To send our values, we'll create a new background coroutine with launch(). Meanwhile, our starting coroutine—the one that runs our suspending main() function—can play the role of the receiver.

channels/v1/src/main/kotlin/com/example/channels/ChannelsApplication.kt

```
suspend fun main() = coroutineScope {
  val channel = Channel<Char>()

  launch { // sender
    for (outgoingChar in "Hi!") {
      println("Sending '$outgoingChar'…")
      channel.send(outgoingChar)
      delay(100)
    }
  }

  repeat(3) { // receiver
    val incomingChar = channel.receive()
    println("Received '$incomingChar'")
  }
}
```

Communicate, Don't Mutate

When you need to share data between tasks, use a Channel instead of a shared variable or mutable data structure. Channels are safe for concurrent use by many different coroutines, and unlock many new ways to organize your code.

Learn to Communicate

The send() and receive() operations are both suspending functions, and the simple channel we've created here will act as a handoff point between our two coroutines. When the sender calls send(), it waits at the meeting point for the receiver to arrive—and when the receiver calls receive(), it does the same to wait for the sender. Once both coroutines have arrived, a value is passed from sender to receiver, and the program continues.

Did you run the code yet? The output looks like this.

```
Sending 'H'…
Received 'H'
Sending 'i'…
Received 'i'
Sending '!'…
Received '!'
```

It's easy to see how the values are being passed between the two coroutines, as each task continues with its own work.

So what have we gained that we couldn't do with a flow? Multitasking! Don't forget, a flow on its own is just a single task. With two communicating coroutines, we can now do two things at once.

If flows are like the multivalued equivalent of suspending functions, it can sometimes be helpful to think of a Channel as the multivalued equivalent of a Deferred. In fact, there's even a produce() coroutine builder that will create and return its own Channel, similar to how async() returns a Deferred. We could have used it in our program just now, but we'll leave it as-is for the time being—our goal is to understand the moving parts, not hide them.

	Code it	Launch it	Retrieve it
One value	suspend fun	scope.async()	Deferred
Many values	flow()	scope.produce(), or flow.produceIn(scope)	Channel

This comparison will help you choose the right tool, but it's not an exact parallel, and it won't hold up to scrutiny in every situation.

Close a Channel

Our simple channel arranged a face-to-face meeting between our coroutines, but that doesn't always have to be the case. A channel can also include its own buffer capacity, allowing it to accept items immediately from the sender, and hold onto them until the receiver arrives. We'll talk a bit more about this when we introduce the flow buffer() operator later on.

If the channel holds data, what happens to those waiting messages when the receiver stops receiving? For some programs, the answer is easy—nothing. When both the sender and receiver have gone away, the channel—and any data it might have been holding onto—can be garbage collected like any other object or collection. Channels themselves don't hold resources, and so they don't necessarily need to be manually cleaned up after use.

But even if the channel itself doesn't need cleaning up, the coroutine it's connected to might! If our sender coroutine fails or finishes, we don't want our receiver to continue waiting for messages that will never come. The receiver might even have its own cleanup code that it needs to run. That's why the sender has access to an additional close() function, which will let any receiving code know that it should stop waiting. Once a channel is closed, it can't be used again.

Just Looking Out For You

On its own, a channel doesn't need closing. The close() and cancel() functions are there to help you manage the other tasks and resources that are using the channel.

In our example, we knew how many items to expect, and we just called receive() the correct number of times before stopping. Let's change that, and have our sender use close() to signal the end of the stream.

channels/v2/src/main/kotlin/com/example/channels/ChannelsApplication.kt

```
suspend fun main() = coroutineScope {
  val channel = Channel<Char>()

  launch { // sender
    for (outgoingChar in "Hi!") {
      println("Sending '$outgoingChar'…")
      channel.send(outgoingChar)
      delay(100)
    }
➤   channel.close()
  }

➤ for (incomingChar in channel) { // receiver
    println("Received '$incomingChar'")
```

```
  }
  println("Done!")
}
```

Now that our stream of characters has an end, we can replace our three explicit receive() calls with a simple for loop. Channels have a special suspending iterator, so this loop will pause each time it needs to wait for an item. When the sender closes the channel, the loop will come to an end.

End with an Error

Like an async() coroutine's Deferred result, a Channel can deliver an error to its consumer. That's done by passing an exception to the close() function.

Trying to receive() more items from a failed channel will rethrow the original error. But just as you saw with async() and await(), this error handling mechanism will often be superseded by the rules of structured concurrency. Depending on the way you've grouped your coroutines, it's likely that an unhandled error in the producer task will cause the consumer to be cancelled before it ever sees that the channel has been closed.

Closed, Cancelled, or Crashed?

There's one final way a channel can be marked as inactive, and that's if the receiver coroutine calls cancel(). This function exists to let the sender know it should stop sending more values—and unlike close(), it will also discard any item's that haven't yet been delivered.

Close That Message

You can send any object you like via a Channel. So what if the messages themselves are closeable resources? The optional onUndeliveredElement handler lets you run cleanup code on in-flight messages when the receiver goes away. Check the docs for details.

That's three different ways to close a channel, so let's recap and compare.

- As the *sender*, call close() with no arguments to signal the end of the data. Calling send() after this is an error, but items that have already been sent will still make it to the receiver. Once everything's delivered, iterator-backed loops complete normally, while manual calls to receive() throw an error.
- As the *sender*, pass an exception to close() to signal that something went wrong while producing the data. This has just two key differences from a normal channel closure. First, iterator-backed receiver loops will end with an error, instead of terminating normally. Second, explicit calls to

send() and receive() will throw the exception you provided, instead of a generic ClosedSendChannelException.

- As the *receiver*, call cancel() to signal that you don't want to receive any more items. All in-flight items are discarded—there's nowhere for them to go. Both sender and receiver will get a CancellationException if they keep trying to use the channel.

Be Careful with Cancelled Channels

As you learned in Chapter 6, Cooperate with Cancellation, on page ?, sending cancellation signals between coroutines can cause some unexpected problems. Take care if you think you might encounter a cancelled channel in a non-cancelled coroutine.

If that sounds like a lot of errors, don't panic! There are alternative functions, such as trySend() and tryReceive(), for situations where closed channels are an expected part of your program's normal operation.