

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including our cookie policy. X

Kotlin: Diving in to Coroutines and Channels



Jag Saund

Jan 20 · 15 min read



If you're coming from Java, you probably associate asynchronous with threads. And that means you've dealt with shared mutable state. You've spent countless hours chasing down deadlocks and race conditions. You've taken care when modifying shared state using locking primitives like `synchronized`. You might have even used higher order locking functions like `Semaphores` and `Latches`.

At this point you can see that concurrency is hard. It's really hard. When you take a single threaded piece of code and make it concurrent, you inherently introduce a tremendous amount of complexity.

If you've been using Kotlin, you've probably heard of Coroutines. Coroutines aren't new. They've existed since the 60s/70s. They're part of a different concurrency model known as: "Communicating Sequential Processes" (CSP). Kotlin's concurrency model

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

X

also one of the more exciting parts of learning the language.

This article introduces you to the concept of Coroutines and Channels through illustrations.

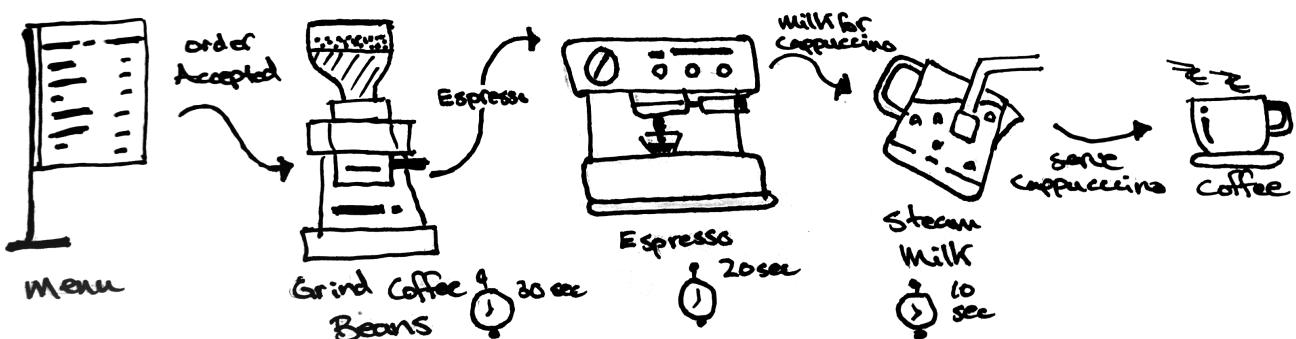
NOTE: At the time of this writing, Channels are in experimental stage.

Single Thread Coffee Shop

I'll use the analogy of ordering a Cappuccino at a coffee shop to explain Coroutines and Channels. Let's start with one Barista serving orders.

The Barista:

1. Takes an order
2. Grinds the coffee beans (30 seconds... it's a really slow coffee grinder)
3. Pulls a shot of espresso (20 seconds)
4. Steams the milk (10 seconds)
5. Combines the steamed milk with the shot of espresso (5 seconds... for some fancy latte art)
6. Serves the Cappuccino



This is like a single thread application — one Barista performing all the work sequentially. We can model each step as a function in a program. And because this is a single thread program, the current function has to complete before moving on to the next function. Here's what the program looks like (try it out):

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including
cookie policy.

X

```
private fun makeCoffee(orders: List<Menu>) {
    for (o in orders) {
        log("Processing order: $o")
        when (o) {
            is Menu.Cappuccino -> {
                val groundBeans = grindCoffeeBeans(o.beans())
                val espressoShot = pullEspressoShot(groundBeans)
                val steamedMilk = steamMilk(o.milk())
                val cappuccino = makeCappuccino(o, espressoShot, steamedMilk)
                log("Serve: $cappuccino")
            }
        }
    }
}
```

The output of this looks like:

```
[main] Processing order: cappuccino: beans=regular milk=whole milk price=$3.50
[main] Grinding beans
[main] Pulling espresso shot
[main] Steaming milk
[main] Combining ingredients
[main] Serve: Cappuccino(order=cappuccino: beans=regular milk=whole milk price=$3.50,
espressoShot=Espresso(beans=ground regular), steamedMilk=steamed milk)
```

Conceptually, this program is very simple. It's easy to reason about and understand. But it's not efficient. What happens when the Coffee Shop gets popular and we hire two more employees. How does our current conceptual model allow for three employees to operate together but independently. How do we scale our program. How can the program take advantage of multiple threads to make Cappuccinos. Concurrency becomes an important part of the solution.

Concurrent Coffee Shop

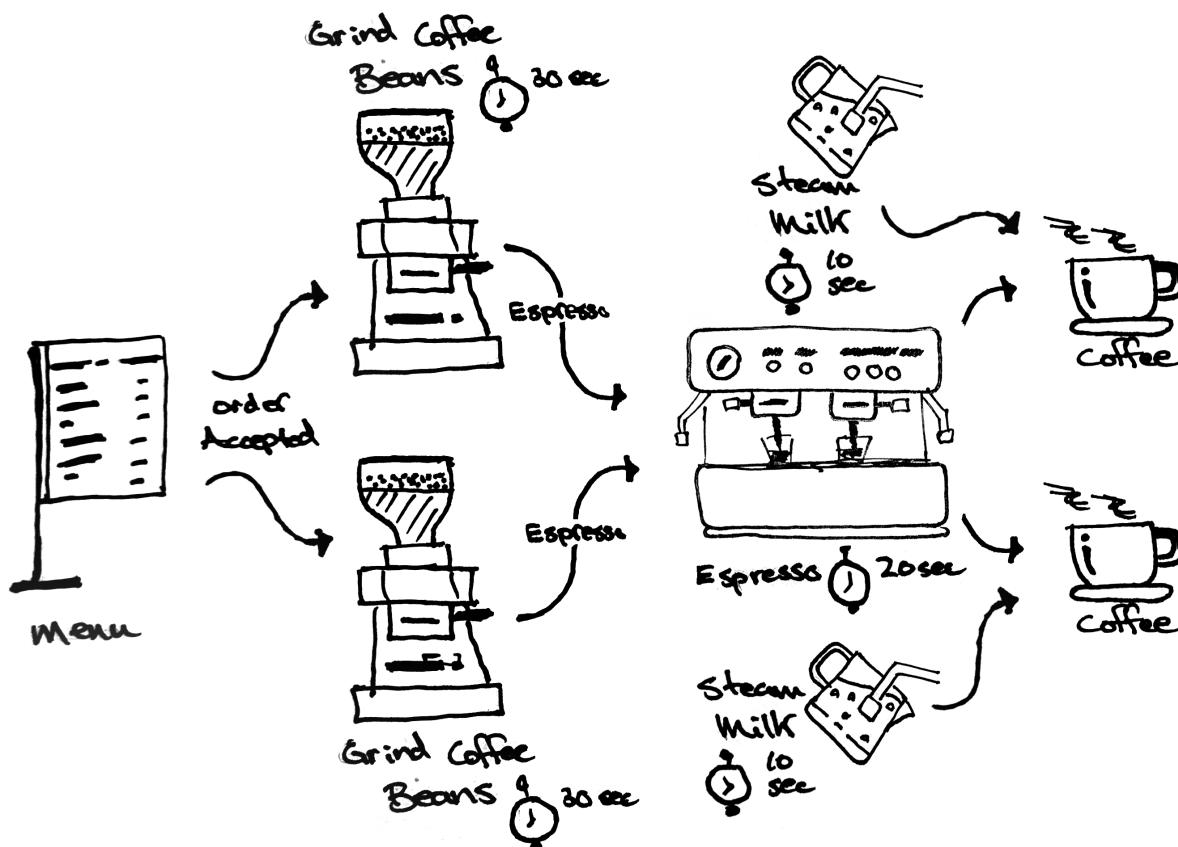
Let's assume, for our Coffee Shop analogy, we've hired another Barista and a cashier. And for now, let's assume that we have two Coffee Grinders and our Espresso Machine can pull two shots at once. We'll make one more optimization. The Barista can pull a

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including our cookie policy.

X

Here's what we have now:

1. The cashier takes a new order. The cashier waits until one of the Baristas starts processing that order before accepting new orders.
2. An available Barista will take the order and grind coffee beans (30 seconds)
3. Take the ground coffee beans to the espresso machine and pull an espresso shot (20 sec)
4. While the espresso is being made, the Barista steams the milk (10 sec)
5. Once the espresso shot and steamed milk are ready, the Barista will combine them to make a Cappuccino (5 seconds)
6. Serve the Cappuccino



Now we have a system that's much more efficient. There's multiple parts of the program that can run concurrently.

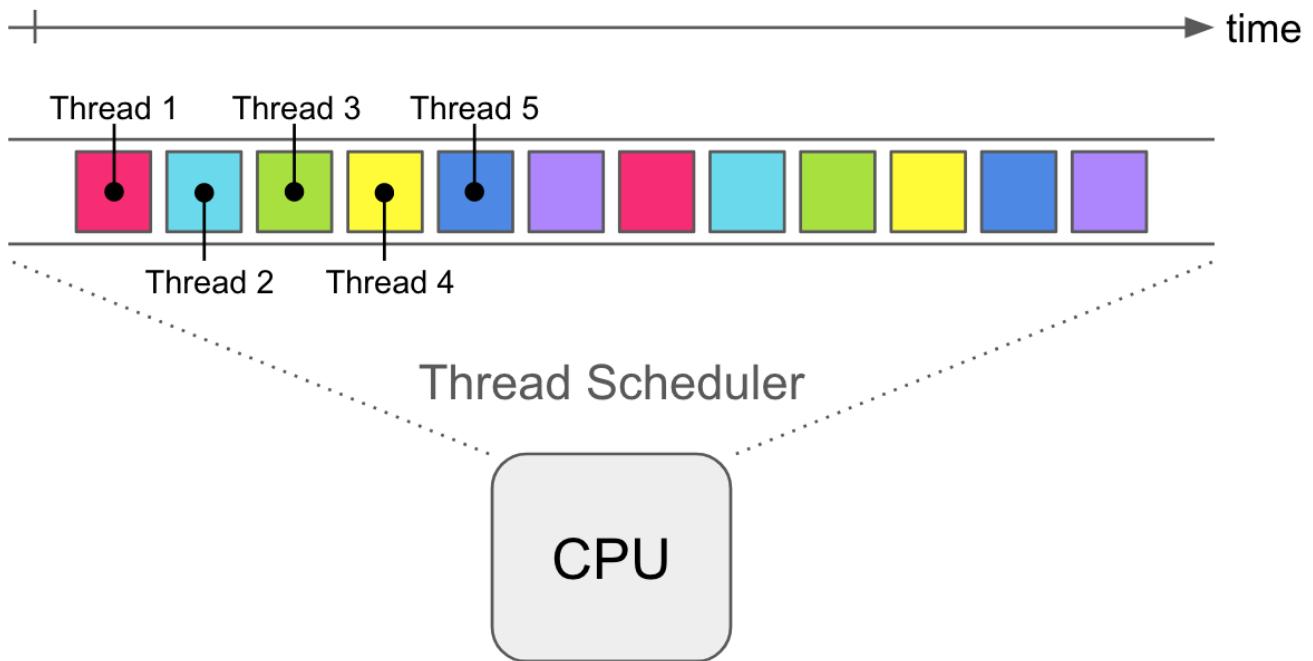
To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy. X

to the cashier. And we'll need a way for the Baristas to operate the Espresso Machine (think of the Espresso Machine as a shared resource) without conflicting with each other.

Threads and Coroutines

In our example above, we can represent the Baristas and the cashier as coroutines. But what exactly are coroutines. To answer that, let's first refresh what threads are.

Concurrency is not parallelism

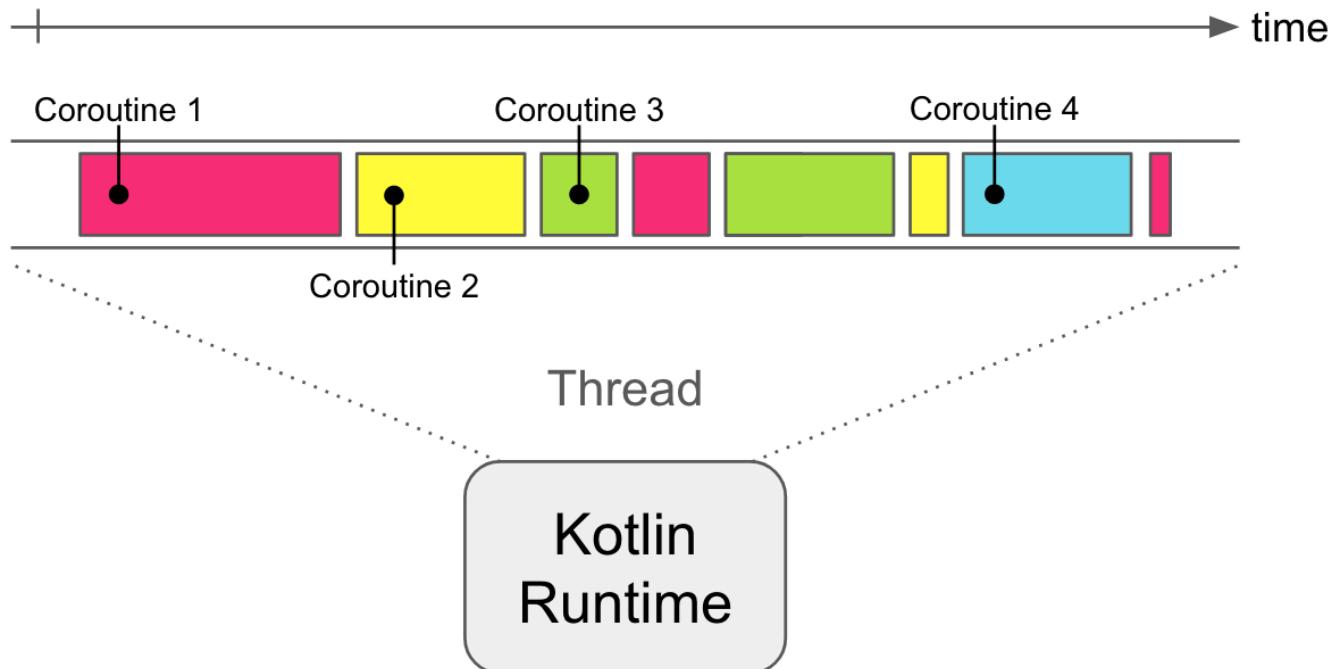


Threads allow units of work to execute concurrently. Each Java thread is allocated in user space but is mapped to a kernel thread. That means the operating system manages these threads. The operating system schedules a slice of time for each thread to run. If the thread isn't done executing in that window, the operating system interrupts the thread and switches to another thread. This is known as preemptive scheduling. On a single core processor, we can only have one thread running at a time. The fast preemptive scheduling of threads by the operating system is what allows for independent units of work to execute concurrently. Now, most modern phones have multi core CPUs. This allows for parallelism. Thread A can be executing on CPU core 1 while Thread B can be executing on CPU core 2. It's an important distinction to make — concurrency is not parallelism. The last thing about threads is that they're

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including our cookie policy.

X

and CPU cache invalidation.



Conceptually, coroutines are like threads. They execute units of work concurrently. But unlike threads, coroutines aren't necessarily bound to any particular thread. A coroutine can start executing in one thread, suspend execution, and resume on a different thread. Coroutines aren't managed by the operating system. They're managed at the user space level by the Kotlin Runtime. That means there is no time slice allocated to a coroutine to perform a unit of work. That also means there's no scheduler overhead. Instead, coroutines perform cooperative multitasking. When one coroutine hits a suspension point, the Kotlin Runtime will find another coroutine to resume. You can think of this like having multiple coroutines multiplexed on to a single thread. Coroutines have a small memory footprint — a few dozen bytes. That gives you a very high level of concurrency with very little overhead.

Let's update our first example so two coroutines process the list of orders concurrently (try it out).



```
fun main(args: Array<String>) = runBlocking {
    val orders = listOf(Menu.Cappuccino(CoffeeBean.Regular, Milk.Whole),
        Menu.Cappuccino(CoffeeBean.Premium, Milk.Breve),
        ...)
```

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

```
    menu.Cappuccino(CoffeeBean.Decaf, Milk.Normal))
    launch(CoroutineName("barista-1")) { makeCoffee(orders) }
    launch(CoroutineName("barista-2")) { makeCoffee(orders) }
}

private suspend fun makeCoffee(orders: List<Menu>) {
    for (o in orders) {
        log("Processing order: $o")
        when (o) {
            is Menu.Cappuccino -> {
                val groundBeans = grindCoffeeBeans(o.beans())
                val espressoShot = pullEspressoShot(groundBeans)
                val steamedMilk = steamMilk(o.milk())
                val cappuccino = makeCappuccino(o, espressoShot, steamedMilk)
                log("Serve: $cappuccino")
            }
        }
    }
}
```

Starting a new coroutine is as simple as calling `launch`. The above example starts two coroutines from `main`. Coroutines must be associated with a coroutine scope. This ensures coroutines are cleaned up without you having to explicitly manage the lifecycle of the coroutine. In our example, we start the coroutines inside the scope defined by `runBlocking`. This means, the `main` function won't terminate until the two child coroutines (`barista-1` and `barista-2`) have completed.

Tip: If you're trying this out on Android, please review Structured Concurrency and CoroutineScope.

The second thing we did is add the `suspend` modifier to the `makeCoffee` function. If you trace each method inside this function, you'll notice they also have the `suspend` modifier applied to their function declarations. And if you look at these functions, you'll notice they all call `delay` instead of `Thread.sleep`. The `delay` function places the coroutine in a suspended state for some period of time without blocking the thread it's running on. That means the Kotlin Runtime can find another coroutine to resume on this thread. This concept allows coroutines to use threads with a high degree of efficiency. Something that an operating system thread scheduler could never achieve.

Tip: Try removing the `suspend` keyword. You'll get a helpful error message 😊.

The output of this looks like:



To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including
cookie policy.

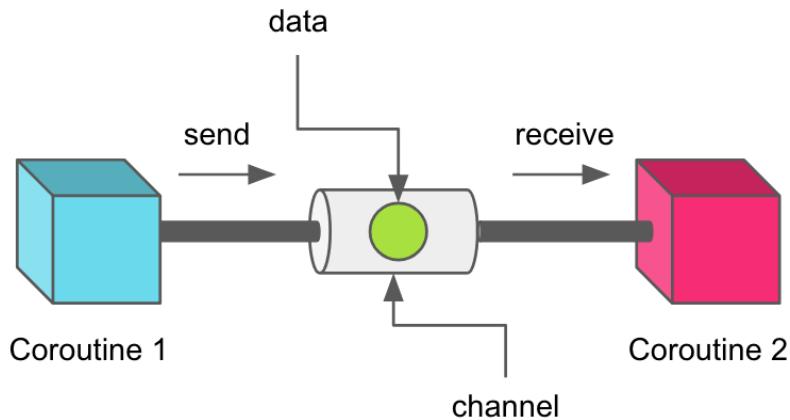
```
[main @barista-1#2] Processing order: cappuccino: beans=regular milk=whole milk price=$3.50
[main @barista-1#2] Grinding beans
[main @barista-2#3] Processing order: cappuccino: beans=regular milk=whole milk price=$3.50
[main @barista-2#3] Grinding beans
[main @barista-1#2] Pulling espresso shot
[main @barista-2#3] Pulling espresso shot
[main @barista-1#2] Steaming milk
[main @barista-2#3] Steaming milk
[main @barista-1#2] Combining ingredients
[main @barista-2#3] Combining ingredients
...
...
```

This is great! We now have two Baristas making coffee concurrently. And, they're both operating on the same thread! But how do we have the two Baristas talk to each other? This is where channels come in.

Tip: By specifying a dispatcher you can change the thread pool a coroutine is assigned to execute in:

```
launch(Dispatchers.Default + CoroutineName("barista-1")) {
    makeCoffee(ordersChannel)
}
```

Channels



You can think of a channel as a pipe between two coroutines. That pipe facilitates the transfer of information between those two coroutines. One coroutine can send

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including
cookie policy.

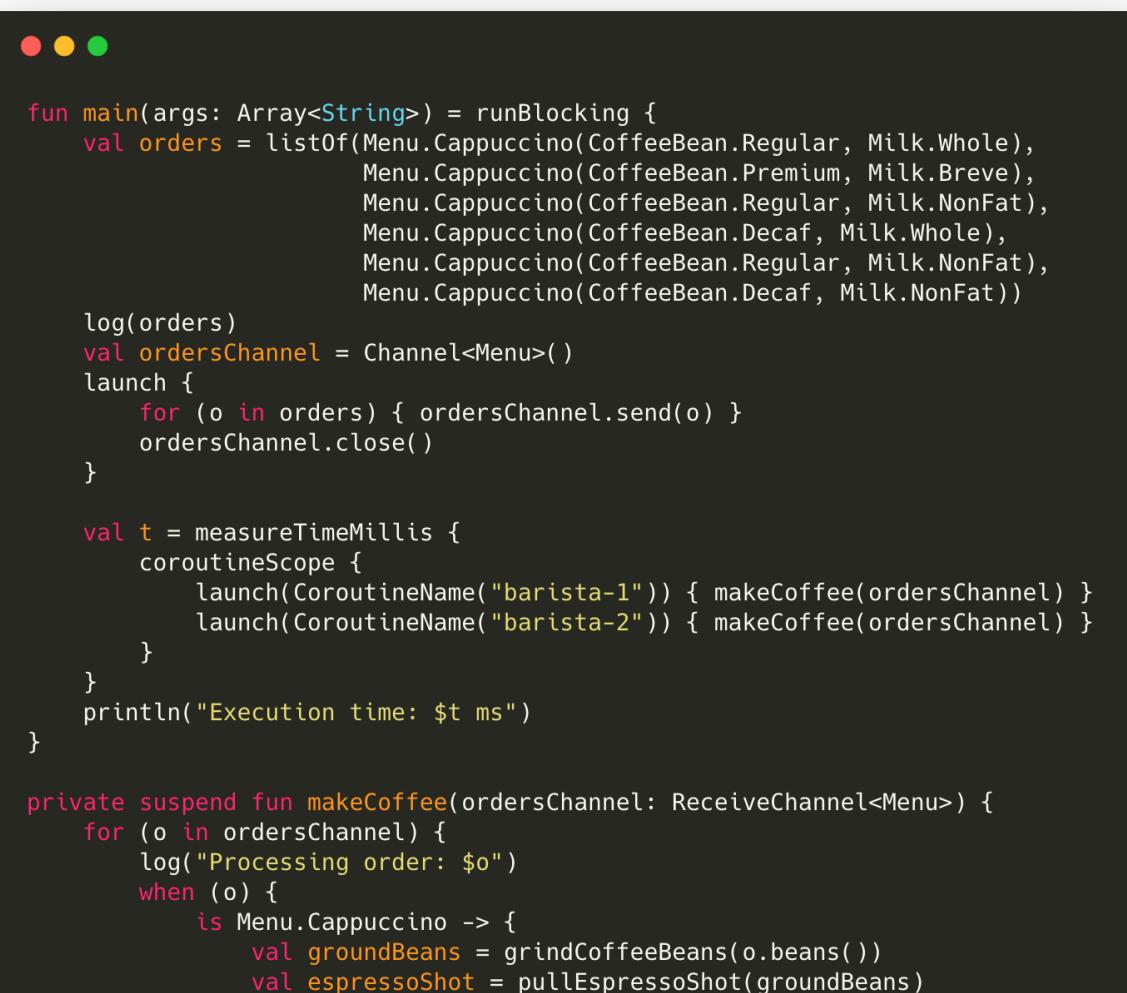
X

Do not communicate by sharing memory; instead, share memory by communicating.

When you're writing software that involves coordinating with blocking resources (network, database, cache, etc) or computationally intensive operations, you offload this to threads. So now you have things that are shared between threads. To share something safely, you rely on locking the resource or memory so that two threads can't read or write to it at the same time. This is typically how threads communicate — through shared memory. But it also causes all sorts of issues like race conditions, deadlocks, etc.

Channels promote a different perspective on communicating: **don't communicate by sharing memory, share by communicating.**

Let's update our example to use a channel to communicate processing orders between the Cashier and the two Baristas (try it out).



```
fun main(args: Array<String>) = runBlocking {
    val orders = listOf(Menu.Cappuccino(CoffeeBean.Regular, Milk.Whole),
        Menu.Cappuccino(CoffeeBean.Premium, Milk.Breve),
        Menu.Cappuccino(CoffeeBean.Regular, Milk.NonFat),
        Menu.Cappuccino(CoffeeBean.Decaf, Milk.Whole),
        Menu.Cappuccino(CoffeeBean.Regular, Milk.NonFat),
        Menu.Cappuccino(CoffeeBean.Decaf, Milk.NonFat))
    log(orders)
    val ordersChannel = Channel<Menu>()
    launch {
        for (o in orders) { ordersChannel.send(o) }
        ordersChannel.close()
    }

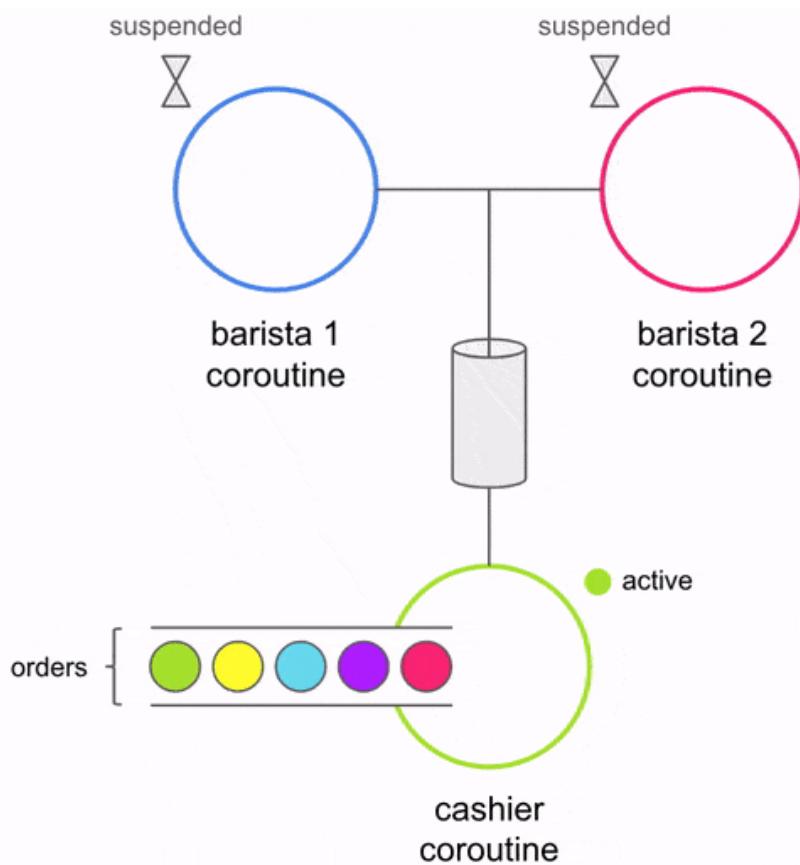
    val t = measureTimeMillis {
        coroutineScope {
            launch(CoroutineName("barista-1")) { makeCoffee(ordersChannel) }
            launch(CoroutineName("barista-2")) { makeCoffee(ordersChannel) }
        }
    }
    println("Execution time: $t ms")
}

private suspend fun makeCoffee(ordersChannel: ReceiveChannel<Menu>) {
    for (o in ordersChannel) {
        log("Processing order: $o")
        when (o) {
            is Menu.Cappuccino -> {
                val groundBeans = grindCoffeeBeans(o.beans())
                val espressoShot = pullEspressoShot(groundBeans)
                ...
            }
        }
    }
}
```

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including
cookie policy. X

```
    }  
}
```

Before we dive in to the code, let's take a moment to understand the changes conceptually. Below is a visualization of what the code above is doing.



There are three coroutines (Cashier, Barista 1, and Barista 2) operating independently and performing specific units of work. The Cashier communicates with the two Baristas via the channel. The Cashier accepts an order, places it on the channel, and waits for one of the two Baristas to accept the order. This is how coroutines synchronize with each other. Once the Barista finishes making coffee, it will sync up with the Cashier to process the next order. What if there aren't any orders yet? The two Baristas will suspend execution and wait until an order arrives on the channel.

The first thing we need to do is create the channel:

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including
cookie policy.

X

Next, we need to send orders on this channel. Sending on a channel is a suspendible operation and needs to be invoked from within a coroutine. We create a new Cashier coroutine for this purpose.

```
launch { // launches the cashier coroutine
    for (o in orders) { ordersChannel.send(o) }
    ordersChannel.close() // don't forget to close channels when
    you're done with them!
}
```

Now we have a way for the cashier to send orders to make coffee in a safe way. The next thing we need to do is update the logic for the two Baristas to consume from this channel.

The `makeCoffee` function now accepts a channel instead of a list. The function will iterate over the channel as it did with the list. When there is nothing available on the channel, the function suspends execution.

```
private suspend fun makeCoffee(ordersChannel: Channel<Menu>) {
    ...
}
```

Closing a channel is like a terminal event. This signals to the functions reading from that channel that there is nothing left to process. This terminates the loop inside `makeCoffee` and allows the coroutine to finish. What if we never closed the channel? That means both Barista coroutines would be in an indefinite suspended state (waiting for something to arrive on the channel). And because the two coroutines belong to the `runBlocking` scope, the main function would never complete.

Here's what the output looks like:

```
[main @barista-1#3] Processing order: cappuccino: beans=regular milk=whole milk price=$3.50
[main @barista-1#3] Grinding beans
[main @barista-2#4] Processing order: cappuccino: beans=premium milk=breve price=$5.50
```

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

```
[main @barista-1#3] Combining ingredients
[main @barista-2#4] Combining ingredients
[main @barista-1#3] Serve: Cappuccino(order=cappuccino: beans=regular milk=whole milk price=$3.50,
espressoShot=Espresso(beans=ground regular), steamedMilk=steamed milk)
[main @barista-1#3] Processing order: cappuccino: beans=regular milk=non-fat milk price=$3.50
[main @barista-1#3] Grinding beans
[main @barista-2#4] Serve: Cappuccino(order=cappuccino: beans=premium milk=breve price=$5.50,
espressoShot=Espresso(beans=ground premium), steamedMilk=steamed milk)
[main @barista-2#4] Processing order: cappuccino: beans=decaf milk=whole milk price=$4.00
```

Notice how both Baristas are concurrently processing different orders. Also, notice that the two coroutines are executing on a single thread (main thread). This is great! Now we have a way for our two Baristas to concurrently process orders and communicate with the cashier.

Properties of Channels

Channels form the foundational component for communicating between coroutines. A `Channel` implements both the `SendChannel` and `ReceiveChannel` interface. Let's take a look at some of the properties of channels.

Suspending Execution while Sending or Receiving

Sending on a channel or receiving from a channel can suspend execution. We saw above, when the cashier sends an order on a channel, the coroutine suspends execution until another coroutine is able to receive from that channel. Similarly, we saw that the Barista suspends execution when receiving from the channel until an order is available on the channel.

Types of Channel Buffers

Channels offer flexibility in terms of communicating messages between coroutines.

Rendezvous (Unbuffered)

This is the default channel buffer type. It doesn't have a buffer. This is why a coroutine is suspended until both the receiving and sending coroutines come together at the same time to transfer the data.

```
val channel = Channel<Menu>(capacity = Channel.RENDEZVOUS)
```



To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

X

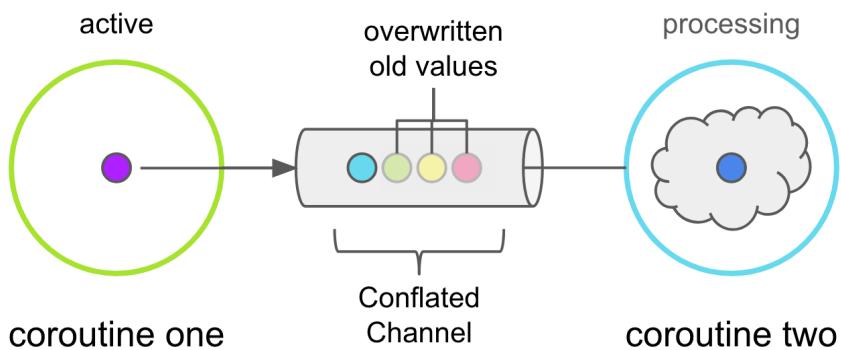
coroutine one

coroutine two

Conflated

This creates a channel with a fixed buffer of size 1. If the receiving coroutine can't keep up with producer, the producer overwrites the last item in the buffer. When the receiving coroutine is ready to take the next value, it receives the last value sent by the producer coroutine. This also means that the producer coroutine doesn't suspend execution when sending to the channel. The receiving coroutine will still suspend execution until something becomes available on the channel.

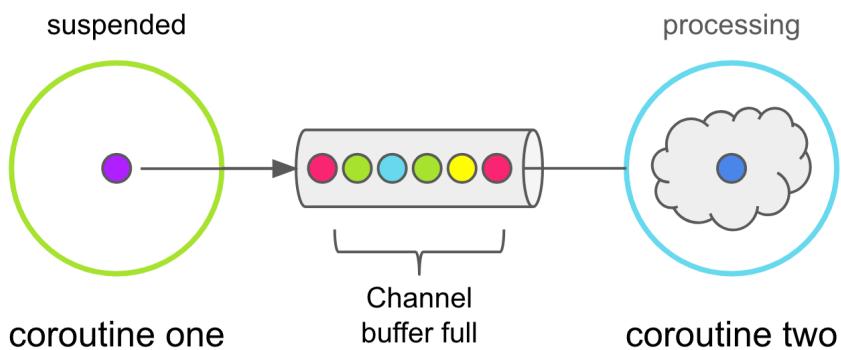
```
val channel = Channel<Menu>(capacity = Channel.CONFLATED)
```



Buffered

This mode creates a channel with a fixed size buffer. The buffer is backed by an Array. The producing coroutine will suspend on send if the buffer is full. The receiving coroutine will suspend if the buffer is empty.

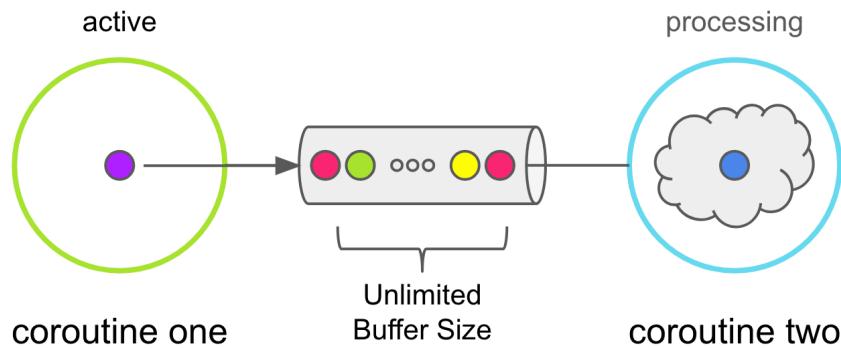
```
val channel = Channel<Menu>(capacity = 10)
```



To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy. X

LinkedList. Items produced too quickly are buffered unboundedly. If the buffer isn't drained, items continue to accumulate until memory is exhausted. This results in an `OutOfMemoryException`. The producing coroutine never suspends sending to the channel. But if the buffer is empty, the receiving coroutine will suspend.

```
val channel = Channel<Menu>(capacity = Channel.UNLIMITED)
```



Coroutine Builders

We've had to associate a channel with a coroutine in order to send to or receive from. But we can use coroutine builders to simplify creating a coroutine and a channel.

Produce

This creates a new `ReceiveChannel`. Internally, it launches a coroutine within a `ProducerScope` to send values on the channel. When there's nothing left to send, the channel is implicitly closed and the coroutine resource is released.

We can simplify the creation of our `orderChannel` in the example above to look like this:

```
// before
val ordersChannel = Channel<Menu>()
launch(CoroutineName("cashier")) {
    for (o in orders) { ordersChannel.send(o) }
    ordersChannel.close()
}

// using produce coroutine builder
val ordersChannel = produce(CoroutineName("cashier")) {
    for (o in orders) { send(o) }
}
```

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

X

Actor

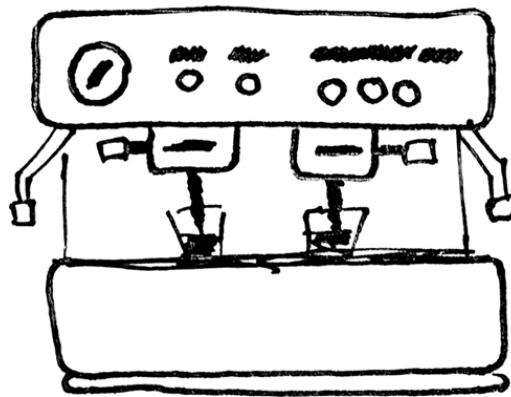
Similar to `produce`, this creates a new `SendChannel`. Internally, it launches a coroutine within an `ActorScope` used to receive values on the channel. You can think of this like exposing a mailbox and internally processing those items within the context of that Actor coroutine.

Tip: Unlike the `produce` coroutine builder, you'll need to explicitly stop the actor when it's no longer needed. Be sure to call `actor.close()`

Backpressure

If you're coming from the RxJava world, then you're probably familiar with the concept and importance of backpressure. What's great about channels is they have backpressure built right in. Backpressure is propagated upstream based on the different channel buffer modes used.

Adding the Espresso Machine



Our Coffee Shop implementation currently supports two Baristas making coffee. But, conceptually, it's like they're using two instances of an espresso machine. How do we construct an espresso machine that the two Baristas can share?

The Baristas also currently execute each step sequentially. There's opportunity to optimize this. How can we change our program so the Baristas can steam the milk while pulling a shot of espresso?

Our espresso machine has two steam wands and two portafilters. The input to a steam wand is milk and the output is steamed milk. The input to a portafilter is ground coffee

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy. X

One approach is to model these two functions as suspendable. That way the caller will provide the input (type of milk or type of ground coffee beans) and await the output (steamed milk or an espresso shot). But what about the internals of the espresso machine? How does the espresso machine limit the number of espresso shots pulled at the same time?

```
class EspressoMachine(scope: CoroutineScope): CoroutineScope by scope {
    suspend fun pullEspressoShot(groundBeans: CoffeeBean.GroundBeans): Espresso {
        // TODO
    }

    suspend fun steamMilk(milk: Milk): Milk.SteamedMilk {
        // TODO
    }
}
```

What if we constructed two channels, one for the portafilter and one for the steam wand, with a fixed buffer size of 2. We could then send the input to the appropriate channel. Using the channel is a good way to communicate. But setting the buffer size to 2 doesn't mean we're pulling two espresso shots at once. Rather, it means we can have up to two pending espresso shot requests while pulling one.

What if we create a channel for each portafilter. And similarly, we create a channel for each steam wand. This is closer to what we want. We can launch a coroutine for each portafilter and associate each portafilter with a channel. This is a good candidate for an actor.

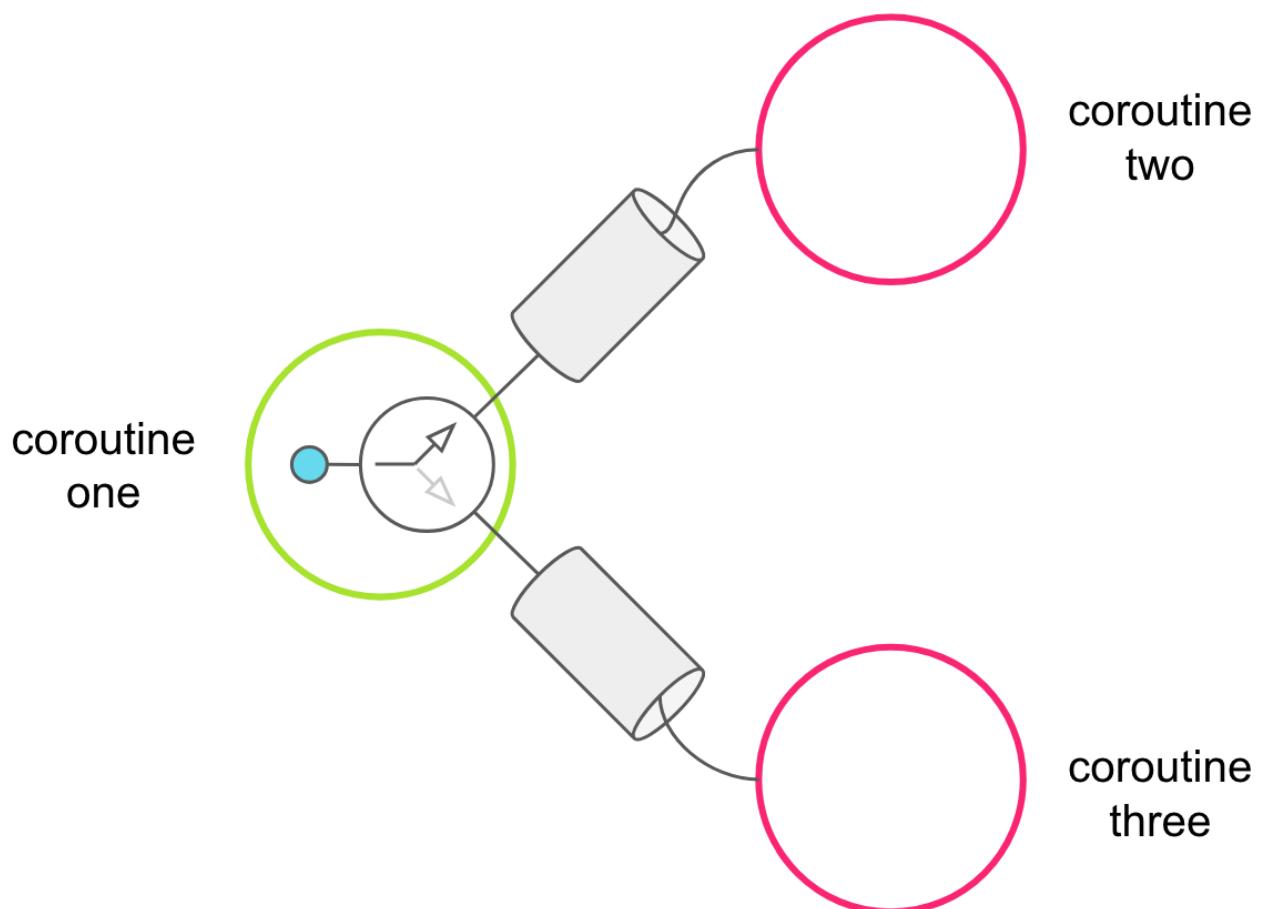
```
class EspressoMachine(scope: CoroutineScope): CoroutineScope by scope {
    private val portafilterOne: SendChannel<PullEspressoShotRequest> = actor() {
        consumeEach {
            log("Pulling espresso shot on portafilter one")
            delay(20)
            // TODO
        }
    }
}
```

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including
cookie policy.

```
private val steamWandOne: SendChannel<SteamMilkRequest> = actor() {  
    consumeEach {  
        log("Steaming milk with steam wand one")  
        delay(10)  
        // TODO  
    }  
}  
  
private val steamWandTwo: SendChannel<SteamMilkRequest> = actor() {  
    // removed for brevity  
}  
// rest of implementation removed for brevity  
}
```

Now we just need a way to select the portafilter to send to. And if both portafilters are in use then we should suspend until one becomes available. The same applies for the two steam wands.

Conceptually, this is what we're trying to do:



To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including
cookie policy.

X

Selecting channels

What we need is a way to select a channel to send to (or receive from). Kotlin provides just that with the `select` expression. It works a lot like a `switch` statement but for channels. `select` picks the first channel that is ready. By ready we mean this could be the first channel that is ready to send to or ready to receive from. The `select` expression suspends if none of the channels are ready.

Here's what the updated espresso machine code looks like now:

```
class EspressoMachine(scope: CoroutineScope): CoroutineScope by scope {
    data class PullEspressoShotRequest(val groundBeans: CoffeeBean.GroundBeans,
                                       val espressoChan: SendChannel<Espresso>)

    data class SteamMilkRequest(val milk: Milk, val steamMilkChan: SendChannel<Milk.SteamedMilk>)

    private val portafilterOne: SendChannel<PullEspressoShotRequest> = actor() {
        consumeEach {
            log("Pulling espresso shot on portafilter one")
            delay(20)
            it.espressoChan.send(Espresso(it.groundBeans))
            it.espressoChan.close()
        }
    }

    // removed portafilter two for brevity

    private val steamWandOne: SendChannel<SteamMilkRequest> = actor() {
        consumeEach {
            log("Steaming milk with steam wand one")
            delay(10)
            it.steamMilkChan.send(Milk.SteamedMilk(it.milk))
            it.steamMilkChan.close()
        }
    }

    // removed steamWandTwo two for brevity

    suspend fun pullEspressoShot(groundBeans: CoffeeBean.GroundBeans): Espresso {
        return select<Espresso> {
            val chan = Channel<Espresso>()
            val req = PullEspressoShotRequest(groundBeans, chan)
            portafilterOne.onSend(req) {
                chan.receive()
            }
            portafilterTwo.onSend(req) {
                chan.receive()
            }
        }
    }

    suspend fun steamMilk(milk: Milk): Milk.SteamedMilk{
        return select<Milk.SteamedMilk> {
            val chan = Channel<Milk.SteamedMilk>()
            val req = SteamMilkRequest(milk, chan)
            steamWandOne.onSend(req) {
                chan.receive()
            }
            steamWandTwo.onSend(req) {
                chan.receive()
            }
        }
    }
}
```

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including our cookie policy.

X

Because both functions are pretty much the same, we'll focus on `pullEspressoShot`. The function selects over the two portafilter channels to send to. But we need a way to communicate the result from the portafilter actor back to the select statement. This is why we create a channel and pass that along with the request to the portafilter. Once the request is sent to the channel, we wait for a response and deliver the result. The portafilter implementation sends the result on the provided channel and then closes the channel.

But we're not done yet. We created four channels, two for the steam wand and two for the portafilters. Those channels are also associated with coroutines. We created them as actors. That means we must close the actors. We introduce a `shutdown` function to do that.

```
class EspressoMachine(scope: CoroutineScope): CoroutineScope by scope {
    fun shutdown() {
        portafilterOne.close()
        portafilterTwo.close()
        steamWandOne.close()
        steamWandTwo.close()
    }
}
```

Have a look at the complete espresso machine here.

The Complete Picture

Now we have a way to share the Espresso Machine between coroutines. Let's update the program to take advantage of this. We can also pull an espresso shot and steam the milk at the same time (try it out).

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

```

    val ordersChannel = produce(CoroutineName("cashier")) {
        for (o in orders) { send(o) }
    }

    // pass along the espresso machine to the makeCoffee function
    coroutineScope {
        launch(CoroutineName("barista-1")) { makeCoffee(ordersChannel, espressoMachine) }
        launch(CoroutineName("barista-2")) { makeCoffee(ordersChannel, espressoMachine) }
    }
    // we need to shutdown the machine which terminates the actors launched internally
    espressoMachine.shutdown()
}

private suspend fun makeCoffee(orders: ReceiveChannel<Menu>, espressoMachine: EspressoMachine) {
    for (o in orders) {
        log("Processing order: $o")
        when (o) {
            is Menu.Cappuccino -> {
                val groundBeans = grindCoffeeBeans(o.beans())
                coroutineScope {
                    // pull an espresso shot but do it async
                    val espressoShotDeferred = async { espressoMachine.pullEspressoShot(groundBeans) }
                    // steam the milk but do it async
                    val steamedMilkDeferred = async { espressoMachine.steamMilk(o.milk()) }
                    // wait for both the espresso shot and steamed milk and then combine them
                    val cappuccino = makeCappuccino(o, espressoShotDeferred.await(),
                        steamedMilkDeferred.await())
                    log("Serve: $cappuccino")
                }
            }
        }
    }
}

```

We create an instance of the Espresso Machine and pass that along to the `makeCoffee` function. Inside the `makeCoffee` function, we request an espresso shot and steamed milk from the espresso machine. But we want to do both of these asynchronously. By wrapping the call within an `async` block, we launch a coroutine and receive a `Deferred`. We can call `await` on the `Deferred` to receive the actual value.

• • •

The purpose of this article was to explain the basics of channels and coroutines. But there's important information about both channels and coroutines that can get you in to some trouble. Channels and coroutines are no silver bullet for avoiding the familiar concurrency problems. Running in to deadlocks, memory leaks, and data races are still possible. But the concepts certainly make reasoning about concurrency simpler.

We looked at the fundamentals of coroutines and channels. We took a simple sequential program and turned it into a concurrent one. We looked at a few different patterns on how to share data and communicate across coroutines.

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including
cookie policy.

X

Further reading

Structured concurrency by Roman Elizarov

A great post explaining the importance of structured concurrency with coroutines.

Kotlin Docs: Channels

The Kotlin docs describe a handful of ways to leverage channels

Deadlocks in non-hierarchical CSP by Roman Elizarov

Channels and coroutines are a great way to reason about concurrency but they don't prevent deadlocks. This is a great post that walks you through a real problem and the gotchas.

GopherCon 2018: Rethinking Classical Concurrency Patterns by Bryan C. Mills

A talk on concurrency patterns using go's concurrency primitives (goroutines and channels).

Programming

Kotlin

Kotlin Beginners

Android App Development

Coroutine

About Help Legal