Chapter 9. Channels

In the previous chapter, you learned how to create coroutines, cancel them, and deal with exceptions. So you know that if task B requires the result of task A, you can implement them as two suspending functions called sequentially. What if task A produces a stream of values? `async` and suspending functions don't fit this use case. This is what `Channel`s[1] are meant for—making coroutines communicate. In this chapter you'll learn in detail what channels are and how to use them.

Using nothing but channels and coroutines, we can design complex asynchronous logic using *communicating sequential processes* (CSP). What is CSP? Kotlin was inspired by several existing programming languages, such as Java, C#, JavaScript, Scala, and Groovy. Notably, Go (the language) inspired coroutines with its "goroutines."

In computer science, CSP is a concurrent programming language which was first described by Tony Hoare in 1978. It has evolved ever since, and the term CSP is now essentially used to describe a programming style. If you're familiar with the Actor model, CSP is quite similar—although there are some differences. If you've never heard of CSP, don't worry—we'll briefly explain the *idea* behind it with practical examples. For now, you can think of CSP as a programming style.

As usual, we'll start with a bit of theory, then implement a real-life problem. In the end, we'll discuss the benefits and trade-offs of CSP, using coroutines.

Channels Overview

Going back to our introductory example, imagine that one task asynchronously produces a list of three `Item` instances (the producer), and another task acts on each of those items (the consumer). Since the producer doesn't return immediately, you could implement it like the following `getItems` suspending function:

```
suspend fun getItems(): List<Item> {
    val items = mutableListOf<Item>()
    items.add(makeItem())
    items.add(makeItem())
```

```
        items.add(makeItem())
        return items
}

suspend fun makeItem(): Item {
    delay(10) // simulate some asynchronism
    return Item()
}
```

As for the consumer, which consumes each of those items, you could simply implement it like so:

```
fun consumeItems(items: List<Item>) {
        for (item in items) println("Do something with $item")
}
```

Putting it all together:

```
fun main() = runBlocking {
        val items = getItems()
        consumeItems(items)
}
```

As you would expect, “Do something with ..” is printed three times. However, in this case, we’re most interested in the order of execution. Let’s take a closer look at what’s really happening, as shown in Figure 9-1.

In Figure 9-1, item consumption only begins after all items have been produced. Producing items might take quite some time, and waiting for all of them to be produced isn’t acceptable in some situations. Instead, we could act on each asynchronously produced item, as shown in Figure 9-2.

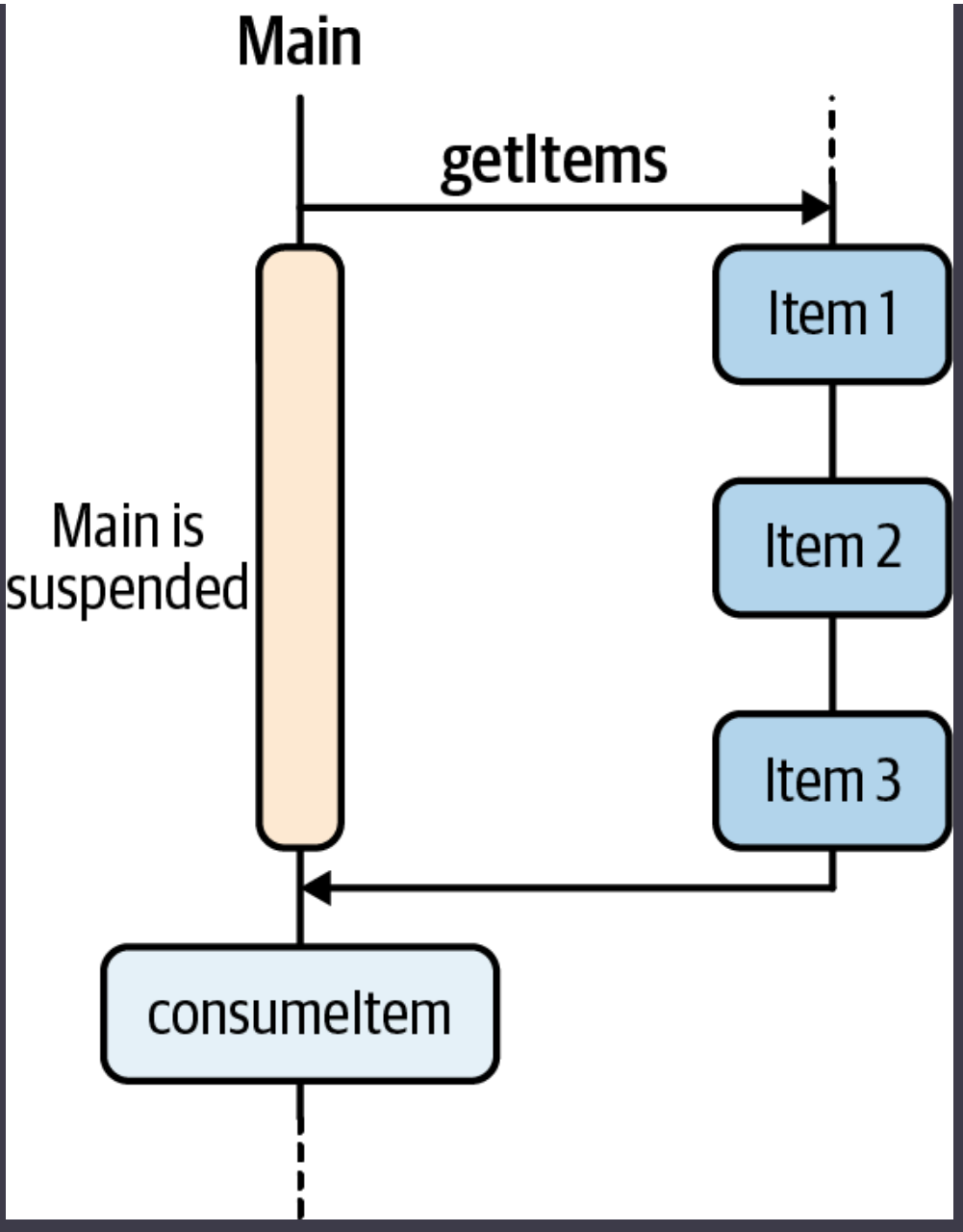

Figure 9-1. Process all at once.

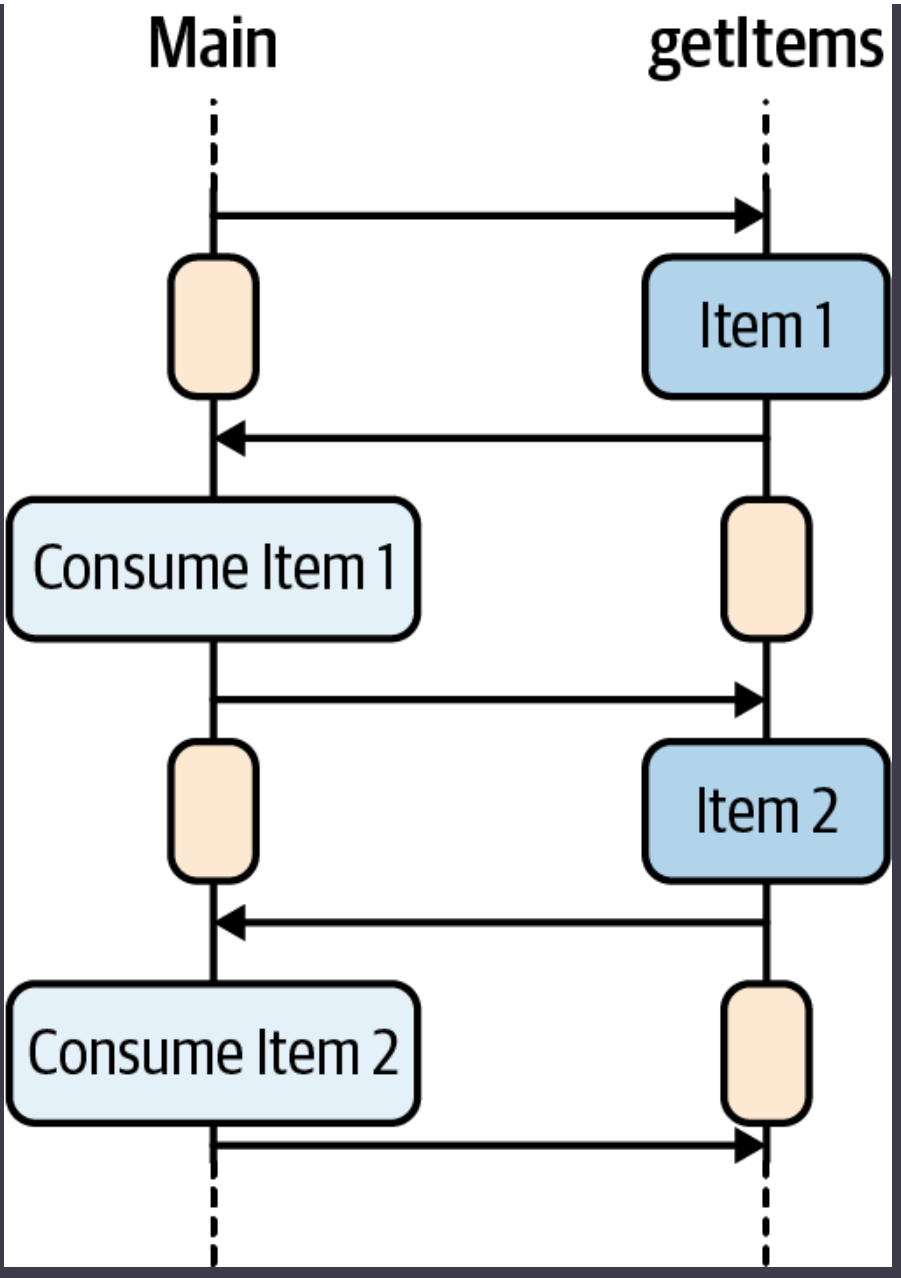

Figure 9-2. Process one after another.

To achieve this, we can't implement `getItems` as a suspending function like before. A coroutine should act as a producer of `Item` instances, and send them to the main coroutine. It's a typical producer-consumer problem.

In Chapter 5, we explained how `BlockingQueue`s can be used to implement *work queues*—or, in this case, a data queue. As a reminder, a `BlockingQueue` has blocking methods `put` and `take` to respectively insert and take an object from the queue. When the queue is used as the only means of communication between two threads (a producer and a consumer), it offers the great benefit of avoiding a shared mutable state. Moreover, if the queue is bounded (has a size limit), a too-fast producer will eventually get blocked in a `put` call if consumers are too slow. This

is known as back pressure: a blocked producer gives the consumers the opportunity to catch up, thus releasing the producer.

Using a `BlockingQueue` as a communication primitive between coroutines wouldn't be a great idea, since a coroutine shouldn't involve blocking calls. Instead, coroutines can suspend. A `Channel` can be seen just like that: a queue with suspending functions `send` and `receive`, as shown in Figure 9-3. A `Channel` also has nonsuspending counterparts: `trySend` and `tryReceive`. These two methods are also nonblocking. `trySend` tries to immediately add an element to the channel, and returns a wrapper class around the result. That wrapper class, `ChannelResult<T>`, also indicates the success or the failure of the operation. `tryReceive` tries to immediately retrieve an element from the channel, and returns a `ChannelResult<T>` instance.

Figure 9-3. Channel.

Like queues, `Channel`s come in several flavors. We'll cover each of those `Channel` variants with basic examples.

Rendezvous Channel

"Rendezvous" is a French word that means "appointment" or "a date"—it depends on the context (we don't mean `CoroutineContext` here). A rendezvous channel does not have any buffer at all. An element is transferred from sender to receiver only when `send` and `receive` invocations meet in time (rendezvous), so `send` suspends until another coroutine invokes `receive`, and `receive` suspends until another coroutine invokes `send`.

As another way to put it, a rendezvous channel involves a back-and-forth communication between producers (coroutines calling `send`) and consumers (coroutines calling `receive`). There can't be two consecutive `send`s without a `receive` in the middle.

By default, when you create a channel using `Channel<T>()`, you get a rendezvous channel.

We can use a rendezvous channel to correctly implement our previous example:

```
fun main() = runBlocking {
    val channel = Channel<Item>()
    launch {                          ❶
        channel.send(Item(1))         ❸
        channel.send(Item(2))         ❹
        println("Done sending")
    }

    println(channel.receive())        ❷
    println(channel.receive())        ❺

    println("Done!")
}

data class Item(val number: Int)
```

The output of this program is:

```
Item(number=1)
Item(number=2)
Done!
Done sending
```

In this example, the main coroutine starts a child coroutine with `launch`, at ❶, then reaches ❷ and suspends until some coroutine sends an `Item` instance in the channel. Shortly after, the child coroutine sends the first item at ❸, then reaches and suspends at the second `send` call at ❹ until some coroutine is ready to receive an item. Subsequently, the main coroutine (which is suspended at ❷) is resumed and receives the first item from the channel and prints it. Then the main coroutine reaches ❺ and immediately receives the second item since the child coroutine was already suspended in a `send` call. Immediately after, the child coroutine continues its execution (prints “Done sending”).

Iterating over a Channel

A `Channel` can be iterated over, using a regular `for` loop. Note that since channels aren’t regular collections,[2] you can’t use `forEach` or other similar functions from the Kotlin Standard Library. Here, channel iteration is a specific language-level feature that can only be done using the `for`-loop syntax:

```
for (x in channel) {
   // do something with x every time some coroutine sends an element in
```

```
    // the channel
}
```

Implicitly, `x` is equal to `channel.receive()` at each iteration. Consequently, a coroutine iterating over a channel could do so indefinitely, unless it contains conditional logic to break the loop. Fortunately, there's a standard mechanism to break the loop: closing the channel. Here is an example:

```
fun main() = runBlocking {
    val channel = Channel<Item>()
    launch {
        channel.send(Item(1))
        channel.send(Item(2))
        println("Done sending")
        channel.close()
    }

    for (x in channel) {
        println(x)
    }
    println("Done!")
}
```

This program has similar output, with a small difference:

```
Item(number=1)
Item(number=2)
Done sending
Done!
```

This time, "Done sending" appears before "Done!" This is because the main coroutine only leaves the `channel` iteration when `channel` is closed. And that happens when the child coroutine is done sending all elements.

Internally, closing a channel sends a special token into the channel to indicate that no other elements will be sent. As items in the channel are consumed *serially* (one after another), all items sent to the rendezvous channel before the close special token are guaranteed to be sent to the receiver.

WARNING

Beware—trying to call `receive` from an already-closed channel will throw a `ClosedReceiveChannelException`. However, trying to iterate on such a channel doesn't throw any exception:

```
fun main() = runBlocking {
    val channel = Channel<Int>()
    channel.close()

    for (x in channel) {
        println(x)
    }
    println("Done!")
}
```

The output is: `Done!`

Other flavors of Channel

In the previous example, the `Channel` appears to be created using a class constructor. If you look at the source code, you can see that it's actually a public function named with a capital C, to give the illusion that you're using a class constructor:

```
public fun <E> Channel(capacity: Int = RENDEZVOUS): Channel<E> =
    when (capacity) {
        RENDEZVOUS -> RendezvousChannel()
        UNLIMITED -> LinkedListChannel()
        CONFLATED -> ConflatedChannel()
        BUFFERED -> ArrayChannel(CHANNEL_DEFAULT_CAPACITY)
        else -> ArrayChannel(capacity)
    }
```

You can see that this `Channel` function has a `capacity` parameter that defaults to `RENDEZVOUS`. For the record, if you step into the `RENDEZVOUS` declaration, you can see that it's equal to 0. For each `capacity` value there is a corresponding channel implementation. There are four different flavors of channels: rendezvous, *unlimited*, *conflated*, and *buffered*. Don't pay too much attention to the concrete implementations (like `RendezvousChannel()`), because those classes are internal and may change in the future. On the other hand, the values `RENDEZVOUS`, `UNLIMITED`, `CONFLATED`, and `BUFFERED` are part of the public API.

We'll cover each of those channel types in the next sections.

Unlimited Channel

An *unlimited* channel has a buffer that is only limited by the amount of available memory. Senders to this channel never suspend, while receivers only suspend when the channel is empty. Coroutines exchanging data via an *unlimited* channel don't need to meet in time.

At this point, you might be thinking that such a channel should have concurrent modification issues when senders and receivers are executed from different threads. After all, coroutines are dispatched on threads, so a channel might very well be used from different threads. Let's check the `Channel`'s robustness ourselves! In the following example, we send `Int`s from a coroutine dispatched on `Dispatchers.Default` while another coroutine reads the same channel from the main thread, and if the `Channel`s aren't thread-safe, we will notice:

```
fun main() = runBlocking {
    val channel = Channel<Int>(UNLIMITED)
    val childJob = launch(Dispatchers.Default) {
        println("Child executing from ${Thread.currentThread().name}")
        var i = 0
        while (isActive) {
            channel.send(i++)
        }
        println("Child is done sending")
    }

    println("Parent executing from ${Thread.currentThread().name}")
    for (x in channel) {
        println(x)

        if (x == 1000_000) {
            childJob.cancel()
            break
        }
    }

    println("Done!")
}
```

The output of this program is:

```
Parent executing from main
Child executing from DefaultDispatcher-worker-2
0
1
..
1000000
Done!
Child is done sending
```

You can run this sample as much as you want, and it always completes without concurrent issues. That's because a `Channel` internally uses a lock-free algorithm.[3]

NOTE

`Channel`s are thread-safe. Several threads can concurrently invoke `send` and `receive` methods in a thread-safe way.

Conflated Channel

This channel has a buffer of size 1, and only keeps the last sent element. To create a *conflated* channel, you invoke `Channel<T>(Channel.CONFLATED)`. For example:

```
fun main() = runBlocking {
    val channel = Channel<String>(Channel.CONFLATED)

    val job = launch {
        channel.send("one")
        channel.send("two")
    }

    job.join()
    val elem = channel.receive()
    println("Last value was: $elem")
}
```

The output of this program is:

```
Last value was: two
```

The first sent element is "one." When "two" is sent, it replaces "one" in the channel. We wait until the coroutine-sending elements complete, us-

ing `job.join()`. Then we read the value `two` from the channel.

Buffered Channel

A *buffered* channel is a `Channel` with a fixed capacity—an integer greater than 0. Senders to this channel don't suspend unless the buffer is full, and receivers from this channel don't suspend unless the buffer is empty. To create a buffered channel of `Int` with a buffer of size 2, you would invoke `Channel<Int>(2)`. Here is an example of usage:

```
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>(2)

    launch {
        for (i in 0..4) {
            println("Send $i")
            channel.send(i)
        }
    }

    launch {
        for (i in channel) {
            println("Received $i")
        }
    }
}
```

The output of this program is:

```
Send 0
Send 1
Send 2
Received 0
Received 1
Received 2
Send 3
Send 4
Received 3
Received 4
```

In this example, we've defined a `Channel` with a fixed capacity of 2. A coroutine attempts to send five integers, while another coroutine consumes elements from the channel. The sender coroutine manages to send 0 and 1 in one go, then attempts to send 3. The `println("Send $i")` is executed for the value 3 but the sender coroutine gets suspended in the

`send` call. The same reasoning applies for the consumer coroutine: two elements are received consecutively with an additional print before suspending.

Channel Producers

Until now, you've seen that a `Channel` can be used for both sending *and* receiving elements. Sometimes you might want to be more explicit about how a channel should be used for either sending or receiving. When you're implementing a `Channel` that is meant to be read only by other coroutines, you can use the `produce` builder:

```
fun CoroutineScope.produceValues(): ReceiveChannel<String> = produce {
    send("one")
    send("two")
}
```

As you can see, `produce` returns a `ReceiveChannel`—which only has methods relevant to receiving operations (`receive` is among them). An instance of `ReceiveChannel` cannot be used to send elements.

TIP

Also, we've defined `produceValues()` as an extension function of `CoroutineScope`. Calling `produceValues` will start a new coroutine that sends elements into a channel. There's a convention in Kotlin: every function that starts coroutines should be defined as an extension function of `CoroutineScope`. If you follow this convention, you can easily distinguish in your code which functions are starting new coroutines from suspending functions.

The main code that makes use of `produceValues` could be:

```
fun main() = runBlocking {
    val receiveChannel = produceValues()

    for (e in receiveChannel) {
        println(e)
    }
}
```

Conversely, a `SendChannel` only has methods relevant to sending operations. Actually, looking at the source code, a `Channel` is an interface de-

riving from both `ReceiveChannel` and `SendChannel`:

```
public interface Channel<E> : SendChannel<E>, ReceiveChannel<E> {
    // code removed for brevity
}
```

Here is how you can use a `SendChannel`:

```
fun CoroutineScope.collectImages(imagesOutput: SendChannel<Image>) {
    launch(Dispatchers.IO) {
        val image = readImage()
        imagesOutput.send(image)
    }
}
```

Communicating Sequential Processes

Enough of the theory, let's get started and see how channels can be used to implement a real-life problem. Imagine that your Android application has to display "shapes" in a canvas. Depending on the inputs of the user, your application has to display an arbitrary number of shapes. We're purposely using generic terms—a shape could be a point of interest on a map, an item in a game, anything that may require some background work like API calls, file reads, database queries, etc. In our example, the main thread, which already handles user input, will simulate requests for new shapes to be rendered. You can already foresee that it's a producer-consumer problem: the main thread makes requests, while some background task handles them and returns the results to the main thread.

Our implementation should:

- Be thread-safe
- Reduce the risk of overwhelming the device memory
- Have no thread contention (we won't use locks)

Model and Architecture

A `Shape` is made of a `Location` and some useful `ShapeData`:

```
data class Shape(val location: Location, val data: ShapeData)
data class Location(val x: Int, val y: Int)
class ShapeData
```

Given a `Location`, we need to fetch the corresponding `ShapeData` to build a `Shape`. So in this example, `Location`s are the input, and `Shape`s the output. For brevity, we'll use the words "location" for `Location` and "shape" for `Shape`.

In our implementation, we'll distinguish two main components:

view-model

: This holds most of the application logic related to shapes. As the user interacts with the UI, the view gives the view-model a list of locations.

`shapeCollector`

: This is responsible for fetching shapes given a list of locations.

Figure 9-4 illustrates the bidirectional relationship between the view-model and the shape collector.

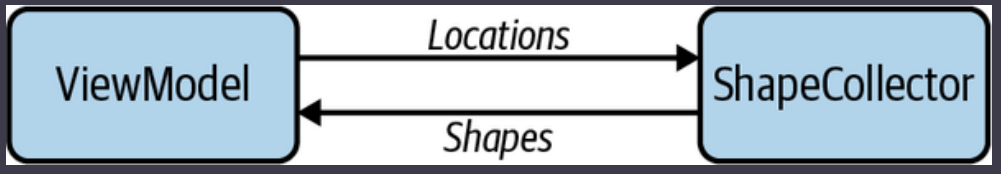

Figure 9-4. High-level architecture.

The `ShapeCollector` follows a simple process:

```
                fetchData
Location ---------------------> ShapeData
```

As an additional prerequisite, our `ShapeCollector` should maintain an internal "registry" of locations being processed. Upon receiving a location to process, the `ShapeCollector` shouldn't attempt to download it if it's already being processed.

A First Implementation

We can start with this first naïve implementation of the `ShapeCollector`, which is far from being complete, but you'll get the idea:

```
class ShapeCollector {
    private val locationsBeingProcessed = mutableListOf<Location>()

    fun processLocation(location: Location) {
        if (locationsBeingProcessed.add(location)) {
            // fetch data, then send back a Shape instance to
            // the view-model
        }
    }
}
```

If we were programming with threads, we would have several threads sharing an instance of `ShapeCollector`, executing `processLocation` concurrently. Using this approach, however, leads to sharing mutable states. In the previous snippet, `locationsBeingProcessed` is one example.

As you learned in Chapter 5, making mistakes using locks is surprisingly easy. Using coroutines, we don't have to share mutable state. How? Using coroutines and channels, we can *share by communicating* instead of *communicate by sharing*.

The key idea is to encapsulate mutable states inside coroutines. In the case of the list of `Location`s being processed, it can be done with:

```
launch {
    val locationsBeingProcessed = mutableListOf<Location>()      ❶

    for (location in locations) {                                ❷
        // same code from previous figure
    }
}
```

❶ In the preceding example, only the coroutine that started with `launch` can touch the mutable state, which is `locationsBeingProcessed`.

❷ However, we now have a problem. How do we provide the `location`s? We have to somehow provide this iterable to the coroutine. So we'll use a `Channel`, and use it as input of a function we'll declare. Since we're launching a coroutine inside a function, we declare this function as an extension function of `CoroutineScope`:

```
private fun CoroutineScope.collectShapes(
     locations: ReceiveChannel<Location>
) = launch {
     // code removed for brevity
}
```

As this coroutine will be receiving `Location`s from the view-model, we declare the `Channel` as a `ReceiveChannel`. By the way, you've seen in the previous section that a `Channel` can be iterated over, just like a list. So now, we can fetch the corresponding `ShapeData` for each `Location` instance received from the channel. As you'll want to do this in parallel, you might be tempted to write something like so:

```
private fun CoroutineScope.collectShapes(
     locations: ReceiveChannel<Location>
) = launch {
     val locationsBeingProcessed = mutableListOf<Location>()

     for (loc in locations) {
          if (!locationsBeingProcessed.contains(loc) {
               launch(Dispatchers.IO) {
                    // fetch the corresponding `ShapeData`
               }
          }
     }
}
```

Beware, as there's a catch in this code. You see, for each received location, we start a new coroutine. Potentially, this code might start a lot of coroutines if the `locations` channel debits a lot of items. For this reason, this situation is also called *unlimited concurrency*. When we introduced coroutines, we said that they are lightweight. It's true, but the work they do might very well consume significant resources. In this case, `launch(Dispatchers.IO)` in itself has an insignificant overhead, while fetching the `ShapeData` could require a REST API call on a server with limited bandwidth.

So we'll have to find a way to limit concurrency—we don't want to start an unlimited number of coroutines. When facing this situation with threads, a common practice is to use a thread pool coupled with a work queue (see Chapter 5). Instead of a thread pool, we'll create a *coroutine pool*, which we'll name *worker pool*. Each coroutine from this worker pool will perform the actual fetch of `ShapeData` for a given location. To communicate with this worker pool, `collectShapes` should use an ad-

ditional channel to which it can send locations to the worker pool, as shown in Figure 9-5.

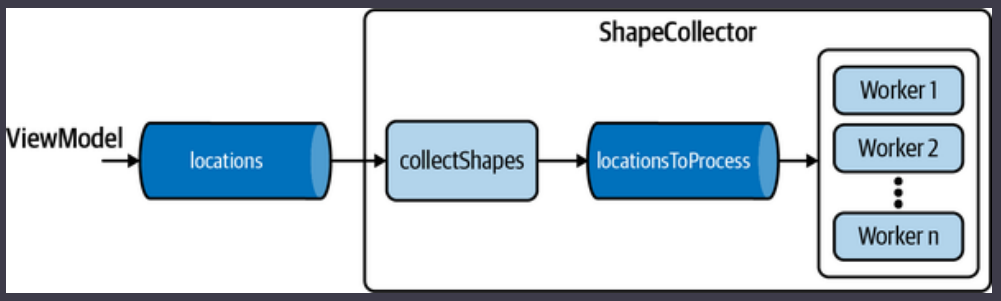

Figure 9-5. Limit concurrency.

WARNING

When you use `Channel`s, *be careful not to have unlimited concurrency*. Imagine that you have to instantiate a lot of `Bitmap` instances. The underlying memory buffer which stores pixel data takes a nonnegligible amount of space in memory. When working with a lot of images, allocating a fresh instance of `Bitmap` every time you need to create an image causes significant pressure on the system (which has to allocate memory in RAM while the garbage collector cleans up all the previously created instances that aren't referenced anymore). A canonical solution to this problem is `Bitmap` pooling, which is only a particular case of the more general pattern of *object pooling*. Instead of creating a fresh instance of `Bitmap`, you can pick one from the pool (and reuse the underlying buffer when possible).

This is how you would modify `collectShapes` to take an additional channel parameter:

```
private fun CoroutineScope.collectShapes(
     locations: ReceiveChannel<Location>,
     locationsToProcess: SendChannel<Location>,
) = launch {
     val locationsBeingProcessed = mutableListOf<Location>()

     for (loc in locations) {
         if (!locationsBeingProcessed.contains(loc) {
              launch(Dispatchers.IO) {
                   locationsToProcess.send(loc)
              }
         }
    }
}
```

Notice how `collectShapes` now sends a location to the `locationsToProcess` channel, only if the location isn't in the list of locations currently being processed.

As for the worker implementation, it simply reads from the channel we just created—except that from the worker perspective, it's a `ReceiveChannel`. Using the same pattern:

```
private fun CoroutineScope.worker(
        locationsToProcess: ReceiveChannel<Location>,
) = launch(Dispatchers.IO) {
        for (loc in locationsToProcess) {
             // fetch the ShapeData, see later
        }
}
```

For now, we are not focusing on how to fetch a `ShapeData`. The most important notion to understand here is the `for` loop. Thanks to the iteration on the `locationsToProcess` channel, each individual `worker` coroutine will receive its own location without interfering with the others. No matter how many workers we'll start, a location sent from `collectShapes` to the `locationsToProcess` channel will only be received by one worker. You'll see that each worker will be created with the same channel instance when we wire all those things up. In message-oriented software, this pattern, which implies delivery of a message to multiple destinations, is called *fan-out*.

Looking back at the missing implementation inside the `for` loop, this is what we'll do:

1. Fetch the `ShapeData` (which from now on we'll simply refer to as "data").
2. Create a `Shape` from the location and the data.
3. Send the shape to some channel, which other components in our application will use to get the shapes from `ShapeCollector`. Obviously, we haven't created such a channel yet.
4. Notify the `collectShapes` coroutine that the given location has been processed, by sending it back to its sender. Again, such a channel has to be created.

Do note that this isn't the only possible implementation. You could imagine other ways and adapt to your needs. After all, this is what this chapter

is all about: to give you examples and inspiration for your next developments.

Back on our horse, Example 9-1 shows the final implementation of the `worker` coroutine.

Example 9-1. Worker coroutine

```
private fun CoroutineScope.worker(
    locationsToProcess: ReceiveChannel<Location>,
    locationsProcessed: SendChannel<Location>,
    shapesOutput: SendChannel<Shape>
) = launch(Dispatchers.IO) {
    for (loc in locationsToProcess) {
        try {
            val data = getShapeData(loc)
            val shape = Shape(loc, data)
            shapesOutput.send(shape)
        } finally {
            locationsProcessed.send(loc)
        }
    }
}
```

Just like the `collectShapes` was adapted earlier to take one channel as an argument, this time we're adding two more channels: `locationsProcessed` and `shapesOutput`.

Inside the `for` loop, we first get a `ShapeData` instance for a location. For the sake of this simple example, Example 9-2 shows our implementation.

Example 9-2. `Getting shape data`

```
private suspend fun getShapeData(
    location: Location
): ShapeData = withContext(Dispatchers.IO) {
        /* Simulate some remote API delay */
        delay(10)
        ShapeData()
}
```

Since the `getShapeData` method might not return immediately, we implement it as a `suspend` function. Imagining that the downstream code involves a remote API, we use `Dispatchers.IO`.

The `collectShapes` coroutine has to be adapted again, since it has to accept one more channel—the one from which the workers send back locations they're done processing. You're starting to get used to it—it'll be a `ReceiveChannel` from the `collectShapes` perspective. Now `collectShapes` accepts two `ReceiveChannel`s and one `SendChannel`.

Let's try it:

```
private fun CoroutineScope.collectShapes(
     locations: ReceiveChannel<Location>,
     locationsToProcess: SendChannel<Location>,
     locationsProcessed: ReceiveChannel<Location>
): Job = launch {
     ...
     for (loc in locations) {
          // same implementation, hidden for brevity
     }
     // but.. how do we iterate over locationsProcessed?
}
```

Now we have a problem. How can you receive elements from multiple `ReceiveChannel`s at the same time? If we add another `for` loop right below the `locations` channel iteration, it wouldn't work as intended as the first iteration only ends when the `locations` channel is closed.

For that purpose, you can use the `select` expression.

The select Expression

The `select` expression waits for the result of multiple suspending functions simultaneously, which are specified using *clauses* in the body of this select invocation. The caller is suspended until one of the clauses is either *selected* or *fails*.

In our case, it works like so:

```
select<Unit> {
    locations.onReceive { loc ->
        // do action 1
    }
    locationsProcessed.onReceive { loc ->
        // do action 2
```

```
    }
}
```

If the `select` expression could talk, it would say: “Whenever the `locations` channel receives an element, I’ll do action 1. Or, if the `locationsProcessed` channel receives something, I’ll do action 2. I can’t do both actions at the same time. By the way, I’m returning `Unit`.”

The “I can’t do both actions at the same time” is important. You might wonder what would happen if action 1 takes half an hour—or worse, if it never completes. We’ll describe a similar situation in “Deadlock in CSP”. However, the implementation that follows is guaranteed *never* to block for a long time in each action.

Since `select` is an expression, it returns a result. The result type is inferred by the return type of the lambdas we provide for each case of the `select`—pretty much like the `when` expression. In this particular example, we don’t want any result, so the return type is `Unit`. As `select` returns after either the `locations` or `locationsProcessed` channel receives an element, it doesn’t iterate over channels like our previous `for` loop. Consequently, we have to wrap it inside a `while(true)`. The complete implementation of `collectShapes` is shown in Example 9-3.

Example 9-3. Collecting shapes

```
private fun CoroutineScope.collectShapes(
    locations: ReceiveChannel<Location>,
    locationsToProcess: SendChannel<Location>,
    locationsProcessed: ReceiveChannel<Location>
) = launch(Dispatchers.Default) {

    val locationsBeingProcessed = mutableListOf<Location>()

    while (true) {
        select<Unit> {
            locationsProcessed.onReceive {                    ❶
                locationsBeingProcessed.remove(it)
            }
            locations.onReceive {                             ❷
                if (!locationsBeingProcessed.any { loc ->
                    loc == it }) {
                    /* Add it to the list of locations being processed */
                    locationsBeingProcessed.add(it)

                    /* Now download the shape at location */
```

```
                    locationsToProcess.send(it)
                }
            }
        }
    }
}
```

1. When the `locationsProcessed` channel receives a location, we know that this location has been processed by a worker. It should now be removed from the list of locations being processed.

2. When the `locations` channel receives a location, we have to first check whether we've already been processing the same location or not. If not, we'll add the location to the `locationsBeingProcessed` list, and then send it to the `locationsToProcess` channel.

Putting It All Together

The final architecture of the `ShapeCollector` takes shape, as shown in Figure 9-6.

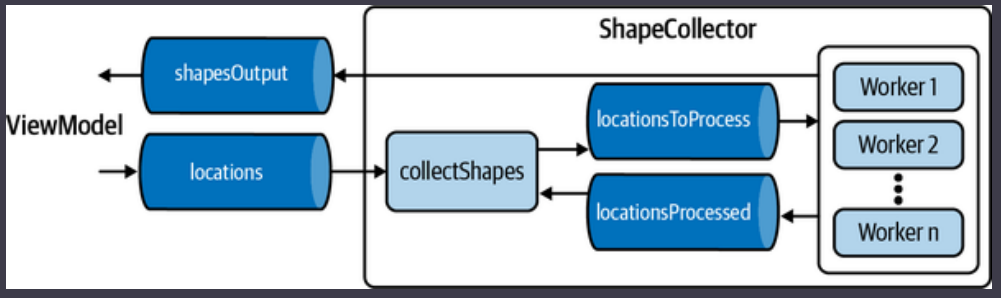

Figure 9-6. Final architecture.

Remember that all the channels we used to implement the `collectShapes` and `worker` methods have to be created somewhere. To respect encapsulation, a good place to do that is in a `start` method, as shown in Example 9-4.

Example 9-4. Shape collector

```
class ShapeCollector(private val workerCount: Int) {
    fun CoroutineScope.start(
        locations: ReceiveChannel<Location>,
        shapesOutput: SendChannel<Shape>
    ) {
        val locationsToProcess = Channel<Location>()
        val locationsProcessed = Channel<Location>(capacity = 1)
```

```
            repeat(workerCount) {
                worker(locationsToProcess, locationsProcessed, shapesOutput)
            }
            collectShapes(locations, locationsToProcess, locationsProcessed)
        }

        private fun CoroutineScope.collectShapes // already implemented

        private fun CoroutineScope.worker        // already implemented

        private suspend fun getShapeData         // already implemented
    }
```

This `start` method is responsible for starting the whole shape collection machinery. The two channels that are exclusively used inside the `ShapeCollector` are created: `locationsToProcess` and `locationsProcessed`. We are not explicitly creating `ReceiveChannel` or `SendChannel` instances here. We're creating them as `Channel` instances because they'll further be used either as `ReceiveChannel` or `SendChannel`. Then the worker pool is created and started, by calling the `worker` method as many times as `workerCount` was set. It's achieved using the `repeat` function from the standard library.

Finally, we call `collectShapes` once. Overall, we started `workerCount + 1` coroutines in this `start` method.

You might have noticed that `locationsProcessed` is created with a capacity of 1. This is intended, and is an important detail. We'll explain why in the next section.

Fan-Out and Fan-In

You just saw an example of multiple coroutines receiving from the same channel. Indeed, all `worker` coroutines receive from the same `locationsToProcess` channel. A `Location` instance sent to the `locationsToProcess` channel will be processed by only one worker, without any risk of concurrent issues. This particular interaction between coroutines is known as *fan-out*, as shown in Figure 9-7. From the standpoint of the coroutine started with the `collectShapes` function, locations are fanned-out to the worker pool.

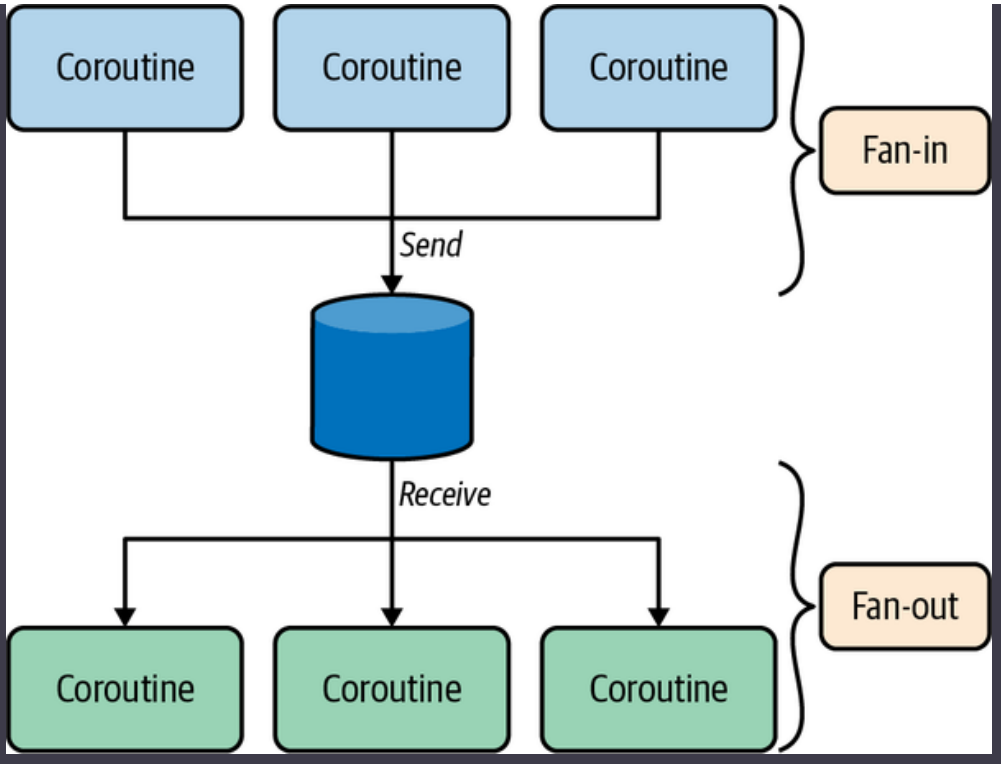

Figure 9-7. Fan-out and fan-in.

Fan-out is achieved by launching several coroutines which all iterate over the same instance of `ReceiveChannel` (see the `worker` implementation in Example 9-1). If one of the workers fails, the other ones will continue to receive from the channel—making the system resilient to some extent.

Inversely, when several coroutines send elements to the same `SendChannel` instance, we're talking about *fan-in*. Again, you've got a good example since all workers send `Shape` instances to `shapesOutput`.

Performance Test

Alright! Time to test the performance of our `ShapeCollector`. The following snippet has a `main` function, which calls the functions `consumeShapes` and `sendLocations`. Those functions start a coroutine that, respectively, consumes `Shape` instances from the `ShapeCollector` and sends `Location` instances. Overall, this code is close to what you'd write in a real view-model, as shown in Example 9-5.

Example 9-5. Shape collector

```
fun main() = runBlocking<Unit> {
    val shapes = Channel<Shape>()                ❶
    val locations = Channel<Location>()
```

```
    with(ShapeCollector(4)) {                    ❷
        start(locations, shapes)
        consumeShapes(shapes)
    }

    sendLocations(locations)
}

var count = 0

fun CoroutineScope.consumeShapes(
    shapesInput: ReceiveChannel<Shape>
) = launch {
    for (shape in shapesInput) {
        // increment a counter of shapes
        count++                                  ❸
    }
}

fun CoroutineScope.sendLocations(
    locationsOutput: SendChannel<Location>
) = launch {
    withTimeoutOrNull(3000) {                    ❹
        while (true) {
            /* Simulate fetching some shape location */
            val location = Location(Random.nextInt(), Random.nextInt())
            locationsOutput.send(location)
        }
    }
    println("Received $count shapes")
}
```

❶ We set up the channels according to the needs of the `ShapeCollector`—see Figure 9-4.

❷ We create a `ShapeCollector` with four workers.

❸ The `consumeShapes` function only increments a counter. That counter is declared globally—which is fine because the coroutine started with `consumeShapes` is the only one to modify `count`.

❹ In the `sendLocations` functions, we set up a timeout of three seconds. `withTimeoutOrNull` is a suspending function that suspends until the provided time is out. Consequently, the coroutine started with `sendLocations` only prints the received count after three seconds.

If you recall the implementation of `getShapeData` in Example 9-2, we added `delay(10)` to simulate a suspending call of 10 ms long. Running four workers for three seconds, we would ideally receive 3,000 / 10 × 4 = 1,200 shapes, if our implementation had zero overhead. On our test machine, we got 1,170 shapes—that's an efficiency of 98%.

Playing a little bit with more workers (64), with `delay(5)` in each worker, we got 122,518 shapes in 10 seconds (the ideal number being 128,000)—that's an efficiency of 96%.

Overall, the throughput of `ShapeCollector` is quite decent, event with a `sendLocations` function that continuously sends `Location` instances without any pause between two sends.

Back Pressure

What happens if our workers are too slow? This could very well happen if a remote HTTP call takes time to respond, or a backend server is overwhelmed—we don't know. To simulate this, we can dramatically increase the delay inside `getShapeData` (see Example 9-2). Using `delay(500)`, we got only 20 shapes in three seconds, with four workers. The throughput decreased, but this isn't the interesting part. As always with producer-consumer problems, issues can arise when consumers slow down—as producers might accumulate data and the system may ultimately run out of memory. You can add `println()` logs inside the producer coroutine and run the program again:

```
fun CoroutineScope.sendLocations(locationsOutput: SendChannel<Location>) = la
    withTimeoutOrNull(3000) {
        while (true) {
            /* Simulate fetching some shape location */
            val location = Location(Random.nextInt(), Random.nextInt())
            println("Sending a new location")
            locationsOutput.send(location)      // suspending call
        }
    }
    println("Received $count shapes")
}
```

Now, "Sending a new location" is printed only about 25 times in the console.

So the producer is being slowed down. How?

Because `locationsOutput.send(location)` is a suspending call. When workers are slow, the `collectShapes` function (see Example 9-3) of the `ShapeCollector` class quickly becomes suspended at the line `locationsToProcess.send(it)`. Indeed, `locationsToProcess` is a rendezvous channel. Consequently, when the coroutine started with `collectShapes` reaches that line, it's suspended until a worker is ready to receive the location from `locationsToProcess`. When the previously mentioned coroutine is suspended, it can no longer receive from the `locations` channel—which corresponds to `locationsOutput` in the previous example. This is the reason why the coroutine that started with `sendLocation` is in turn suspended. When workers finally do their job, `collectShapes` can resume, and so does the producer coroutine.

Similarities with the Actor Model

In CSP, you create coroutines that encapsulate mutable state. Instead of communicating by sharing their state, they share by communicating (using `Channel`s). The coroutine started with the `collectShapes` function (see Example 9-3) uses three channels to communicate with other coroutines—one `SendChannel` and two `ReceiveChannel`s, as shown in Figure 9-8.

In CSP parlance, `collectShapes` and its three channels is a *process*. A process is a computational entity that communicates with other actors using asynchronous message passing (channels). It can do only one thing at a time—reading, writing to channels, or processing.

In the Actor model, an *actor* is quite similar. One noticeable difference is that an actor only has one channel—called a mailbox. If an actor needs to be responsive and nonblocking, it must delegate its long-running processing to child actors. This similarity is the reason why CSP is sometimes referred to as an Actor model implementation.

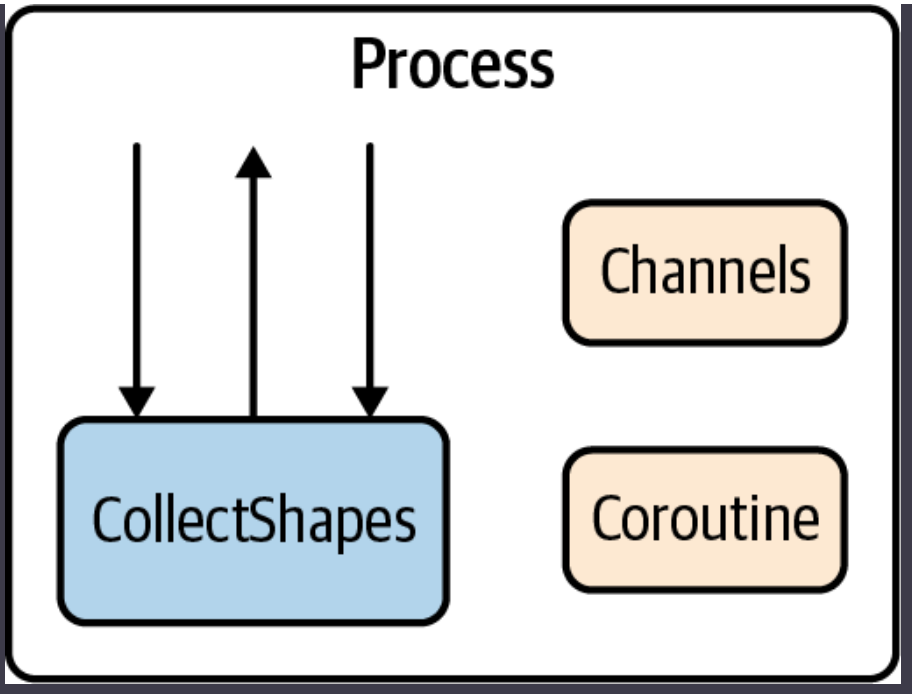

Figure 9-8. Process in CSP.

Execution Is Sequential Inside a Process

We've just seen that a *process* is made of a single coroutine and channels. The very nature of a coroutine is for it to be executed on some thread. So unless this coroutine starts other child coroutines (which run concurrently, and in some cases in parallel), all lines of that coroutine are executed sequentially. That includes receiving from channels, sending objects to other channels, and mutating some private state. Consequently, the actors implemented in this chapter could either receive from a channel or send to another channel, but not do both at the same time. Under load, this kind of actor can be efficient because it doesn't involve blocking calls, only suspending functions. When a coroutine is suspended, the overall efficiency isn't necessarily affected, because the thread executing the suspended coroutine can then execute another coroutine which has something to do. This way, threads can be used to their full potential, never contending to some lock.

Final Thoughts

This mechanism using CSP style has very little internal overhead. Thanks to `Channel`s and coroutines, our implementation is lock-free. Therefore, there's *no thread contention*—the `ShapeCollector` is less likely to impact other threads of your application. Similarly, there's a chance that the `Dispatchers` we use in the `ShapeCollector` might also be used in other features in our application. By leveraging lock-free implementations, a coroutine suspended while receiving from a channel won't pre-

vent the underlying thread from executing other coroutines. In other words, we can do more with the same resources.

Moreover, this architecture provides built-in back pressure. If some `ShapeData` instances suddenly take more time to fetch, producers of `ShapeLocation` instances will be slowed down so that locations don't accumulate—which reduces the risk of running out of memory. This back pressure comes for free—you didn't explicitly write code for such a feature.

The example given in this chapter is generic enough to be taken as is and adapted to fit your needs. In the event that you need to significantly deviate from our example, then we owe you a deeper explanation. For example, why did we set a capacity of 1 for the `locationsProcessed` channel in Example 9-4? The answer is admittedly nontrivial. If we had created a regular rendezvous channel, our `ShapeCollector` would have suffered from a *deadlock*—which brings us to the next section.

Deadlock in CSP

Deadlocks are most commonly encountered when working with threads. When thread A holds lock 1 and attempts to seize lock 2, while thread B holds lock 2 and attempts to seize lock 1, you have a deadlock. The two threads indefinitely wait for each other and neither progresses. Deadlocks can have disastrous consequences when they happen in critical components of an application. An efficient way to avoid such a situation is to ensure that a deadlock cannot happen under any imaginable circumstances. Even when conditions are highly unlikely to be met, you can trust Murphy's Law to strike some day.

However, deadlocks can also happen in CSP architecture. We can do a little experiment to illustrate this. Instead of setting a capacity of 1 to the channel `locationsProcessed` in Example 9-4, let's use a channel with no buffer (a rendezvous channel) and run the performance test sample in Example 9-5. The result printed in the console is:

```
Received 4 shapes
```

For the record, we should have received 20 shapes. So, what's going on?

NOTE

Fair warning: the following explanation goes into every necessary detail, and is quite long. We encourage you to take the time to read it carefully until the end. It's the ultimate challenge to test your understanding of channels.

You might also skip it entirely and jump to "TL;DR".

Let's have a closer look at the internals of our `ShapeCollector` class and follow each step as though we were a live debugger. Imagine that you've just started the performance test sample in Example 9-5, and the first `Location` instance is sent to the `locations` channel. That location goes through the `collectShapes` method with its `select` expression. At that moment, `locationsProcessed` has nothing to provide, so the `select` expression goes through the second case: `locations.onReceive{..}`. If you look at what's done inside this second case, you can see that a location is sent to the `locationsToProcess` channel—which is a receive channel for each worker. Consequently, the coroutine started by the `collectShapes` method (which we'll refer to as the `collectShapes` coroutine) is suspended at the `locationsToProcess.send(it)` invocation until a worker handshakes the `locationsToProcess` rendezvous channel. This happens fairly quickly, since at that time all workers are idle.

When a worker receives the first `Location` instance, the `collectShapes` coroutine is resumed and is able to receive other locations. As in our worker implementation, we've added some delay to simulate a background processing, you can consider workers slow compared to other coroutines—which are the `collectShapes` coroutine and the producer coroutine started with the `sendLocations` method in the test sample (which we'll refer to as the `sendLocations` coroutine). Therefore, another location is received by the `collectShapes` coroutine while the worker that which took the first location is still busy processing it. Similarly, a second worker quickly handles the second location, and a third location is received by the `collectShapes` coroutine, etc.

The execution continues until all four workers are busy, while a fifth location is received by the `collectShapes` coroutine. Following the same logic as before, the `collectShapes` coroutine is suspended until a worker is ready to take the `Location` instance. Unfortunately, all workers are busy. So the `collectShapes` coroutine isn't able to take incoming locations anymore. Since the `collectShapes` and `sendLocations` coroutines communicate through a rendezvous channel, the

`sendLocations` coroutine is in turn suspended until `collectShapes` is ready to take more locations.

Time goes by until a worker makes itself available to receive the fifth location. Eventually, a worker (probably the first worker) is done processing its `Location` instance. Then it sends the result to the `shapesOutput` channel and it tries to send back the processed location to the `collectShapes` coroutine, using the `locationsProcessed` channel. Remember that this is our mechanism to notify the `collectShapes` coroutine when a location has been processed. However, the `collectShapes` coroutine is suspended at the `locationsToProcess.send(it)` invocation. So `collectShapes` can't receive from the `locationsProcessed` channel. There's no issue to this situation: this is a *deadlock*,[4] as shown in Figure 9-9.

Eventually, the first four locations processed by the workers are processed and four `Shape` instances are sent to the `shapesOutput` channel. The delay in each worker is only of 10 ms, so all workers have time to complete before the three-second timeout. Hence the result:

```
Received 4 shapes
```

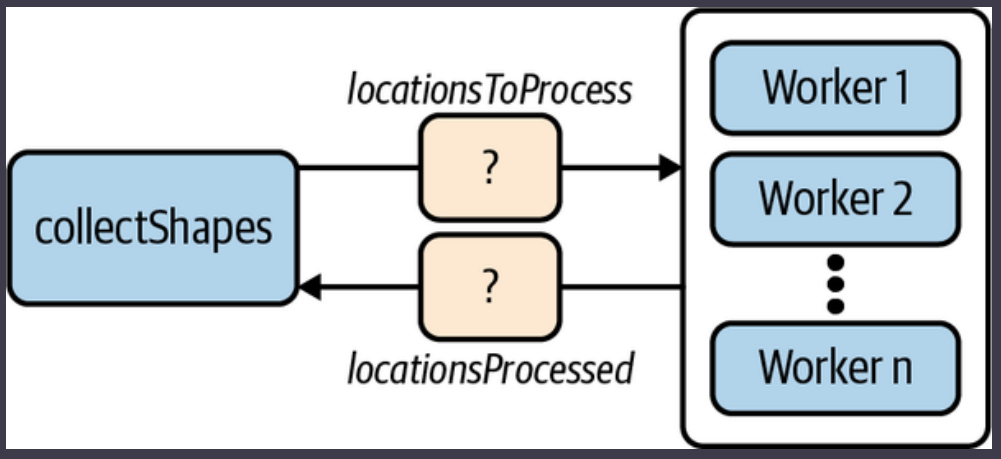

Figure 9-9. Deadlock in CSP.

If the `locationsProcessed` channel had a capacity of at least 1, the first available worker would have been able to send back its `Location` instance and then receive from the `locationsToProcess` channel—releasing the `collectShapes` coroutine. Subsequently, in the `select` expression of the `collectShapes` coroutine, the `locationsToProcess` channel is *always* checked before the `locations` channel. This ensures that when the `collectShapes` coroutine is eventually suspended at the `locationsToProcess.send(it)` invocation, the buffer of the `locationsProcessed` channel is guaranteed to be empty—so a worker

can send a location without being suspended. If you're curious, try to revert the two cases `locationsProcessed.onReceive {..}` and `locations.onReceive {..}` while having a capacity of 1 for the `locationsProcessed` channel. The result will be: "Received 5 shapes."

TL;DR

Not only is the capacity of 1 for the `locationsProcessed` channel extremely important, the order in which channels are read in the `select` expression of the `collectShapes` coroutine also matters.[5] What should you remember from this? Deadlocks are possible in CSP. Even more important, understanding what caused the deadlock is an excellent exercise to test your understanding of how channels work.

If we look back at the structure of the `ShapeCollector`, we can represent the structure as a cyclic graph, as shown in Figure 9-10.

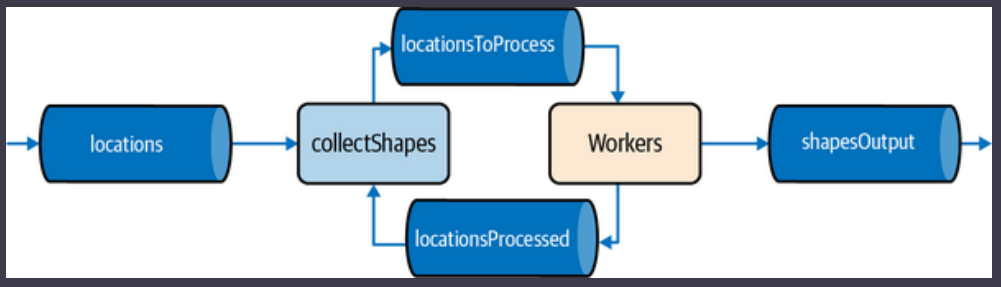

Figure 9-10. Cyclic graph.

This new representation emphasizes an important property of the structure: it's *cyclic*. `Location` instances travel back and forth between the `collectShapes` coroutine and workers.

Cycles in CSP are actually the cause of deadlocks. Without cycles, there's no possibility of deadlock. Sometimes, however, you'll have no choice but to have those cycles. In this case, we gave you the key ideas to reason about CSP, so you can find solutions by yourself.

Limitations of Channels

Up until now, we've held off on discussing the limitations of channels, so we'll describe some of those limitations now. Using notions from this chapter, creating a stream of `Int` values is typically done as shown in Example 9-6.

Example 9-6. Producing numbers

```
fun CoroutineScope.numbers(): ReceiveChannel<Int> = produce {
    send(1)
    send(2)
    // send other numbers
}
```

On the receiving side, you can consume those numbers like so:

```
fun main() = runBlocking {
    val channel = numbers()
    for (x in channel) {
        println(x)
    }
}
```

Pretty straightforward. Now, what if you need to apply a transformation for each of those numbers? Imagine that your transformation function was:

```
suspend fun transform(n: Int) = withContext(Dispatchers.Default) {
    delay(10) // simulate some heavy CPU computations
    n + 1
}
```

You could modify the `numbers` function like so:

```
fun CoroutineScope.numbers(): ReceiveChannel<Int> = produce {
    send(transform(1))
    send(transform(2))
}
```

It works, but it's not elegant. A much nicer solution would look like this:

```
fun main() = runBlocking {
    /* Warning - this doesn't compile */
    val channel = numbers().map {
        transform(it)
    }
    for (x in channel) {
        println(x)
    }
}
```

Actually, as of Kotlin 1.4, this code doesn't compile. In the early days of channels, we had "channel operators" such as `map`. However, those operators have been deprecated in Kotlin 1.3, and removed in Kotlin 1.4.

Why? Channels are communication primitives between coroutines. They are specifically designed to distribute values so that every value is received by only one receiver. It's not possible to use channels to broadcast values to multiple receivers. The designers of coroutines have created `Flow`s specifically for asynchronous data streams on which we can use transformation operators; we'll see how in the next chapter.

So, channels are not a convenient solution to implement pipelines of data transformations.

Channels Are Hot

Let's have a look at the source code of the `produce` channel builder. Two lines are interesting, as shown in the following:

```
public fun <E> CoroutineScope.produce(                          ❶
    context: CoroutineContext = EmptyCoroutineContext,
    capacity: Int = 0,
    @BuilderInference block: suspend ProducerScope<E>.() -> Unit
): ReceiveChannel<E> {
    val channel = Channel<E>(capacity)
    val newContext = newCoroutineContext(context)
    val coroutine = ProducerCoroutine(newContext, channel)
    coroutine.start(CoroutineStart.DEFAULT, coroutine, block)   ❷
    return coroutine
}
```

❶ `produce` is an extension function on `CoroutineScope`. Remember the convention? It indicates that this function starts a new coroutine.

❷ We can confirm that with the `coroutine.start()` invocation. Don't pay too much attention to how this coroutine is started—it's an internal implementation.

Consequently, when you invoke the `produce` channel builder, a new coroutine is started and immediately starts producing elements and send-

ing them to the returned channel even if no coroutine is consuming those elements.

This is the reason why channels are said to be *hot*: a coroutine is actively running to produce or consume data. If you know RxJava, this is the same concept as hot observables: they emit values independently of individual subscriptions. Consider this simple stream:

```
fun CoroutineScope.numbers(): ReceiveChannel<Int> = produce {
    use(openConnectionToDatabase()) {
        send(1)
        send(2)
    }
}
```

Also, imagine that no other coroutines are consuming this stream. As this function returns a rendezvous channel, the started coroutine will suspend on the first `send`. So you might say: "OK, we're fine—no background processing is done until we provide a consumer to this stream." It's true, but if you forget to consume the stream, the database connection will remain open—notice that we used the `use` function from the standard library, which is the equivalent of the `try`-with-`resources` statement in Java. While it might not be harmful as is, this piece of logic could be part of a retry loop, in which case a significant amount of resources would leak.

To sum up, channels are intercoroutine communication primitives. They work really well in a CSP-like architecture. However, we don't have handy operators such as `map` or `filter` to transform them. We can't broadcast values to multiple receivers. Moreover, their hot nature can cause memory leaks in some situations.

Flows have been created to address those channels' limitations. We'll cover flows in the next chapter.

Summary

- Channels are communication primitives that provide a way to transfer streams of values between coroutines.
- While channels are conceptually close to Java's `BlockingQueue`, the fundamental difference is that `send` and `receive` methods of a channel are suspending functions, not blocking calls.

- Using channels and coroutines, you can *share by communicating* instead of the traditional *communicate by sharing*. The goal is to avoid shared mutable-state and thread-safety issues.
- You can implement complex logic using CSP style, leveraging back pressure. This results in potentially excellent performance since the nonblocking nature of suspending functions reduces thread contention to its bare minimum.
- Beware that deadlock in CSP is possible, if your architecture has cycles (a coroutine sends objects to another coroutine, while also receiving objects from the same coroutine). You can fix those deadlocks by, for example, tweaking the order in which the `select` expression treats each cases, or by adjusting the capacity of some channels.
- Channels should be considered low-level primitives. Deadlocks in CSP are one example of misuse of channels. The next chapter will introduce *flows*—higher-level primitives that exchange streams of data between coroutines. It doesn't mean that you shouldn't use channels—there are still situations where channels are necessary (the `ShapeCollector` in this chapter is an example). However, you'll see that in many situations, flows are a better choice. In any case, it's important to know about channels because (as you'll see) flows sometimes use channels under the hood.

1 We'll sometimes refer to `Channel`s as channels in the rest of this chapter.

2 Specifically, `Channel` doesn't implement `Iterable`.

3 If you want to learn how such an algorithm works, we recommend that you read Section 15.4, "NonBlocking Algorithms," in *Java Concurrency in Practice*, by Brian Goetz et al. There is also this interesting YouTube video, Lock-Free Algorithms for Kotlin Coroutines (Part 1) from Roman Elizarov, lead designer of Kotlin coroutines.

4 While there's no lock or mutex involved here, the situation is very similar to a deadlock involving threads. This is why we use the same terminology.

5 Actually, our implementation, which uses a capacity of 1 for `locationsProcessed`, isn't the only possible implementation that works without deadlocks. There's at least one solution that uses `locationsProcessed` as a rendezvous channel. We leave this as an exercise for the reader.