Chapter 7: Controlling the Data Flow

The previous chapter covered an important **Kotlin** concurrency primitive: **coroutines**. In this chapter, we'll discuss two other vital concurrent primitives in Kotlin: **channels** and **flows**. We'll also touch on **higher-order functions** for **collections**, as their API is very similar to that of channels and flows.

The idea of making extensive use of small, reusable, and composable functions comes directly from the **functional programming** paradigm, which we discussed in the previous chapter. These functions allow us to write code in a manner that describes *what* we want to do instead of *how* we want to do it.

In this chapter, we'll cover the following topics:

- Reactive principles
- Higher-order functions for collections
- Concurrent data structures
- Sequences
- Channels
- Flows

After reading this chapter, you'll be able to efficiently communicate between different coroutines and process your data with ease.

Technical requirements

In addition to the technical requirements from the previous chapters, you will also need a **Gradle**-enabled Kotlin project to be able to add the required dependencies.

You can find the source code used in this chapter on **GitHub** at the following location:

https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter07

Reactive principles

We'll start this chapter with a brief detour into **Reactive programming**, as it forms the foundation of the **data streaming** concept.

Reactive programming is a paradigm based on functional programming in which we model our logic as a set of operations in a data stream. The fundamental concepts of reactive programming are summarized nicely in *The Reactive Manifesto* (https://www.reactivemanifesto.org).

According to this manifesto, reactive programs should be all of the following:

- Responsive
- Resilient
- Elastic
- Message-driven

To understand these four principles, we'll use an example.

Let's imagine you are calling your Internet Service Provider, since your internet is slow, for example. *Do you have this picture in your mind?* Let's start then.

Responsive principle

How much time are you willing to spend waiting on the line? That depends on the urgency of the situation and how much time you have. If you're in a hurry, you'll probably drop the call sooner rather than later because you don't know how much time you'll need to wait while listening to that horrible music.

That's the system being *unresponsive* to you. This also happens with web systems. A request to a web server may get stuck in a queue when waiting for other requests to be processed.

On the other hand, a responsive call center may tell you in a pleasant voice once in a while how many people are in the queue before you – or even how much time you'll have to wait.

In both cases, the result is the same. You've wasted your time waiting on the line. But the second system was responsive to your needs, and you could make decisions based on that.

Resilient principle

Let's move on to the **resilient** principle. Imagine you're waiting on the line for 10 minutes and then the line drops. That's the system not being resilient to failures.

The Reactive Manifesto recommends several ways to achieve resiliency:

- **Delegation**: You'll probably hear, "*Our current representative is unable to resolve your slow internet; we are redirecting you to someone else.*"
- **Replication**: Then, you may hear, "*We are aware that many people are on the line; we are adding more representatives as we speak.*" This also relates to *elasticity*, which we'll cover in the next section.
- **Containment** and **isolation**: Finally, the automatic voice tells you, "*If you don't want to wait, please leave your number and we'll get back to you.*" *Containment* means that you are now decoupled from the scalability problems the system is having (that is, the system not having enough representatives). In contrast, *isolation* means that even if the system has issues with a phone line not being reliable, you don't care.

Elastic principle

In the previous section, we discussed replication. To prevent failures, our call center always has at least three representatives on shift. Maybe all of them are answering calls, or perhaps they're just patiently waiting.

What happens, though, if some rabid mole chews through the internet cable?

Suddenly, there is a surge of calls from disgruntled customers.

If our call center has only three phones, there is not much we can do about this. But if we had some extra resources, we could bring more representatives in to handle the incident and calm our customers. And after the cable was finally fixed, we could let them go back to their business. That's the system being *elastic* in response to the workload.

Elasticity builds on *scalability*. For example, we could manage all of the incoming calls if each representative could work independently by having their own phone. If we had more representatives than phones, the number of phones would become a *bottleneck*, with some representatives unable to answer any calls.

Message-driven principle

The **message-driven** principle is also referred to as **asynchronous message passing**. So, in the previous section, we saw that if you could leave a message for any representative to call back, it could make the system more resilient.

So, what if all customers only leave messages?

Then, each representative could *prioritize* those messages or *batch* them. For example, printing all of the billing receipts together instead of working through the messages in a random order.

Using messages also allows applying backpressure. If a representative receives too many messages, they may collapse from stress. To avoid that, they may text you to say that you'll have to wait a bit longer to receive your answer. Again, we're also talking about *delegation* here, as all of these principles overlap.

Messages are also *non-blocking*. After you leave the message, you don't sit there waiting for the representative's response. Instead, you usually go back to your regular tasks. The ability to perform other tasks while you wait is one of the cornerstones of *concurrency*.

In this section, we learned about the four reactive principles. Reactive applications are responsive, resilient, elastic, and message-driven. In the following sections, we'll see how these principles are applied in Kotlin. We'll start with *collections*, or as they'd be referred to in reactive programming terms, *static data streams*.

Higher-order functions on collections

We briefly touched on this topic in *Chapter 1*, *Getting Started with Kotlin*, but before we can discuss streams, let's make sure that those of us who come from languages that don't have higher-order functions on collections know what they are, what they do, and what the benefits of using them are.

We won't be able to cover all of the functions available on collections, but we'll cover the most widely used ones.

Mapping elements

The `map()` function takes each element of a collection and returns a new element of a possibly different type. To understand this idea better, let's say we have a list of letters and we would like to output their ASCII values.

First, let's implement it in an imperative way:

```
val letters = 'a'..'z'
val ascii = mutableListOf<Int>()
for (l in letters) {
    ascii.add(l.toInt())
}
```

Notice that even for such a trivial task, we had to write quite a lot of code. We also had to define our output list as mutable.

Now, the same code using the **`map()`** function would look like this:

```
val result: List<Int> = ('a'..'z').map { it.toInt() }
```

Notice how much shorter the implementation is. We don't need to define a mutable list, nor do we need to write a **`for-each`** loop ourselves.

Filtering elements

Another common task is filtering a collection. You know the drill – you iterate over it and only put values that fit your criteria in a new collection. For example, if given a range of numbers between **`1`** and **`100`**, we would like to return only those that are divisible by **`3`** or divisible by **`5`**.

In the imperative way, this function might look something like this:

```
val numbers = 1..100
val notFizzbuzz = mutableListOf<Int>()
for (n in numbers) {
    if (n % 3 == 0 || n % 5 == 0) {
        notFizzbuzz.add(n)
    }
}
```

In its functional variant, we would use the **`filter()`** function:

```
val filtered: List<Int> = (1..100).filter { it % 3 ==
0 ||
  it % 5 == 0 }
```

Again, notice how much more concise our code becomes. We only specify *what* needs to be done, filtering elements that match the criteria, and not *how* this should be done (for example, using an **`if`** statement).

Finding elements

Finding the first element in a collection is another common task. If we were to write a function for finding a number that is divisible by both **`3`** and **`5`**, we could implement it like this:

```
fun findFizzbuzz(numbers: List<Int>): Int? {
    for (n in numbers) {
        if (n % 3 == 0 && n % 5 == 0) {
            return n
        }
    }
    return null
}
```

The same functionality can be achieved using the **`find`** function:

```
val found: Int? = (1..100).find { it % 3 == 0 && it %
5 ==   0 }
```

In a similar way to the preceding imperative function, the **`find`** function returns **`null`** if there is no element that meets our criteria.

There's also an accompanying **`findLast()`** method, which does the same, but which starts with the last element of the collection.

Executing code for each element

All previous families of functions had one common characteristic: they all resulted in a stream. But not all the higher-order functions return streams. Some will return a single value, such as `Unit` or, for example, a number. Those functions are called **terminator functions**.

In this section, we'll deal with the first terminator function. Terminator functions return something else rather than a new collection, so you can't chain the result of this call to other calls. Therefore, they *terminate* the chain.

In the case of `forEach()`, it returns the result of the `Unit` type. The `Unit` type is akin to `void` in **Java** and means that the function doesn't return anything useful. So, the `forEach()` function is like the plain old `for` loop:

```
val numbers = (0..5)
numbers.map { it * it}          // Can continue
       .filter { it < 20 }      // Can continue
       .forEach { println(it) } // Cannot continue
```

Note that `forEach()` has some minor performance impacts compared to the traditional `for` loop.

There's also `forEachIndexed()`, which provides an index in the collection alongside the actual value:

```
numbers.map { it * it }
        .forEachIndexed { index, value ->
    print("$index:$value, ")
}
```

The output for the preceding code will be as follows:

```
> 0:1, 1:4, 2:9, 3:16, 4:25,
```

Since Kotlin 1.1, there's also the `onEach()` function, which is a bit more useful because it returns the collection again:

```
numbers.map { it * it}
       .filter { it < 20 }
```

```
        .sortedDescending()
        .onEach { println(it) } // Can continue now
        .filter { it > 5 }
```

As you can see, this function is not terminating.

Summing up elements

Much like **`forEach()`**, **`reduce()`** is a terminating function. But instead of terminating with **`Unit`**, which is not very useful, it terminates with a single value of the same type as the collection it operates on.

To see how **`reduce()`** works in practice, let's summarize all numbers between **`1`** and **`100`**:

```
val numbers = 1..100
var sum = 0
for (n in numbers) {
    sum += n
}
```

Now, let's write the same code using **`reduce`**:

```
val reduced: Int = (1..100).reduce { sum, n -> sum +
n }
```

Note that here it lets us avoid declaring a mutable variable for storing the sum of the elements. Unlike previous higher-order functions we've seen, **`reduce()`** receives not one but two arguments. The first argument is the accumulator. In the imperative example, it's the **`sum`** variable. The second argument is the next element. We used the same names for the arguments, so it should be relatively easy to compare both implementations.

Getting rid of nesting

Sometimes when working with collections, we may end up with a *collection of collections*. For example, consider the following code:

```
val listOfLists: List<List<Int>> = listOf(listOf(1,
2), listOf(3, 4, 5), listOf(6, 7, 8))
```

But what if we wanted to turn this collection into a single list containing all of the nested elements?

Then, the output would look like this:

```
> [1, 2, 3, 4, 5, 6, 7, 8]
```

One option is to iterate our input and use the **`addAll`** method that the mutable collections have:

```
val flattened = mutableListOf<Int>()
for (list in listOfLists) {
    flattened.addAll(list)
}
```

A better option is to use a **`flatMap()`** function, which will do the same:

```
val flattened: List<Int> = listOfLists.flatMap { it }
```

This concrete example could be simplified even further by using a **`flatten()`** function:

```
val flattened: List<Int> = listOfLists.flatten()
```

But the **`flatMap()`** function is usually more useful, as it allows you to apply other functions to each collection, in an **Adapter** like pattern.

There are many other higher-order functions declared on collections, so we couldn't cover all of them in this short section. You must browse through the official documentation and learn about them. Nevertheless, the functions discussed previously should provide a solid ground for the next topic we'll cover.

Now, when you're familiar with how to transform and iterate over the *static data streams*, let's see how we can apply the same operations to *dynamic data streams*.

Exploring concurrent data structures

Now we're familiar with some of the most common higher-order functions on collections, let's combine this knowledge with what we learned in the previous chapter about concurrency primitives in Kotlin to discuss the *concurrent data structures* Kotlin provides.

The two most essential concurrent data structures are *channels* and *flows*. However, before we can discuss them, we need to look at another data structure: **sequences**. While this data structure is not concurrent itself, it will provide us with a bridge into the concurrent world.

Sequences

Higher-order functions on collections existed in many functional programming languages for a long time. But for Java developers, the higher-order functions for collections first appeared in Java 8 with the introduction of the **Stream API**.

Despite providing developers with valuable functions such as **`map()`**, **`filter()`**, and some of the others we already discussed, there were two major drawbacks to the Stream API. First, in order to use these functions, you had to migrate to Java 8. And second, your collection had to be converted to something called a **stream**, which had all of the functions defined on it. If you want to return a collection again after mapping and filtering your stream, you can collect it back.

There is also another significant difference between streams and collections. Unlike collections, streams can be infinite. Since Kotlin doesn't limit itself to only **JVM** and is also backward-compatible to Java 6, it needed to provide another solution for the possibility of infinite collections. This solution was named **sequence** to avoid clashing with Java streams when they're available.

We can create a new sequence using the **generateSequence()** function. For example, the next function will create an infinite sequence of numbers:

```
val seq: Sequence<Long> = generateSequence(1L) { it +
1 }
```

As the first argument we specify the initial value, while the second argument is a lambda that generates the next value based on the previous one. The returned type, as you can see, is **Sequence**.

A regular collection or a range can be converted to a sequence using the **asSequence()** function:

```
(1..100).asSequence()
```

If we need to build a sequence using more complex logic, you can use a **sequence()** builder:

```
val fibSeq = sequence {
    var a = 0
    var b = 1
    yield(a)
    yield(b)
    while (true) {
        yield(a + b)
        val t = a
        a = b
        b += t
    }
}
```

In this example, we create a sequence of Fibonacci numbers. Then, we use the **yield()** function to return the next value in the series. Every time the sequence is used, the code will resume from the last **yield()** function invoked.

While the concept of sequences doesn't seem very useful in itself, there is a significant difference between sequences and collections. Sequences are *lazy*, while collections are *eager*.

This means that using higher-order functions on collections has a hidden cost for collections beyond a certain size. Most of them will copy the collection for the sake of immutability.

To understand this difference, let's look at the following code. First, we'll create a list containing a million numbers and measure how much time it takes to square each number in the list – once while operating on a *collection* and another while working on a *sequence*:

```
val numbers = (1..1_000_000).toList()
println(measureTimeMillis {
    numbers.map {
        it * it
    }.take(1).forEach { it }
}) // ~50ms
println(measureTimeMillis {
    numbers.asSequence().map {
        it * it
    }.take(1).forEach { it }
}) // ~5ms
```

We use the **`take()`** function, which is another higher-order function on collections, to *take* just the first element of the calculation.

You can see that the code that uses a sequence executes much faster. This is because sequences, being lazy, execute the chain for each element. This means that only a single number from the entire list is squared.

On the other hand, functions on collections work on the entire collection. This means that first, all of the numbers are squared, then put in a new collection, and only a single number is taken from the results.

Sequences, channels, and flows follow the *reactive principles*, so it's essential to understand them before moving on. Note that reactive principles are not tied to functional programming. You can also be reactive while writing object-oriented or procedural code. However, it's still easier to discuss these principles after learning about functional programming and its foundations.

Channels

In the previous chapter, we learned how to spawn coroutines and control them.

But, what if two coroutines need to communicate with each other?

In Java, threads communicate either by using the **`wait()`**/**`notify()`**/**`notifyAll()`** pattern or by using one of the rich set of classes from the **`java.util.concurrent`** package – for example, **`BlockingQueue`**.

In Kotlin, as you may have noticed, there are no **`wait()`**/**`notify()`** methods. Instead, to communicate between coroutines, Kotlin uses channels. **Channels** are very similar to **`BlockingQueue`**, but instead of blocking a thread, channels suspend a coroutine, which is a lot cheaper. We'll use the following steps to create a channel and a coroutine:

1. First, let's create a channel:
   ```
   val chan = Channel<Int>()
   ```
   Channels are typed. This channel can only receive integers.
2. Then, let's create a coroutine that reads from this channel:
   ```
   launch {
       for (c in chan) {
           println(c)
       }
   }
   ```
   Reading from a channel is as simple as using a **`for-each`** loop.
3. Now, let's send some values to this channel. This is as simple as using the **`send()`** function:
   ```
   (1..10).forEach {
   ```

```
        chan.send(it)
    }
    chan.close()
```

4. Finally, we close the channel. Once closed, the coroutine that listens to the channel will also break out of the `for-each` loop, and if there's nothing else to do, the coroutine will terminate.

This style of communication is called **Communicating Sequential Processes**, or more simply, **CSP**.

As you can see, channels are a convenient and type-safe way to communicate between different coroutines. But we had to define the channels manually. In the following two sections, we'll see how this can be further simplified.

Producers

If we need a coroutine that supplies a stream of values, we could use the `produce()` function. This function creates a coroutine that is backed up by `ReceiveChannel<T>`, where `T` is the type the coroutine produces.

We could rewrite the example from the previous section, as follows, by using the `produce()` function:

```
val chan = produce {
    (1..10).forEach {
        send(it)
    }
}
launch {
    for (c in chan) {
        println(c)
    }
}
```

Note that inside the **`produce()`** block, the **`send()`** function is readily available for us to push new values to the channel.

Instead of using a **`for-each`** loop in our consumer coroutine, we can use a **`consumeEach()`** function:

```
launch {
    chan.consumeEach {
        println(it)
    }
}
```

Now, it's time to look at another example where a coroutine is bound to a channel.

Actors

Similar to **`producer()`**, **`actor()`** is a coroutine bound to a channel. But instead of a channel going *out* of the coroutine, there's a channel going *into* the coroutine.

Let's look at the following example:

```
val actor = actor<Int> {
    channel.consumeEach {
        println(it)
    }
}
(1..10).forEach {
    actor.send(it)
}
```

In this example, our main function is again producing the values and the actors consume them through the channel. This is very similar to the first example we saw, but instead of explicitly creating a channel and a separate coroutine, we have them bundled together.

If you've worked with **Scala** or any other programming language that has actors, you may be familiar with a slightly different actor model from what we've described. For example, in some implementations, actors have both inbound and outbound channels (often called **mailboxes**). But in Kotlin, an actor has only an inbound mailbox in the form of a channel.

Buffered channels

In all of the previous examples, whether creating channels explicitly or implicitly, we in fact used their *unbuffered* version.

To demonstrate what this means, let's take a look at a slightly altered example from the previous section:

```
val actor = actor<Long> {
    var prev = 0L
    channel.consumeEach {
        println(it - prev)
        prev = it
        delay(100)
    }
}
```

Here, we have almost the same **`actor`** object, which receives timestamps and prints the difference between every two timestamps it gets. We also introduce a small delay before it can read the next value.

Instead of sending a sequence of numbers, we would send the current timestamp to this **`actor`** object:

```
repeat(10) {
    actor.send(System.currentTimeMillis())
}
actor.close().also { println("Done sending") }
```

Now, let's take a look at the output of our code:

```
> ...
> 101
> 103
> 101
> Done sending
```

Notice that our producer is suspended until the channel is ready to accept the next value. Therefore, the **`actor`** object is able to apply backpressure on the producer, telling it not to send the next value until the **`actor`** object is ready.

Now, let's make a minor change to the way we define our **`actor`** object:

```
val actor = actor<Long>(capacity = 10) {
...
}
```

Every channel has a *capacity*, which is zero by default. This means until a value is consumed from a channel, no other value can be sent over it.

Now, if we run our code again, we'll see a completely different output:

```
> Done sending
> ...
> 0
> 0
```

The producer doesn't have to wait for the consumer anymore because the channel now buffers the messages. So, the messages are sent as fast as possible and the actor is still able to consume them at its own pace.

In a similar manner, **`capacity`** could be defined on the producer channel:

```
val chan = produce(capacity = 10) {
    (1..10).forEach {
        send(it)
    }
}
```

And it could be defined on the raw channel as well:

```
val chan = Channel<Int>(10)
```

Buffered channels are a very powerful concept that allow us to *decouple* producers from consumers. You should use them carefully, though, as the larger the capacity of the channel is, the more memory it will require.

Channels are a relatively low-level concurrency construct. So, let's take a look at another type of stream, which provides us with a higher level of abstraction.

Flows

A **flow** is a cold, asynchronous stream and is an implementation of the **Observable design pattern** we covered in *Chapter 4*, *Getting Familiar with Behavioral Patterns*.

As a quick reminder, the Observable design pattern has two methods: **`subscribe()`** (which allows consumers to, well, subscribe for messages) and **`publish()`** (which sends a new message to all of the subscribers).

The publish method of the **`Flow`** object is called **`emit()`**, while the subscribe method is called **`collect()`**.

We can create a new flow using the **`flow()`** function:

```
val numbersFlow: Flow<Int> = flow {
    ...
}
```

Inside the **`flow`** constructor, we can use the **`emit()`** function to publish a new value to all listeners.

For example, here we create a flow that would publish ten numbers using the **`flow`** constructor:

```
flow {
    (0..10).forEach {
        println("Sending $it")
```

```
            emit(it)
        }
    }
```

Now that we've covered how to publish a message, let's discuss how to subscribe to a flow.

For that, we can use the **`collect()`** function available on the **`flow`** object:

```
numbersFlow.collect { number ->
    println("Listener received $number")
}
```

If you run this code now, you'll see that the listener prints all the numbers it receives from the flow.

Unlike some other reactive frameworks and libraries, there is no special syntax to raise an exception to the listener. Instead, we can simply use the standard **`throw`** expression to do that:

```
flow {
    (1..10).forEach {
    ...
        if (it == 9) {
            throw RuntimeException()
        }
    }
}
```

From the listener side, handling exceptions is as simple as wrapping the **`collect()`** function in a **`try`**/**`catch`** block:

```
try {
    numbersFlow.collect { number ->
        println("Listenerreceived $number")
    }
}
catch (e: Exception) {
    println("Got an error")
}
```

Like channels, the Kotlin flows are suspending, but they are not concurrent. Flows support backpressure, although this is completely transparent to the user. To see what this means, let's create multiple subscribers for the same flow:

```
(1..4).forEach { coroutineId ->
    delay(5000)
    launch(Dispatchers.Default) {
        numbersFlow.collect { number ->
            delay(1000)
            println("Coroutine $coroutineId received
              $number")
        }
    }
}
```

Each subscriber runs in its own coroutine, with a delay of five seconds between each new subscription. This allows us to see them run concurrently.

Now, let's take a look at the output:

```
> ...
> Sending 1
> Coroutine 1 received 5
> Sending 6
> Coroutine 2 received 1
> Sending 2
> Coroutine 1 received 6
> ...
```

From this output, we can learn two important lessons:

- **Flows are cold streams**: This means for each new subscriber, the flow starts anew. In our case, each new subscriber will receive all numbers, starting from `1`.

- **Flows use backpressure**: Note that the next number is not sent until the previous number is received. This is similar to the behavior of un-buffered channels and different from buffered channels, where the producer can send numbers faster than the consumer can consume them.

Next, let's see how these two properties of flows can be altered, if necessary.

Buffering flows

In some cases, for example, when we have plenty of available memory, we aren't interested in applying backpressure on the producer right away. To do so, each consumer can specify that the flow should be *buffered* by using the `buffer()` function:

```
numbersFlow.buffer().collect { number ->
    delay(1000)
    println("Coroutine $coroutineId received
$number")
}
```

If we look at the output of the preceding code again, we'll see a dramatic change:

```
> ...
> Sending 8
> Sending 9
> Sending 10
> Coroutine 1 received 1
> Coroutine 1 received 2
> ...
```

With a buffer, the flow produces values without any backpressure from the consumer until the buffer is filled. Then, the consumer is still able to collect the values at its own pace. This behavior is similar to buffered channels, and in fact, the implementation uses a channel under the hood.

Buffering a flow is useful when it takes a considerable amount of time to process each message. Take uploading images from your phone as an example. Of course, the upload will take a different amount of time based on the size of the image. You don't want to block the user interface until the image is uploaded because that would be a bad user experience and against reactive principles.

Instead, you could define a buffer that fits into the memory, upload the images at your own pace, and block the user interface only once the buffer is full of tasks.

In the case of images, we are dealing with a series of elements we don't want to lose. So, let's consider a different example, where we could allow dropping some of the elements in our flow.

Conflating flows

Imagine we have a flow that produces changes in stock prices at a rate of ten times a second, and we have a UI that needs to display the latest stock values. To do this, we'll just use a number that goes up by 1 for every tick:

```
val stock: Flow<Int> = flow {
    var i = 0
    while (true) {
        emit(++i)
        delay(100)
    }
}
```

The UI itself, however, doesn't have to be refreshed ten times every second. Once every second is more than enough. If we simply try to use **collect()**, as in the previous example, we'll be constantly behind the producer:

```
var seconds = 0
stock.collect { number ->
    delay(1000)
    seconds++
    println("$seconds seconds -> received $number")
}
```

The preceding code outputs the following:

```
> 1 seconds -> received 1
> 2 seconds -> received 2
> 3 seconds -> received 3
> ...
```

The preceding output is incorrect. The reason for this is that we apply backpressure to the flow, slowing it down. Another option would be to buffer 10 values, as we've seen in the previous example. But since we want to refresh the UI ten times slower than the flow refreshes itself, we'll have to discard nine values out of ten. We'll leave it to the readers to try and implement that logic.

A better solution would be to *conflate* the flow. A conflated flow doesn't store all of the messages. Instead, it keeps only the most recent values. We implement this in the following code:

```
stock.conflate().collect { number ->
    delay(1000)
    seconds++
    println("$seconds seconds -> received $number")
}
```

Let's first look at the output:

```
> ...
> 4 seconds -> received 30
> 5 seconds -> received 40
> 6 seconds -> received 49
> ...
```

You can see that now the values are correct. On average, our counter is incremented ten times every second.

Now, our flow will never be suspended and the subscriber will receive only the most recent value that the flow has calculated.

Summary

This chapter was dedicated to practicing functional programming with reactive principles and learning the building blocks of functional programming in Kotlin. We also learned about the main benefits of reactive systems. For example, such systems should be responsive, resilient, elastic, and driven by messaging.

Now, you should know how to transform your data, filter your collections, and find elements within the collection that meet your criteria.

You should also better understand the difference between *cold* and *hot* streams. A cold stream, such as a *flow*, starts working only when someone subscribes to it. A new subscriber will usually receive all of the events. On the other hand, a hot stream, such as a *channel*, continuously emits events, even if nobody is listening to them. A new subscriber will receive only the events that were sent after the subscription was made.

We also discussed the concept of backpressure, which can be implemented in a flow. For example, if the consumer is not able to process all of the events, it may suspend the producer, buffer the events in the hope of catching up, or conflate the stream, handling only some of the events.

The next chapter will cover concurrent design patterns, which allow us to architect concurrent systems in a scalable, maintainable, and extensible manner, using coroutines and reactive streams as building blocks.

Questions

1. What is the difference between higher-order functions on collections and on concurrent data structures?
2. What is the difference between cold and hot streams of data?
3. When should a conflated channel or flow be used?