Chapter 5: Introducing Functional Programming

This chapter will discuss the fundamental principles of **functional programming** and how they fit into the **Kotlin** programming language.

As you'll discover, we've already touched on some of the concepts in this chapter, as it would have been hard to discuss the benefits of the language up until now without touching on functional programming concepts such as **data immutability** and **functions as values**. But as we did before, we'll look at those features from a different angle.

In this chapter, we will cover the following topics:

- Reasoning behind the functional approach
- Immutability
- Functions as values
- Expressions, not statements
- Recursion

After completing this chapter, you'll understand how the concepts of functional programming are embedded in the Kotlin language and when to use them.

Technical requirements

For this chapter, you will need to install the following:

- **IntelliJ IDEA Community Edition** (https://www.jetbrains.com/idea/download/)
- **OpenJDK 11** (or higher) (https://openjdk.java.net/install/)

You can find the code files for this chapter on **GitHub** at https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter05.

Reasoning behind the functional approach

Functional programming has been around for as long as other programming paradigms, for example, procedural and object-oriented programming. But in the past 15 years, it has gained significant momentum. The reason for this is that something else stalled: **CPU** speeds. We cannot speed up our CPUs as much as we did in the past, so we must **parallelize** our programs. And it turns out that the functional programming paradigm is exceptionally good at running parallel tasks.

The evolution of multicore processors is a fascinating topic in itself, but we'll cover it only briefly here. Workstations have had multiple processors since at least the 1980s to support the running of tasks from different users in parallel. Since workstations were massive during this era, they didn't need to worry about cramming everything into one chip. But when multiprocessors came to the consumer market around 2005, it became necessary to have one physical unit that could work in parallel. This is why we have multiple cores in one chip in our PC or laptop.

But that's not the only reason we use functional programming. Here are a few more:

- Functional programming favors pure functions, and pure functions are usually easier to reason about and test.
- Code written in a functional way is often more declarative than imperative, dealing with the *what* instead of the *how*, and this can be a benefit.

In the following sections, we'll explore the different aspects of functional programming, starting with *immutability*.

Immutability

One of the fundamental concepts of functional programming is **immutability**. This means that from the moment the function receives input to the moment the function returns output, the object doesn't change. *But how could it change?* Well, let's look at a simple example:

```
fun <T> printAndClear(list: MutableList<T>) {
    for (e in list) {
        println(e)
        list.remove(e)
    }
}
printAndClear(mutableListOf("a", "b", "c"))
```

This code would output `a` first, and then we would receive `ConcurrentModificationException`.

The reason for this is that the `for-each` loop uses an iterator (which we already discussed in the previous chapter), and by mutating the list inside the loop, we interfere with its operation. However, this raises a question:

Wouldn't it be great if we could protect ourselves from these runtime exceptions in the first place?

Let's see how *immutable collections* can help us with this.

Immutable collections

In *Chapter 1*, *Getting Started with Kotlin*, we already mentioned that collections in Kotlin are immutable by default, which is unlike many other languages.

The previous problem is caused by us not following the *single-responsibility principle*, which states that a function should do only one thing and do it well. Our function tries both to remove elements from an array and to print them at the same time.

If we change the argument from **`MutableList`** to **`List`**, we won't be able to invoke the **`remove()`** function on it, resolving our current problem. But this raises another question:

What if we need an empty list?

In this case, our function should return a new object:

```
private fun <T> printAndClear(list: MutableList<T>):
  MutableList<T> {
    for (e in list) {
        println(e)
    }
    return mutableListOf()
}
```

In general, functions that don't return any values should be avoided in functional programming, as it usually means that they have a side effect.

However, it's not enough that the collection *type* is immutable. The *content* of the collection should be immutable as well. To understand this better, let's look at the following simple class:

```
data class Player(var score: Int)
```

You can see that this class has only one variable: **`score`**.

Next, we'll create a single instance of the **`data class`** and put it in an immutable collection:

```
val scores = listOf(Player(0))
```

We could put multiple instances of this class inside the collection, but to illustrate our point, only one is needed.

Next, let's introduce the concept of *threads*.

The problem with shared mutable state

If you aren't familiar with **threads**, don't worry, we'll discuss them in detail in *Chapter 6*, *Threads and Coroutines*. All you need to know for now is that threads allow the code to run *concurrently*. When using concurrent code and code that utilizes multiple CPUs, functional programming really helps. You may find that any other example that doesn't involve concurrency at all may seem rather convoluted or artificial.

For now, let's create a list that contains two threads:

```
val threads = List(2) {
        thread {
            for (i in 1..1000) {
                scores[0].score++
            }
        }
    }
```

As you can see, each thread increments `score` by `1000` in total, using a regular `for` loop.

We wait for the threads to complete by using `join()`, and then we check the counter value:

```
for (t in threads) {
    t.join()
}
println(scores[0].score) // Less than 2000 for sure
```

If you run the code yourself, the value will be anything under `2000`.

This is a classic case of a *race condition* for mutable variables. The number you'll get will be different every time you run this code. The reason for this may be familiar to you if you have encountered concurrency pre-

viously. And, it has nothing to do with threads not completing their work, by the way. You can make sure of this by adding a print message after the loop:

```
thread {
    for (i in 1..1000) {
        scores[0].score = scores[0].score + 1
    }
    println("Done")
}
```

This also isn't the fault of using the increment (**++**) operator. As you can see, we used the long notation to increment the value, but if you run it again as many times as possible, you would still get the wrong results.

The reason for this behavior is that the addition operation and the assignment operation are not *atomic*. Two threads may override the addition operations of each other, resulting in the number not being incremented enough times.

Here, we used an extreme example of a collection that contains exactly one element. In the real world, the collections you will be working with will usually contain multiple elements. For example, you would track scores for multiple players, or even maintain a ranking system for thousands of players simultaneously. This would complicate the example even further.

What you need to remember is the following: even if a collection is immutable, it may still *contain* mutable objects inside. Mutable objects are not thread-safe.

Next, let's look at *tuples*, which are an example of immutable objects.

Tuples

In functional programming, a **tuple** is a piece of data that cannot be changed after it is created. One of the most basic tuples in Kotlin is **`pair`**:

```
val pair = "a" to 1
```

`pair` contains two properties – called **`first`** and **`second`** – and is immutable:

```
pair.first = "b" // Doesn't work
pair.second = 2  // Still doesn't
```

We can *destructure* **`pair`** into two separate values using a *destructure declaration*:

```
val (key, value) = pair
println("$key => $value")
```

When iterating over a map, we also work with another type of tuple: **`Map.Entry`**:

```
for (p in mapOf(1 to "Sunday", 2 to "Monday")) {
   println("${p.key} ${p.value}")
}
```

This tuple already has **`key`** and **`value`** members, instead of **`first`** and **`second`**.

In addition to **`pair`**, there is a **`Triple`** tuple that also contains a **`third`** value:

```
val firstEdition = Triple("Design Patterns with
Kotlin",   310, 2018)
```

In general, **`data`** classes are usually a good implementation for tuples because they provide clear naming. If you look at the preceding example, it's not immediately obvious that the **`310`** value represents the number of pages.

However, as we saw in the previous section, not every **`data`** class is a proper tuple. You need to make sure that all of its members are *values* and not *variables*. You also need to check whether any nested collections or classes it has are immutable as well.

Now, let's discuss another important topic in functional programming: functions as a first-class citizen of the language.

Functions as values

We already covered some of the functional capabilities of Kotlin in the chapters dedicated to design patterns. The **Strategy** and **Command** design patterns are only two examples that rely heavily on the ability to accept functions as arguments, return functions, store functions as values, or put functions inside of collections. In this section, we'll cover some other aspects of functional programming in Kotlin, such as *function purity* and *currying*.

Learning about higher-order functions

As we discussed previously, in Kotlin, it's possible for a function to return another function. Let's look at the following simple function to understand this syntax in depth:

```
fun generateMultiply(): (Int) -> Int {
    return fun(x: Int): Int {
        return x * 2
    }
}
```

Here, our **`generateMultiply`** function returns another function that doesn't have a name. Functions without a name are called **anonymous functions**.

We could also rewrite the preceding code using shorter syntax:

```
fun generateMultiply(): (Int) -> Int {
    return { x: Int ->
        x * 2
    }
```

```
}
```

If a function without a name uses short syntax, it's called a **lambda function**.

Next, let's look at the signature of the return type:

```
(Int) -> Int
```

From that signature, we know that the function that we return will accept a single integer as input and produce an integer as output.

If a function doesn't accept any arguments, we denote that using empty round brackets:

```
() -> Int
```

If a function doesn't return anything, we use the **`Unit`** type to specify that:

```
(Int) -> Unit
```

Functions in Kotlin can be assigned to a variable or value to be invoked later on:

```
val multiplyFunction = generateMultiply()
...
println(multiplyFunction(3, 4))
```

The function assigned to a variable is usually called a **literal function**.

We applied this in *Chapter 4*, *Getting Familiar with Behavioral Patterns*, when discussing the Strategy design pattern.

It's also possible to specify a function as a parameter:

```
fun mathInvoker(x: Int, y: Int, mathFunction: (Int,
Int) ->   Int) {
    println(mathFunction(x, y))
}
mathInvoker(5, 6, multiplyFunction)
```

If a function is the last parameter, it can also be supplied in an ad hoc fashion, outside of the brackets:

```
mathInvoker(7, 8) { x, y ->
   x * y
}
```

This syntax is also called **trailing lambda** or **call suffix**. We saw an example of this in *Chapter 4*, *Getting Familiar with Behavioral Patterns*, when discussing the Interpreter design pattern.

Now that we've covered the basic syntax of functions, let's see how they can be used.

Higher-order functions in a standard library

When working with Kotlin, something you will be doing on a daily basis is working with *collections*. As we mentioned briefly in *Chapter 1*, *Getting Started with Kotlin*, collections have support for higher-order functions.

For example, in the previous chapters, to print elements of a collection one by one, we used a boring `for-each` loop:

```
val dwarfs = listOf("Dwalin", "Balin", "Kili",
"Fili",   "Dori", "Nori", "Ori", "Oin", "Gloin",
"Bifur", "Bofur",   "Bombur", "Thorin")
for (d in dwarfs) {
    println(d)
}
```

Many of you probably groaned at seeing this. But I hope you didn't stop reading the book altogether. Of course, there is also another way to achieve the same goal that is common in many programming languages: a `forEach` function:

```
dwarfs.forEach { d ->
```

```
    println(d)
}
```

This function is one of the most basic examples of a higher-order function. Let's see how it's declared:

```
fun <T> Iterable<T>.forEach(action: (T) -> Unit)
```

Here, **`action`** is a function that receives an element of a collection and doesn't return anything. This function presents an opportunity to discuss another aspect of Kotlin: the **`it`** notation.

The it notation

It is very common in functional programming to keep your functions small and simple. The simpler the function, the easier it is to understand, and the more chances it has to be reused in other places. And the aim *of reusing* code is one of the basic Kotlin principles.

Notice that in the preceding example, we didn't specify the type of the **`d`** variable. We could do this using the same colon notation we have used elsewhere:

```
dwarfs.forEach { d: String ->
    println(d)
}
```

However, usually, we don't need to do this because the compiler can figure this out from the generic types that we use. After all, **`dwarfs`** is of the **`List<String>`** type, so **`d`** is of the **`String`** type as well.

The type of the argument is not the only part that we can omit when writing short lambdas like this one. If a lambda takes a single argument, we can use the implicit name for it, which in this case, is **`it`**:

```
dwarfs.forEach {
    println(it)
}
```

In cases where we need to invoke a single function to a single parameter, we could also use a *function reference*. We saw an example of this in *Chapter 4*, *Getting Familiar with Behavioral Patterns*, when discussing the Strategy design pattern:

```
dwarfs.forEach(::println)
```

We'll use the shortest notation in most of the following examples. It is advised to use the longer syntax for cases such as *one lambda nested in another*. In those cases, giving proper names for the parameters is more important than conciseness.

Closures

In the object-oriented paradigm, state is always stored within objects. But in functional programming, this isn't necessarily the case. Let's look at the following function as an example:

```
fun counter(): () -> Int {
    var i = 0
    return { i++ }
}
```

The preceding example is clearly a higher-order function, as you can see by its **`return`** type. It returns a function with zero arguments that produces an integer.

Let's store it in a variable, in the way we've already learned, and invoke it multiple times:

```
val next = counter()
println(next())
println(next())
println(next())
```

As you can see, the function is able to keep a state, in this case, the value of a counter, even though it is not part of an object.

This is called a **closure**. The lambda has access to all of the local variables of the function that wraps it, and those local variables persist, as long as the reference to the lambda is kept.

The use of closures is another tool in the functional programming toolbox that reduces the need to define lots of classes that simply wrap a single function with some state.

Pure functions

A **pure function** is a function without any side effects. A **side effect** can be considered anything that accesses or changes the external state. The external state can be a non-local variable (where a variable from a closure is still considered to be non-local) or any kind of IO (that is, reading or writing to a file or using any kind of network capabilities).

IMPORTANT NOTE:

For those not familiar with the term, **IO** *stands for* **Input/Output***, and this covers any kind of interaction that is external to our program, such as writing to files or reading from a network.*

For example, the lambda we just discussed in the *Closures* section is not considered *pure* because it can return different output for the same input when it is invoked multiple times.

Impure functions are hard to test and to reason about in general, as the result they return may depend on the order of execution or on factors that we can't control (such as network issues).

One thing to remember is that logging or even printing to a console still involves IO and is subject to the same set of problems.

Let's look at the following simple function:

```
fun sayHello() = println("Hello")
```

So, in this case, how do you ensure that Hello is printed? The task is not as simple as it seems, as we'll need some way to capture the standard output – that is, the same console where we usually see stuff printed.

We'll compare it to the following function:

```
fun hello() = "Hello"
```

The following function doesn't have any side effects. That makes it a lot easier to test:

```
fun testHello(): Boolean {
    return "Hello" == hello()
}
```

The **`hello()`** function may look a bit meaningless, but that's actually one of the properties of pure functions. Their invocation could be replaced by their result if we knew it ahead of time. This is often called **referential transparency**.

As we mentioned earlier, not every function written in Kotlin is a pure function:

```
fun <T> removeFirst(list: MutableList<T>): T {
    return list.removeAt(0)
}
```

If we call the function twice on the same list, it will return different results:

```
val list = mutableListOf(1, 2, 3)
println(removeFirst(list)) // Prints 1
println(removeFirst(list)) // Prints 2
```

Compare the preceding function to this one:

```
fun <T> withoutFirst(list: List<T>): T {
    return ArrayList(list).removeAt(0)
}
```

Now, our function is totally predictable, no matter how many times we invoke it:

```
val list = mutableListOf(1, 2, 3)
println(withoutFirst(list)) // It's 1
println(withoutFirst(list)) // Still 1
```

As you can see, in this instance, we used an immutable interface, **`List<T>`**, which helps us by preventing the possibility of mutating our input. When combined with the immutable values we discussed in the previous section, pure functions allow easier testing by providing predictable results and the parallelization of our algorithms.

A system that utilizes pure functions is easier to reason about because it doesn't rely on any external factors – what you see is what you get.

Currying

Currying is a way to translate a function that takes a number of arguments into a chain of functions, where each function takes a single argument. This may sound confusing, so let's look at a simple example:

```
fun subtract(x: Int, y: Int): Int {
    return x - y
}
println(subtract(50, 8))
```

This is a function that takes two arguments as an input and returns the difference between them. However, some languages allow us to invoke this function with the following syntax:

```
subtract(50)(8)
```

This is what currying looks like. Currying allows us to take a function with multiple arguments (in our case, two) and convert this function into a set of functions, where each one takes only a single argument.

Let's examine how this can be achieved in Kotlin. We've already seen how we can return a function from another function:

```
fun subtract(x: Int): (Int) -> Int {
    return fun(y: Int): Int {
        return x - y
    }
}
```

Here is the shorter form of the preceding code:

```
fun subtract(x: Int) = fun(y: Int): Int {
    return x - y
}
```

In the preceding example, we use single-expression syntax to return an anonymous function without the need to declare the **`return`** type or use the **`return`** keyword.

And here it is in an even shorter form:

```
fun subtract(x: Int) = {y: Int -> x - y}
```

Now, an anonymous function is translated to a lambda, with the **`return`** type of the lambda inferred as well.

Although not very useful by itself, it's still an interesting concept to grasp. And if you're a **JavaScript** developer looking for a new job, make sure you understand it fully, since it's asked about in nearly every interview.

One real-world scenario where you might want to use currying is *logging*. A **`log`** function usually looks something like this:

```
enum class LogLevel {
    ERROR, WARNING, INFO
}
fun log(level: LogLevel, message: String) =
    println("$level: $message")
```

We could fix the log level by storing the function in a variable:

```
val errorLog = fun(message: String) {
    log(LogLevel.ERROR, message)
}
```

Notice that the **`errorLog`** function is easier to use than the regular **`log`** function because it accepts one argument instead of two. However, this raises a question:

What if we don't want to create all of the possible loggers ahead of time?

In this case, we can use currying. The *curried* version of this code would look like this:

```
fun createLogger(level: LogLevel): (String) -> Unit {
    return { message: String ->
        log(level, message)
    }
}
```

Now, it's up to whoever uses our code to create the logger they want:

```
val infoLogger = createLogger(LogLevel.INFO)
infoLogger("Log something")
```

This, in fact, is very similar to the Factory design pattern we covered in *Chapter 2*, *Working with Creational Patterns*. Again, the power of a modern language decreases the number of custom classes we need to implement to achieve the same behavior.

Next, let's talk about another powerful technique that can save us from having to do the same computation over and over again.

Memoization

If our function always returns the same output for the same input, we can easily map its input to the output, caching the results in the process. This technique is called **memoization**.

A common task when developing different types of systems or solving problems is finding a way to avoid repeating the same computation multiple times. Let's assume we receive multiple lists of integers, and for each list, we would like to print its sum:

```
val input = listOf(
    setOf(1, 2, 3),
    setOf(3, 1, 2),
    setOf(2, 3, 1),
    setOf(4, 5, 6)
)
```

Looking at the input, you can see that the first three sets are in fact equal – the difference is only in the order of the elements, so calculating the sum three times would be wasteful.

The sum calculation can be easily described as a pure function:

```
fun sum(numbers: Set<Int>): Double {
    return numbers.sumByDouble { it.toDouble() }
}
```

This function does not depend on any external state and doesn't change the external state in any way. So, it is safe for the same input to replace the call to this function with the value it had returned previously.

We could store the results of a previous computation for the same set in a mutable map:

```
val resultsCache = mutableMapOf<Set<Int>, Double>()
```

To avoid creating too many classes, we could use a higher-order function that would wrap the result in the cache that we created earlier:

```
fun summarizer(): (Set<Int>) -> Double {
    val resultsCache = mutableMapOf<Set<Int>, Double>
()
    return { numbers: Set<Int> ->
```

```
            resultsCache.computeIfAbsent(numbers, ::sum)
        }
    }
```

Here, we use a method reference operator (`::`) to tell **`computeIfAbsent`** to use the **`sum()`** method in the event where the input hasn't been cached yet.

Note that **`sum()`** is a pure function, while **`summarize()`** is not. The latter will behave differently for the same input. But that's exactly what we want in this case.

Running the following code on the preceding input will invoke the sum function only twice:

```
val summarizer = summarizer()
input.forEach {
    println(summarizer(it))
}
```

The combination of immutable objects, pure functions, and closures provides us with a powerful tool for performance optimization. Just remember: nothing is free. We trade one resource, CPU time, for another resource, which is memory. And it's up to you to decide which resource is more expensive in each case.

Using expressions instead of statements

A **statement** is a block of code that *doesn't return* anything. An **expression**, on the other hand, *returns a new value*. Since statements produce no results, the only way for them to be useful is to mutate the state, whether that's changing a variable, changing a data structure, or performing some kind of IO.

Functional programming tries to avoid mutating the state as much as possible. Theoretically, the more we rely on expressions, the more our func-

tions will be pure, with all the benefits of functional purity.

We've used the **`if`** expression many times already, so one of its benefits should be clear: it's less verbose and, for that reason, less error-prone than the **`if`** statement from other languages.

Pattern matching

The concept of **pattern matching** will seem like **`switch`**/**`case`** on steroids. We've already seen how the **`when`** expression can be used, which we explored in *Chapter 1*, *Getting Started with Kotlin*, so let's briefly discuss why this concept is important for the functional paradigm.

You may know that in Java, **`switch`** accepts only some primitive types, strings, or enums.

Consider the following code, which is usually used to demonstrate how polymorphism is implemented in the language:

```
class Cat : Animal {
    fun purr(): String {
        return "Purr-purr";
    }
}
class Dog : Animal {
    fun bark(): String {
        return "Bark-bark";
    }
}
interface Animal
```

If we were to decide which of the functions to call, we would need to write code akin to the following:

```
fun getSound(animal: Animal): String {
    var sound: String? = null;
```

```
    if (animal is Cat) {
        sound = (animal as Cat).purr();
    }
    else if (animal is Dog) {
        sound = (animal as Dog).bark();
    }
    if (sound == null) {
        throw RuntimeException();
    }
    return sound;
}
```

This code attempts to figure out at runtime what methods the **`getSound`** class implements.

This method could be shortened by introducing multiple returns, but in real projects, multiple returns are usually a bad practice.

Since we don't have a **`switch`** statement for classes, we need to use an **`if`** statement instead.

Now, let's compare the preceding code with the following Kotlin code:

```
fun getSound(animal: Animal) = when(animal) {
    is Cat -> animal.purr()
    is Dog -> animal.bark()
    else -> throw RuntimeException("Unknown animal")
}
```

Since **`when`** is an expression, we avoided declaring the intermediate variable we previously had altogether. In addition, the code that uses pattern matching doesn't need any type checks and casts.

Now we've learned how to replace imperative **`if`** statements with much more functional **`when`** expressions, let's see how we can replace imperative loops in our code by using *recursion*.

Recursion

Recursion is a function invoking itself with new arguments. Many well-known algorithms, such as **Depth First Search**, rely on recursion.

Here is an example of a very inefficient function that uses recursion to calculate the sum of all the numbers in a given list:

```
fun sumRec(i: Int, sum: Long, numbers: List<Int>):
Long {
    return if (i == numbers.size) {
        return sum
    } else {
        sumRec(i+1, numbers[i] + sum, numbers)
    }
}
```

We often try to avoid recursion due to the stack overflow errors that we may receive if our call stack is too deep. You can call this function with a list that contains a million numbers to demonstrate this:

```
val numbers = List(1_000_000) {it}
println(sumRec(0,  numbers))
// Crashed pretty soon, around 7K
```

However, Kotlin supports an optimization called **tail recursion**. One of the great benefits of tail recursion is that it avoids the dreaded stack over-flow exception. If there is only a single recursive call in our function, we can use that optimization.

Let's rewrite our recursive function using a new keyword, **`tailrec`**, to avoid this problem:

```
tailrec fun sumRec(i: Int, sum: Long, numbers:
List<Int>):
  Long {
```

```
    return if (i == numbers.size) {
        return sum
    } else {
        sumRec(i+1, numbers[i] + sum, numbers)
    }
}
```

Now, the compiler will optimize our call and avoid the exception completely.

However, this optimization doesn't work if you have multiple recursive calls, such as in the **Merge Sort** algorithm.

Let's examine the following function, which is the *sort* part of the Merge Sort algorithm:

```
tailrec fun mergeSort(numbers: List<Int>): List<Int>
{
    return when {
        numbers.size <= 1 -> numbers
        numbers.size == 2 -> {
            return if (numbers[0] < numbers[1]) {
                numbers
            } else {
                listOf(numbers[1], numbers[0])
            }
        }
        else -> {
            val left = mergeSort(numbers.slice
              (0..numbers.size / 2))
            val right = mergeSort(numbers.slice
              (numbers.size / 2 + 1 until
numbers.size))
            return merge(left, right)
        }
    }
```

```
}
```

Notice that there are two recursive calls instead of one. The Kotlin compiler will then issue the following warning:

```
> "A function is marked as tail-recursive but no tail
calls are found"
```

Summary

You should now have a better understanding of functional programming and its benefits, as well as how Kotlin approaches this topic. We've discussed the concepts of *immutability* and *pure functions*, and how combining these results in more testable code that is easier to maintain.

We discussed how Kotlin supports *closures*, which allow a function to access the variables of the function that wraps it and effectively store the state between executions. This enables techniques such as *currying* and *memoization* that allow us to fix some of the function arguments (by acting as defaults) and remember the value returned from a function in order to avoid recalculating it.

We learned that Kotlin uses the **`tailrec`** keyword to allow the compiler to optimize *tail recursion*. We also looked at *higher-order functions*, *expressions versus statements*, and *pattern matching*. All of these concepts allow us to write code that is easier to test and has less risk of concurrency bugs.

In the next chapter, we'll put this knowledge to practical use and discover how **reactive programming** builds upon functional programming to create scalable and resilient systems.

Questions

1. What are higher-order functions?
2. What is the **`tailrec`** keyword in Kotlin?

3. What are pure functions?