Marcin Moskała

Functional Kotlin

with exercises

Functional Kotlin

Marcin Moskała

This book is available at http://leanpub.com/kotlin_functional

This version was published on 2024-12-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

For my friends, Olga and Marek Kamiński

Contents

Introduction

At the beginning of the 21st century, Java mostly dominated commercial programming. Therefore, the object-oriented paradigm ruled in our discipline. Many thought that the holy war between the two biggest paradigms - object-oriented and functional programming - was resolved, but then Scala showed us that they had never needed to fight with each other in the first place. A programming language can have both functional and object-oriented features that complement each other. This has started a renaissance in functional programming, as many functional programming features have been introduced in many popular languages. Nowadays, most mainstream languages support both functional and object-oriented features, but the problem is that people often still don't know how to use them effectively and efficiently.

This book is about functional programming features in Kotlin. It first covers the essentials, and then it builds on them: it presents important and practical topics like collection processing, function references, scope functions, DSL usage and creation, and context receivers.

Who is this book for?

This book is dedicated to developers with basic experience in using Kotlin or who have read my other book *Kotlin Essentials*.

What will be covered?

This book focuses on Kotlin's functional features, including:

- function types,
- anonymous functions,
- lambda expressions,
- function references,
- functional interfaces,
- collection processing functions,
- sequences,
- DSL usage and creation,
- scope functions,
- context receivers.

This book is based on a workshop I conducted.

The Kotlin for Developers series

This book is a part of a series of books called *Kotlin for Developers*, which includes the following books:

- *Kotlin Essentials*, which covers all the basic Kotlin features.
- *Functional Kotlin*, which is dedicated to functional Kotlin features, including function types, lambda expressions, collection processing, DSLs, and scope functions.
- *Kotlin Coroutines: Deep Dive*, which covers all the Kotlin Coroutines features, including how to use and test them, using flow, the best practices, and the most common mistakes.
- *Advanced Kotlin*, which is dedicated to advanced Kotlin features, including generic variance modifiers, delegation, multiplatform programming, annotation processing, KSP and compiler plugins.
- *Effective Kotlin: Best Practices*, which is dedicated to the best practices of Kotlin programming.

In this book, I assume that a reader has the knowledge presented in *Kotlin Essentials*, which I reference explicitly. However, readers with experience in Kotlin, or at least in Java, should be perfectly fine starting their adventure from this book.

Conventions

When I mean a concrete element from code, I will use code font. To name a concept, I will use uppercase. To reference an arbitrary element of some type, I will use lowercase. For example:

- `List` is a type or an interface, so it is printed in code font (as in “Function needs to return `List`”),
- List represents a concept, so it starts with uppercase (as in “This explains the essential difference between List and Set”),
- a list is an instance, which is why it is lowercase (as in “the `list` variable holds a list”).

In this book, I decided to use a dash between “if”, “when”, “try”, “while”, and “for”, and the word describing it, like “condition”, “loop”, “statement”, or “expression”. I do this to improve readability. So, I will write “if-condition” instead of “if condition”, or “while-loop” instead of “while loop”. “if” and “when” are conditions, “while” and “for” are loops. All of them can be used as statements, while “if”, “when” and “try” can also be used as expressions. I also decided not to use a dash after “if-else”, “if-else-if”, or “try-catch” and their descriptor, like in an “if-else statement”.

Code conventions

Most of the presented snippets are executable, so if you copy-paste them to a Kotlin file, you should be able to execute them. The source code of all the snippets is published in the following repository:

https://github.com/MarcinMoskala/functional_kotlin_sources

I often use comments to explain what will be printed by a particular line.

```
fun main() {
    val cheer: () -> Unit = fun() {
        println("Hello")
    }
    cheer.invoke() // Hello
    cheer() // Hello
}
```

Sometimes, I also move all such comments to the end of a snippet.

```
fun main() {
    val cheer: () -> Unit = fun() {
        println("Hello")
    }
    cheer.invoke()
    cheer()
}
// Hello
// Hello
```

Sometimes, some parts of code or results are shortened with ... in a comment. In such cases, you can read it as "there should be more here, but the author decided to omit it".

```
adapter.setOnSwipeListener { /*...*/ }
```

In some snippets, you might notice strange formatting. This is because the line length in this book is only 67 characters, so I adjusted the formatting to fit the page width.

Exercises and solutions

At the end of most chapters, you will find exercises. They are designed to help you understand the material better. Starting code and unit tests for most of those exercises can be found in the MarcinMoskala/kotlin-exercises project on GitHub. You can clone this project and solve these exercises locally. Solutions can be found at the end of the book.

Suggestions

If you have any suggestions or corrections regarding this book, send them to contact@kt.academy

Acknowledgments

This book would not be so good without the reviewers' great suggestions and comments. I would like to thank all of them. Here is the whole list of reviewers, starting from those who influenced it most.

Owen Griffiths has been developing software since the mid 1990s and remembers the productivity of languages such as Clipper and Borland Delphi. Since 2001, He moved to Web, Server based Java and the Open Source revolution. With many years of commercial Java experience, He picked up on Kotlin in early 2015. After taking detours into Clojure and Scala, like Goldilocks, He thinks Kotlin is just right and tastes the best. Owen enthusiastically helps Kotlin developers continue to succeed.

Endre Deak is a software architect building AI infrastructure at Disco, a market leading legal tech company. He has 15 years of experience building complex, scalable systems, and he thinks Kotlin is one of the best programming languages ever created.

Piotr Prus is an Android developer and mobile technology enthusiast since the first Maemo and Android systems. Loves clean simple designs and readable code. Shares knowledge with the community by writing articles and speaking at conferences. Currently, KMMing and Composing all the things.

Jacek Kotorowicz is an Android developer based in Lublin, graduated from UMCS. Wrote his Master's thesis in C++ in Vim and LaTeX. Later, fell in love-hate relationship with JVM languages and Android platform. First used Kotlin (or at least tried to do so) in versions before 1.0. Still learning how NOT to be a perfectionist and how to find time for learning and hobbies.

Anna Zharkova is a Lead Mobile developer with more than 8 years of experience. Kotlin GDE. Develop both native (iOS - Swift/Objective-c, Android - Kotlin/Java) and cross platform (Xamarin, Kotlin Multiplatform) applications. Design architectural solution in mobile projects. Leading mobile team, mentorship. Public speaker on conferences and meetups (Droidcon, Android Worldwide, SwiftHero, Mobius). Tutor in Otus. Writing articles about mobile development (especially KMM and Swift)

Norbert Kiesel is a backend Kotlin and Java developer and architect who started using Kotlin 5 years ago as a "better Java" and never looked back. His initiative made Kotlin a recommended language in his company, and he helped its adoption by running a Kotlin user group.

Jana Jarolimova is an Android developer at Avast. She started her career teaching Java classes at Prague City University, before moving on to mobile development, which inevitably led to Kotlin and her love thereof.

Aasif Sheikh and **Sunny Aditya**.

I would also like to thank **Michael Timberlake**, our language reviewer, for his excellent corrections to the whole book.

Introduction to functional programming with Kotlin

What is functional programming? This is not an easy question to answer. There is a popular saying that if you ask two developers about what functional programming is, you will get at least three answers. I don't think there is a single definition that everyone will agree on. However, there are several concepts that are often associated with functional programming, including:

- treating functions as objects,
- higher-order functions,
- data immutability,
- using statements as expressions[1],
- lazy evaluation,
- pattern-matching,
- recursive function calls.

There is also a way of thinking that stands behind functional programming. In the object-oriented approach, we see the world as a set of objects; in contrast, in a functional approach, we see the world as a set of functions. Think of a bedroom: is it a room with a bed, a nightstand, a bedside lamp, etc., or is it just a place where we sleep?

Is Kotlin a functional language? One programmer will say "yes", whereas another might say "no". I am certain of two things:

1. Kotlin has powerful support for many features that are typical of functional programming languages.
2. Kotlin is not a **purely** functional language.

Kotlin has powerful support for many features that are typical to functional programming languages. Let's consider our previous list of concepts that are typical of functional programming and let's look at how Kotlin supports them.

[1]In his presentation at Kotlin Dev Days 2022, Andrey Breslav claimed that he had asked Martin Odersky (the creator of Scala) about what makes a language functional, and he answered that every statement is an expression. A statement is a single command that a programmer expresses in a programming language, typically a single line of code. An expression is something that returns a value.

Feature	Support
Treating functions as objects	Function types, lambda expressions, function references
Higher-order functions	Full support
Data immutability	`val` support, default collections are read-only, `copy` in data classes
Expressions	if-else, try-catch, when statements are expressions
lazy evaluation	`lazy` delegate.
Pattern-matching	`when` together with smart casting
Recursive function calls	`tailrec` modifier

Kotlin was designed to support functional programming (FP), but not as much as Haskell or Scala. However, there are many functional programming features that it does not support. We might mention currying, partial function application, etc. Kotlin's creators wanted to take the best features from FP that they believed are best for practical applications without taking features that might make programs harder to understand or modify. Did they do a good job? Who knows, but it seems that many developers like the final result.

Some developers miss some FP features that are not supported by Kotlin, so they have implemented external libraries like Arrow to make at least some of them available. This book concentrates on the functional features built into Kotlin, but the last chapter presents an overview of the essential Arrow features. It was written by Alejandro Serrano Mena, Simon Vergauwen, and Raúl Raja Martínez, who are Arrow maintainers and co-creators.

Kotlin is not a purely functional language. It has support for features that are typical of an Object-Oriented (OO) approach, and it is often used as a Java successor. Kotlin tries to take the best from both OOP and FP.

If you are reading this book, I assume that you already know the basic Kotlin features. No matter if you use Kotlin in your daily work as a developer, just learned the Kotlin basis, or finished my previous book *Kotlin Essentials*. I assume that you know what a data class is, what the difference is between `val` and `var`, how different statements can be used as expressions, etc. In this book, I will focus on what I believe is the essence of functional programming: using functions as objects. So, we will learn about function types, anonymous functions, lambda expressions, function references, etc. Then, I would like to focus on the most important practical application of using functions as objects: functional-style collections processing. Then, we will look at two other applications: type-safe DSL builders and scope functions. In my opinion, these are the most important aspects of Kotlin's support for and application of functional programming.

For now, let's focus on what I find essential: using functions as objects in Kotlin. Why do we need this?

Why do we need to use functions as objects?

To understand why we need to use functions as objects, take a look at these functions:

```
fun sum(a: Int, b: Int): Int {
    var sum = 0
    for (i in a..b) {
        sum += i
    }
    return sum
}

fun product(a: Int, b: Int): Int {
    var product = 1
    for (i in a..b) {
        product *= i
    }
    return product
}

fun main() {
    sum(5, 10) // 45
    product(5, 10) // 151200
}
```

The first one calculates the sum of all the numbers in a range; the second one calculates a product. The bodies of these two functions are nearly identical: the only difference is in the initial value and the operation. Yet, without support for treating functions as objects, extracting the common parts would not make any sense. Just think about how it would look in Java before version 8. In such a case, we had to create classes to represent operations and interfaces to specify what you expect... It would be absurd.

```
// Java 7
public class RangeOperations {

    public static int sum(int a, int b) {
        return fold(a, b, 0, new Operation() {
            @Override
            public int invoke(int left, int right) {
                return a + b;
            }
```

```
        });
    }

    public static int product(int a, int b) {
        return fold(a, b, 1, new Operation() {
            @Override
            public int invoke(int left, int right) {
                return a * b;
            }
        });
    }

    private interface Operation {
        int invoke(int left, int right);
    }

    private static int fold(
            int a,
            int b,
            int initial,
            Operation operation
    ) {
        int acc = initial;
        for (int i = a; i <= b; i++) {
            acc = operation.invoke(acc, i);
        }
        return acc;
    }
}
```

This is where functional programming features come to the rescue. They allow us to easily create a function and pass it as an object. To create a function, we can use a lambda expression. To express what kind of function a parameter expects, we can use a function type. This is what our code might look like if we use lambda expressions and function types:

```kotlin
fun sum(a: Int, b: Int) = fold(a, b, 0, { acc, i -> acc + i })

fun product(a: Int, b: Int) = fold(a, b, 1, { acc, i -> acc * i })

fun fold(
    a: Int,
    b: Int,
    initial: Int,
    operation: (Int, Int) -> Int
): Int {
    var acc = initial
    for (i in a..b) {
        acc = operation(acc, i)
    }
    return acc
}
```

Functional programmers noticed long ago that many repetitive code patterns could be extracted into separated functions with the help of functional programming features. `fold` is a great example. Its more universal form was defined years ago and nowadays it is a part of the Kotlin Standard Library (stdlib). This is why we can define our `sum` and `product` in the following way:

```kotlin
fun sum(a: Int, b: Int) = (a..b).fold(0) { acc, i -> acc + i }

fun product(a: Int, b: Int) = (a..b).fold(1) { acc, i -> acc * i }
```

However, if we use function references, they can also be defined in the following way:

```kotlin
fun sum(a: Int, b: Int) = (a..b).fold(0, Int::plus)

fun product(a: Int, b: Int) = (a..b).fold(1, Int::times)
```

If you are acquainted with the collection processing functions well, you know that calculating the sum of all the numbers in an iterable can be done with the `sum` method:

```kotlin
fun sum(a: Int, b: Int) = (a..b).sum()

fun product(a: Int, b: Int) = (a..b).fold(1, Int::times)
```

In this book, we will learn this and much more. We will learn about advanced collection processing functions and optimizations. So processing like this will be a piece of cake for you:

```
fun produceDepartmentSummaries(
    employees: List<Employee>
): List<DepartmentSummary> = employees
    .filter { it.age > 30 }
    .groupBy { it.department }
    .map { (department, employees) ->
        DepartmentSummary(
            department = department,
            topEmployees = employees.asSequence()
                .sortedByDescending { it.salary }
                .take(3)
                .map { EmployeeInfo(it.name, it.salary) }
                .toList()
        )
    }
```

You will also learn not only how to use, but also how to define your own DSL builders.

```
val html = html {
    head {
        title { +"HTML encoding with Kotlin" }
    }
    body {
        div {
            a("https://kotlinlang.org") {
                target = ATarget.blank
                +"Main site"
            }
        }
        +"Some content"
    }
}
```

You will learn how to use scope functions and function references to make your code more functional and expressive:

```kotlin
class UserCreationService(
    private val userRepository: UserRepository,
    private val userDtoFactory: UserDtoFactory,
) {
    fun addUser(request: UserCreationRequest): User? =
        request.let(userDtoFactory::fromRequest)
            .also(userRepository::addUser)
            ?.let(userDtoFactory::toUser)
}
```

You will also learn how to use contracts, context receivers, and other advanced features of Kotlin. Finally, you will learn about the Arrow library, that provides a lot of functional programming features and tools. Does it sound interesting? So, let's get started.

Function types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Defining function types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Using function types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Named parameters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Type aliases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

A function type is an interface

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Anonymous functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Lambda expressions

Lambda expressions are a shorter alternative to anonymous functions. They are also used to define objects that represent functions. Both notations compile to the same result, but lambda expressions support more features (most of which will be presented in this chapter). In the end, lambda expressions are the most popular and idiomatic approach to create objects that represent functions, therefore understanding them is essential for using Kotlin's functional programming features.

> An expression used to create an object representing a function is called a *function literal*, so both lambda expressions and anonymous functions are function literals.

Tricky braces

Lambda expressions are defined in braces (curly brackets). What is more, even just empty braces define a lambda expression.

```
fun main() {
    val f: () -> Unit = {}
    f()
    // or f.invoke()
}
```

But be careful because all braces that are not part of a Kotlin structure are lambda expressions (we can call them orphaned lambda expressions). This can lead to a lot of problems. Take a look at the following example: What does the following `main` function print?

```
fun main() {
    {
        println("AAA")
    }
}
```

The answer is nothing. It creates a lambda expression that is never invoked. Another question: What does the following `produce` function return?

```
fun produce() = { 42 }

fun main() {
    println(produce()) // ???
}
```

Counterintuitively, it is not `42`. Braces are not a part of single-expression function notation. The `produce` function returns a lambda expression of type `() -> Int`, so the above code on JVM should print something like `Function0<java.lang.Integer>`, or just `() -> Int`. To fix this code, we should either call the produced function or remove the braces inside the single-expression function definition.

```
fun produceFun() = { 42 }
fun produceNum() = 42

fun main() {
    val f = produceFun()
    println(f()) // 42
    println(produceFun()()) // 42
    println(produceFun().invoke()) // 42

    println(produceNum()) // 42
}
```

Parameters

If a lambda expression has parameters, we need to separate the content of the braces with an arrow `->`. Before the arrow, we specify parameter names and types, separated by commas. After the arrow, we specify the function body.

```
fun main() {
    val printTimes = { text: String, times: Int ->
        for (i in 1..times) {
            print(text)
        }
    } // the type is (text: String, times: Int) -> Unit
    printTimes("Na", 7) // NaNaNaNaNaNaNa
    printTimes.invoke("Batman", 2) // BatmanBatman
}
```

Most often, we define lambda expressions as arguments to some functions. Regular functions need to define their parameter types, based on which lambda expression parameter types can be inferred.

```
fun setOnClickListener(listener: (View, Click) -> Unit) {}

fun main() {
    setOnClickListener({ view, click ->
        println("Clicked")
    })
}
```

If we want to ignore a parameter, we can use underscore (_) instead of its name. This is a placeholder that shows that this parameter is ignored.

```
setOnClickListener({ _, _ ->
    println("Clicked")
})
```

IDEA IntelliJ suggests transforming unused parameters into underscores.

We can also use destructuring when defining a lambda expression's parameters[2].

```
data class User(val name: String, val surname: String)
data class Element(val id: Int, val type: String)

fun setOnClickListener(listener: (User, Element) -> Unit) {}

fun main() {
    setOnClickListener({ (name, surname), (id, type) ->
        println(
            "User $name $surname clicked " +
            "element $id of type $type"
        )
    })
}
```

[2]More about destructuring in *Kotlin Essentials*, *Data modifier* chapter.

Trailing lambdas

Kotlin introduced a convention: if we call a function whose last parameter is of a functional type, we can define a lambda expression **outside** the parentheses. This feature is known as *trailing lambda*. If it is the only argument we define, we can skip the parameter bracket and just define a lambda expression. Take a look at these examples.

```
inline fun <R> run(block: () -> R): R = block()

inline fun repeat(times: Int, block: (Int) -> Unit) {
    for (i in 0 until times) {
        block(i)
    }
}

fun main() {
    run({ println("A") }) // A
    run() { println("A") } // A
    run { println("A") } // A

    repeat(2, { print("B") }) // BB
    println()
    repeat(2) { print("B") } // BB
}
```

> In the example above, both `run` and `repeat` are simplified functions from the standard library.

This means that we can call our `setOnClickListener` in the following way:

```
setOnClickListener { _, _ ->
    println("Clicked")
}
```

Remember `sum` and `product` from the introduction? We have implemented them using the `fold` function with a trailing lambda.

```
fun sum(a: Int, b: Int) = (a..b).fold(0) { acc, i -> acc + i }

fun product(a: Int, b: Int) = (a..b).fold(1) { acc, i -> acc * i }
```

But be careful because this convention works only for the last parameter. Take a look at the snippet below and guess what will be printed.

```
fun call(before: () -> Unit = {}, after: () -> Unit = {}) {
    before()
    print("A")
    after()
}

fun main() {
    call({ print("C") })
    call { print("B") }
}
```

The answer is "CAAB". Tricky, isn't it? If you call a function with more than one functional parameter, use the *named argument convention*[3].

```
fun main() {
    call(before = { print("C") })
    call(after = { print("B") })
}
```

Result values

Lambda expressions were initially designed to implement short functions. Their bodies were designed to be minimalistic; therefore, inside them, instead of using an explicit `return`, the result of the last statement is returned. For example, `{ 42 }` returns `42` because this number is the last statement. `{ 1; 2 }` returns `2`. `{ 1; 2; 3 }` returns `3`.

```
fun main() {
    val f = {
        10
        20
        30
    }
    println(f()) // 30
}
```

In most use cases, this is really convenient, but what can we do if we need to finish our function prematurely? A simple `return` will not help (for reasons we will cover later).

[3]Best practices regarding naming arguments are explained in *Effective Kotlin*, *Item 17: Consider naming arguments*. The named argument convention is explained in *Kotlin Essentials*, *Functions* chapter.

```
fun main() {
    onUserChanged { user ->
        if (user == null) return // compilation error
        cheerUser(user)
    }
}
```

To use `return` in the middle of a lambda expression, we need to use a label that marks this lambda expression. We specify a label before a lambda expression by using the label name followed by `@`. Then, we can return from this lambda expression calling `return` on the defined label.

```
fun main() {
    onUserChanged someLabel@{ user ->
        if (user == null) return@someLabel
        cheerUser(user)
    }
}
```

To simplify this process, there is a convention: if a lambda expression is used as an argument to a function, the name of this function becomes its default label. So, without specifying a label, we could return from the lambda using the `onUserChanged` label in the example above.

```
fun main() {
    onUserChanged { user ->
        if (user == null) return@onUserChanged
        cheerUser(user)
    }
}
```

This is how we typically return from a lambda expression prematurely. In theory, specifying custom labels might be useful for returning from outer lambda expressions.

```
fun main() {
    val magicSquare = listOf(
        listOf(2, 7, 6),
        listOf(9, 5, 1),
        listOf(4, 3, 8),
    )
    magicSquare.forEach line@ { line ->
        var sum = 0
        line.forEach { elem ->
            sum += elem
            if (sum == 15) {
                return@line
            }
        }
        print("Line $line not correct")
    }
}
```

However, in practice, this is not only rare but also considered a poor practice[4], because it violates the usual encapsulation rules. This is similar to throwing an exception from an inner function, but in this case the caller can at least decide to catch and react. However, returning from an outer label completely ignores the intermediate callers.

Lambda expression examples

The previous chapter showed a set of functions implemented with *anonymous functions*. This is how they might be defined with *lambda expressions*:

```
fun main() {
    val cheer: () -> Unit = {
        println("Hello")
    }
    cheer.invoke() // Hello
    cheer() // Hello

    val printNumber: (Int) -> Unit = { i: Int ->
        println(i)
    }
    printNumber.invoke(10) // 10
```

[4] Also, the above algorithm is poorly implemented. It should instead use `sumOf` function, which we will present later in this book.

```
    printNumber(20) // 20

    val log: (String, String) -> Unit =
        { ctx: String, message: String ->
            println("[$ctx] $message")
        }
    log.invoke("UserService", "Name changed")
    // [UserService] Name changed
    log("UserService", "Surname changed")
    // [UserService] Surname changed

    data class User(val id: Int)

    val makeAdmin: () -> User = { User(id = 0) }
    println(makeAdmin()) // User(id=0)

    val add: (String, String) -> String =
        { s1: String, s2: String -> s1 + s2 }
    println(add.invoke("A", "B")) // AB
    println(add("C", "D")) // CD

    data class Name(val name: String)

    val toName: (String) -> Name =
        { name: String -> Name(name) }
    val name: Name = toName("Cookie")
    println(name) // Name(name=Cookie)
}
```

A lambda expression can specify the types of parameters, so the result type can be inferred:

```
val cheer = {
    println("Hello")
}
val printNumber = { i: Int ->
    println(i)
}
val log = { ctx: String, message: String ->
    println("[$ctx] $message")
}
val makeAdmin = { User(id = 0) }
val add = { s1: String, s2: String -> s1 + s2 }
val toName = { name: String -> Name(name) }
```

On the other hand, when parameter types can be inferred, lambda expressions do not need to define them:

```
val printNumber: (Int) -> Unit = { i ->
    println(i)
}
val log: (String, String) -> Unit = { ctx, message ->
    println("[$ctx] $message")
}
val add: (String, String) -> String = { s1, s2 -> s1 + s2 }
val toName: (String) -> Name = { name -> Name(name) }
```

An implicit name for a single parameter

When a lambda expression has **exactly one parameter**, we can reference it using the `it` keyword instead of specifying its name. Since the type of `it` cannot be specified explicitly, it needs to be inferred. Despite this, it is still a very popular feature.

```
val printNumber: (Int) -> Unit = { println(it) }
val toName: (String) -> Name = { Name(it) }

// Real-life example, functions will be explained later
val newsItemAdapters = news
    .filter { it.visible }
    .sortedByDescending { it.publishedAt }
    .map { it.toNewsItemAdapter() }
```

Closures

A lambda expression can use and modify variables from the scope where it is defined.

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

fun main() {
    val counter1 = makeCounter()
    val counter2 = makeCounter()

    println(counter1()) // 0
    println(counter1()) // 1
    println(counter2()) // 0
    println(counter1()) // 2
    println(counter1()) // 3
    println(counter2()) // 1
}
```

A lambda expression that refers to an object defined outside its scope, like the lambda expression in the above example that refers to the local variable `i`, is called a *closure*.

Lambda expressions vs anonymous functions

Let's compare lambda expressions to anonymous functions. They are both *function literals*, i.e., structures that create an object representing a function. Under the hood, their efficiency is the same. So, when should we choose one over the other? Take a look at the `processor` variable below, which is defined using both approaches.

```
val processor = label@{ data: String ->
    if (data.isEmpty()) {
        return@label null
    }

    data.uppercase()
}

val processor = fun(data: String): String? {
    if (data.isEmpty()) {
        return null
    }
```

```
    return data.uppercase()
}
```

Lambda expressions are shorter but also less explicit. They return the last expression without an explicit `return` keyword. To use `return` we need to have a label.

Anonymous functions are longer, but it is clear that they define a function. They use an explicit `return` and must specify the result type.

Lambda expressions were mainly designed for single-expression functions, and the documentation suggests using anonymous functions for longer bodies. Although developers used to use lambda expressions practically everywhere, nowadays anonymous functions seem nearly forgotten.

The popularity of lambda expressions is supported by the additional features: trailing lambda, an implicit name for a single parameter, and non-local return (this will be explained later). So, I understand if you decide to forget about anonymous functions and use lambda expressions everywhere. Many developers have already done this.

However, before we close this discussion, we must introduce one more approach for creating objects representing functions. This will be a serious competitor to lambda expressions because it is shorter and has a good-looking, functional style. Let's talk about function references.

Exercise: Function types and literals

The following class shows example implementations of methods `add`, `printNum`, `triple`, `produceName` and `longestOf`:

```
class FunctionsClassic {

    fun add(num1: Int, num2: Int): Int = num1 + num2

    fun printNum(num: Int) {
        print(num)
    }

    fun triple(num: Int): Int = num * 3

    fun produceName(name: String): Name = Name(name)

    fun longestOf(
        str1: String,
        str2: String,
```

```
        str3: String,
    ): String = maxOf(str1, str2, str3, compareBy { it.length })
}

data class Name(val name: String)
```

Your task is to write similar classes, but instead of defining functions, they should define properties with function types. Those properties should represent the same functions as in the `FunctionsClassic` class. For instance, the `add` function should be represented by a property named `add` of type `(Int, Int) -> Int`. The behavior of those properties should also be identical to the behavior of functions from `FunctionsClassic` Implement classes with the following implementation of functional properties:

- `AnonymousFunctionalTypeSpecified` - should define properties with explicit function types and define their values using anonymous functions. The types of parameters in those anonymous functions should be inferred.
- `AnonymousFunctionalTypeInferred` - should define properties with inferred function types from anonymous function definitions that should be used to define values. The parameters of those anonymous functions should be explicitly typed.
- `LambdaFunctionalTypeSpecified` - should define properties with explicit function types and define their values using lambda expressions. The types of parameters in those lambda expressions should be inferred. You should not use the implicit parameter `it` in this class.
- `LambdaFunctionalTypeInferred` - should define properties with inferred function types from lambda expression definitions that should be used to define values. The parameters of those lambda expressions should be explicitly typed.

- `LambdaUsingImplicitParameter` - should define properties with explicit function types and define their values using lambda expressions, just like `LambdaFunctionalTypeSpecified`, but it should use implicit parameter convention whenever possible.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/base/Functional.kt`. You can clone this project and solve this exercise locally.

Exercise: Observable value

Your task is to implement the `Observable` class, which should hold a value and allow observing its changes. It should have a `value` property, which should be settable. It should also have a `observe` function, which should take a function that will be called whenever the value changes.

```
val observable = Observable(1)
println(observable.value) // 1
observable.observe { println("Changed to $it") }
observable.value = 2 // Changed to 2
println(observable.value) // 2
observable.observe { println("now it is $it") }
observable.value = 3
// Changed to 3
// now it is 3
```

Starting code:

```
class Observable<T>(initial: T) {
    var value: T = initial
}
```

Starting code, example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/base/Observable.kt`. You can clone this project and solve this exercise locally.

Function references

When we need a function as an object, we can create it with a lambda expression, but we can also reference an existing function. The second approach is often shorter and more convenient. In this chapter, we will learn about the different kinds of function references, and we will see how they might be used in practice.

In our examples, we will reference the functions from the following code. These will be the basic functions in this chapter.

```
data class Complex(val real: Double, val imaginary: Double) {
    fun doubled(): Complex =
        Complex(this.real * 2, this.imaginary * 2)
    fun times(num: Int) =
        Complex(real * num, imaginary * num)
}

fun zeroComplex(): Complex = Complex(0.0, 0.0)

fun makeComplex(
    real: Double = 0.0,
    imaginary: Double = 0.0
) = Complex(real, imaginary)

fun Complex.plus(other: Complex): Complex =
    Complex(real + other.real, imaginary + other.imaginary)
fun Int.toComplex() = Complex(this.toDouble(), 0.0)
```

Top-level functions references

We use `::` and a function name to reference a top-level function[5]. Function references are part of the Kotlin reflection API and support introspection. If you include the `kotlin-reflect` dependency in your project, you can use a function reference to check if the referenced function has the `open` modifier, what annotation it has, etc.[6]

[5]Top-level function is a function defined outside a class, so in a file.
[6]More about reflection in *Advanced Kotlin*, *Reflection* chapter.

```kotlin
fun add(a: Int, b: Int) = a + b

fun main() {
    val f = ::add // function reference
    println(f.isOpen) // false
    println(f.visibility) // PUBLIC
    // The above statements require `kotlin-reflect`
    // dependency
}
```

However, function references also implement function types and can be used as function literals. Such usages are not considered "real" reflection and introduce no performance overhead compared to lambda expressions[7].

```kotlin
fun add(a: Int, b: Int) = a + b

fun main() {
    val f: (Int, Int) -> Int = ::add
    // an alternative to:
    // val f: (Int, Int) -> Int = { a, b -> add(a, b) }
    println(f(10, 20)) // 30
}
```

Notice that `add` is a function with two parameters of type `Int`, and result type `Int`, so its reference function type is `(Int, Int) -> Int`.

Let's get back to our basic functions. Can you guess what the function type of `zeroComplex` and `makeComplex` should be?

A function type specifies the parameters and the result type. The function `zeroComplex` has no parameters, and its result type is `Complex`, so the function type of its function reference is `() -> Complex`. The function `makeComplex` has two parameters of type `Double`, and its result type is `Complex`, so the function type of its function reference is `(Double, Double) -> Complex`.

[7]For this, the reference needs to be immediately typed as a function type.

```kotlin
fun zeroComplex(): Complex = Complex(0.0, 0.0)

fun makeComplex(
    real: Double = 0.0,
    imaginary: Double = 0.0
) = Complex(real, imaginary)

data class Complex(val real: Double, val imaginary: Double)

fun main() {
    val f1: () -> Complex = ::zeroComplex
    println(f1()) // Complex(real=0.0, imaginary=0.0)

    val f2: (Double, Double) -> Complex = ::makeComplex
    println(f2(1.0, 2.0)) // Complex(real=1.0, imaginary=2.0)
}
```

Since the function `makeComplex` has default arguments for its parameters, it should also implement `(Double) -> Complex` and `() -> Complex`. Limited support for such behavior was introduced in Kotlin 1.4, but a reference must still be used as an argument.

```kotlin
fun produceComplex1(producer: ()->Complex) {}
produceComplex1(::makeComplex)
fun produceComplex2(producer: (Double)->Complex) {}
produceComplex2(::makeComplex)
```

Method references

When you reference a method, you need to start with a type, followed by `::` and the method name. Every method needs a receiver, namely the object on which the function should be called. Function references expect it as the first parameter. Take a look at the example below.

```kotlin
data class Number(val num: Int) {
    fun toFloat(): Float = num.toFloat()
    fun times(n: Int): Number = Number(num * n)
}

fun main() {
    val numberObject = Number(10)
    // member function reference
    val float: (Number) -> Float = Number::toFloat
    // `toFloat` has no parameters, but its function type
    // needs a receiver of type `Number`
    println(float(numberObject)) // 10.0
    val multiply: (Number, Int) -> Number = Number::times
    println(multiply(numberObject, 4)) // Number(num = 40.0)
    // `times` has one parameter of type `Int`, but its
    // function type also needs a receiver of type `Number`
}
```

The `toFloat` function has no explicit parameters, but its function reference requires a receiver of type `Number`. The `times` function has only one explicit parameter of type `Int`, but it also requires another one for the receiver.

Do you remember `sum` and `product` from the introduction? We implemented them using lambda expressions, but we could also have used method references.

```kotlin
fun sum(a: Int, b: Int) =
    (a..b).fold(0, Int::plus)

fun product(a: Int, b: Int) =
    (a..b).fold(1, Int::times)
```

Getting back to our basic functions, can you deduce the function type of `Complex::doubled` and `Complex::times`?

`doubled` has no explicit parameters, a receiver of type `Complex`, and the result type is `Complex`; therefore, the function type of its function reference is `(Complex) -> Complex`. `times` has an explicit parameter of type `Int`, a receiver of type `Complex`, and the result type is `Complex`; therefore, the function type of its function reference is `(Complex, Int) -> Complex`.

```
data class Complex(val real: Double, val imaginary: Double) {
   fun doubled(): Complex =
       Complex(this.real * 2, this.imaginary * 2)
   fun times(num: Int) =
       Complex(real * num, imaginary * num)
}

fun main() {
   val c1 = Complex(1.0, 2.0)

   val f1: (Complex) -> Complex = Complex::doubled
   println(f1(c1)) // Complex(real=2.0, imaginary=4.0)

   val f2: (Complex, Int) -> Complex = Complex::times
   println(f2(c1, 4)) // Complex(real=4.0, imaginary=8.0)
}
```

Extension function references

We can reference extension functions in the same way as member functions. Their function types are also analogous.

```
data class Number(val num: Int)

fun Number.toFloat(): Float = num.toFloat()
fun Number.times(n: Int): Number = Number(num * n)

fun main() {
   val num = Number(10)
   // extension function reference
   val float: (Number) -> Float = Number::toFloat
   println(float(num)) // 10.0
   val multiply: (Number, Int) -> Number = Number::times
   println(multiply(num, 4)) // Number(num = 40.0)
}
```

Can you now guess the function type of `Complex::plus` and `Int::toComplex` from our basic functions?

`plus` has a `Complex` parameter, a receiver of type `Complex`, and it returns `Complex`; therefore, the function type of its function reference is `(Complex, Complex) -> Complex`. The `toComplex` function has no parameters, a receiver of type `Int`, and it returns `Complex`; therefore, the function type of its function reference is `(Int) -> Complex`.

```
data class Complex(val real: Double, val imaginary: Double)

fun Complex.plus(other: Complex): Complex =
   Complex(real + other.real, imaginary + other.imaginary)

fun Int.toComplex() = Complex(this.toDouble(), 0.0)

fun main() {
   val c1 = Complex(1.0, 2.0)
   val c2 = Complex(4.0, 5.0)

   // extension function reference
   val f1: (Complex, Complex) -> Complex = Complex::plus
   println(f1(c1, c2)) // Complex(real=5.0, imaginary=7.0)

   val f2: (Complex, Int) -> Complex = Complex::times
   println(f2(c1, 4)) // Complex(real=4.0, imaginary=8.0)
}
```

Method references and generic types

We reference a method on a type, not a property. So, if you want to reference `sum`, which is an extension function on the type `List<Int>`, you need to use `List<Int>::sum`. If you want to reference `isNullOrBlank`, which is an extension property on the type `String?`, you should use `String?::isNullOrBlank`[8].

```
class TeamPoints(val points: List<Int>) {
   fun <T> calculatePoints(operation: (List<Int>) -> T): T =
       operation(points)
}

fun main() {
   val teamPoints = TeamPoints(listOf(1, 3, 5))

   val sum = teamPoints
       .calculatePoints(List<Int>::sum)
   println(sum) // 9

   val avg = teamPoints
```

[8]It is possible to reference this function by `String::isNullOrBlank`, but such reference function type is `(String) -> Boolean`, makes it not accept `null` and effectively behave like `String::isBlank`.

```
        .calculatePoints(List<Int>::average)
    println(avg) // 3.0

    val invalid = String?::isNullOrBlank
    println(invalid(null)) // true
    println(invalid("  ")) // true
    println(invalid("AAA")) // false
}
```

When you reference a method from a generic class, its type arguments need to be explicit. So, in the example below, to reference the `unbox` method, we need to use `Box<String>::unbox`, and the `Box::unbox` notation is not acceptable.

```
class Box<T>(private val value: T) {
    fun unbox(): T = value
}

fun main() {
    val unbox = Box<String>::unbox
    val box = Box("AAA")
    println(unbox(box)) // AAA
}
```

Bounded function references

We have learned how to reference a method on a type, but there is also another option: we can reference a method on an object instance. Such references are called **bounded function references**.

```
data class Number(val num: Int) {
    fun toFloat(): Float = num.toFloat()
    fun times(n: Int): Number = Number(num * n)
}

fun main() {
    val num = Number(10)
    // bounded function reference
    val getNumAsFloat: () -> Float = num::toFloat
    // There is no need for receiver type in function type,
    // because reference is already bound to an object
    println(getNumAsFloat()) // 10.0
    val multiplyNum: (Int) -> Number = num::times
```

```
    println(multiplyNum(4)) // Number(num = 40.0)
}
```

Notice that the function type of `num::toFloat` is `() -> Float` in the example above. We have previously learned that the function type of `Number::toFloat` is `(Number) -> Float`; therefore, in the regular method reference notation, the receiver type will be in the first position. In bounded function references, the receiver object is already provided in the reference, so there is no need to specify it additionally.

Getting back to our basic functions, can you deduce the type of the bounded references to `doubled`, `times`, `plus`, and `toComplex`? The answers can be found in the code below.

```
data class Complex(val real: Double, val imaginary: Double) {
    fun doubled(): Complex =
        Complex(this.real * 2, this.imaginary * 2)
    fun times(num: Int) =
        Complex(real * num, imaginary * num)
}

fun Complex.plus(other: Complex): Complex =
    Complex(real + other.real, imaginary + other.imaginary)
fun Int.toComplex() = Complex(this.toDouble(), 0.0)

fun main() {
    val c1 = Complex(1.0, 2.0)

    val f1: () -> Complex = c1::doubled
    println(f1()) // Complex(real=2.0, imaginary=4.0)

    val f2: (Int) -> Complex = c1::times
    println(f2(17)) // Complex(real=17.0, imaginary=34.0)

    val f3: (Complex) -> Complex = c1::plus
    println(f3(Complex(12.0, 13.0)))
    // Complex(real=13.0, imaginary=15.0)

    val f4: () -> Complex = 42::toComplex
    println(f4()) // Complex(real=42.0, imaginary=0.0)
}
```

Bounded function references also work on object expressions and object declarations[9].

[9]More about object expressions and object declarations in *Kotlin Essentials*, *Objects* chapter.

```kotlin
object SuperUser {
    fun getId() = 0
}

fun main() {
    val myId = SuperUser::getId
    println(myId()) // 0

    val obj = object {
        fun cheer() {
            println("Hello")
        }
    }
    val f = obj::cheer
    f() // Hello
}
```

I find bounded function references especially useful when using libraries like RxJava or Reactor, where we often set handlers for different kinds of events. Small, simple handlers can be defined using lambda expressions. However, extracting them as member functions and setting bounded function references as handlers is a good idea for larger and more complicated handlers.

```kotlin
class MainPresenter(
    private val view: MainView,
    private val repository: MarvelRepository
) : BasePresenter() {

    fun onViewCreated() {
        subscriptions += repository.getAllCharacters()
            .applySchedulers()
            .subscribeBy(
                onSuccess = this::show,
                onError = view::showError
            )
    }

    fun show(items: List<MarvelCharacter>) {
        // ...
        view.show(items)
    }
}
```

Using the bounded function reference is really convenient in this case because handlers need to have access to the `MainPresenter` properties, but `getAllCharacters` should not know anything about this.

A bounded function reference on the receiver (`this`) can be used implicitly, so `this::show` can also be replaced with `::show`.

Constructor references

A constructor is also considered a function in Kotlin. We call and reference it in the same way as all other functions. This means that to reference the `Complex` class constructor, we need to use `::Complex`. The constructor reference has the same parameters as the constructor it references, and its result type is the type of the class whose constructor it is.

```
data class Complex(val real: Double, val imaginary: Double)

fun main() {
  // constructor reference
  val produce: (Double, Double) -> Complex = ::Complex
  println(produce(1.0, 2.0))
  // Complex(real=1.0, imaginary=2.0)
}
```

I find constructor references useful when I map elements from one type to another using a constructor. This could be especially useful for mapping to wrapper classes. However, mapping using a constructor should not be used too often as we prefer factory functions (like conversion functions) instead of secondary constructors[10].

```
class StudentId(val value: Int)
class UserId(val value: Int) {
    constructor(studentId: StudentId) : this(studentId.value)
}

fun main() {
    val ints: List<Int> = listOf(1, 1, 2, 3, 5, 8)
    val studentIds: List<StudentId> = ints.map(::StudentId)
    val userIds: List<UserId> = studentIds.map(::UserId)
}
```

[10]See *Effective Kotlin, Item 32: Consider factory functions instead of secondary constructors*.

Bounded object declaration references

One of the motivations for the introduction of bounded function references was to make a simple way to reference object declaration methods[11]. Every object declaration is a singleton, so its name serves as the only object reference. Thanks to the bounded function reference feature, we can reference object declaration methods using its name, followed by two colons (::), then the method name.

```
object Robot {
    fun moveForward() {
        /*...*/
    }
    fun moveBackward() {
        /*...*/
    }
}

fun main() {
    Robot.moveForward()
    Robot.moveBackward()

    val action1: () -> Unit = Robot::moveForward
    val action2: () -> Unit = Robot::moveBackward
}
```

Companion objects are also a form of object declaration. However, referencing their methods using the class name is not enough. We need to use the real companion name, which is `Companion` by default.

```
class Drone {
    fun setOff() {}
    fun land() {}

    companion object {
        fun makeDrone(): Drone = Drone()
    }
}

fun main() {
    val maker: () -> Drone = Drone.Companion::makeDrone
}
```

[11]For details, see KEEP, link: kt.academy/l/keep-bound-ref

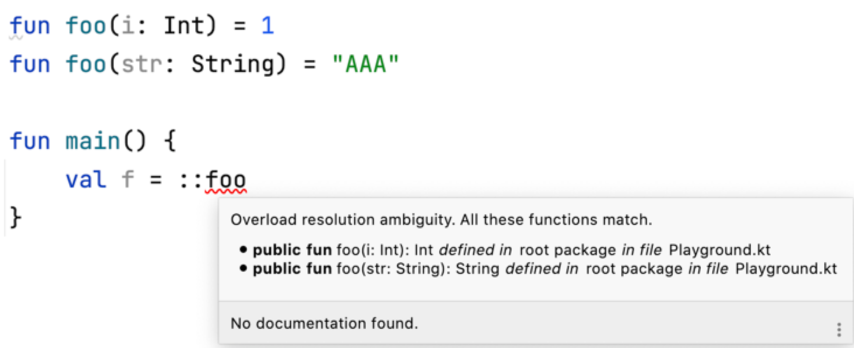
Function overloading and references

Kotlin allows function overloading, which means defining multiple functions with the same name. During compilation, the Kotlin compiler decides which function should be used based on the types of arguments used.

```
fun foo(i: Int) = 1
fun foo(str: String) = "AAA"

fun main() {
  println(foo(123)) // 1
  println(foo("")) // AAA
}
```

The same logic is used when we use function references. The compiler determines which function should be chosen based on the expected type. Without a specified type, our code will not compile due to ambiguity.



Therefore, when we eliminate ambiguity with a type, everything will be correctly determined and resolved.

```
fun foo(i: Int) = 1
fun foo(str: String) = "AAA"

fun main() {
  val fooInt: (Int) -> Int = ::foo
  println(fooInt(123)) // 1
  val fooStr: (String) -> String = ::foo
  println(fooStr("")) // AAA
}
```

The same is true when we have multiple constructors.

```
class StudentId(val value: Int)
data class UserId(val value: Int) {
    constructor(studentId: StudentId) : this(studentId.value)
}

fun main() {
    val intToUserId: (Int) -> UserId = ::UserId
    println(intToUserId(1)) // UserId(value=1)

    val studentId = StudentId(2)
    val studentIdToUserId: (StudentId) -> UserId = ::UserId
    println(studentIdToUserId(studentId)) // UserId(value=2)
}
```

Property references

A property can be considered as a getter or as a getter and a setter. That is why its reference implements the getter function type.

```
data class Complex(val real: Double, val imaginary: Double)

fun main() {
    val c1 = Complex(1.0, 2.0)
    val c2 = Complex(3.0, 4.0)

    // property reference
    val getter: (Complex) -> Double = Complex::real

    println(getter(c1)) // 1.0
    println(getter(c2)) // 3.0

    // bounded property reference
    val c1ImgGetter: () -> Double = c1::imaginary
    println(c1ImgGetter()) // 2.0
}
```

For `var`, you can reference the setter using the `setter` property from the property reference, but this requires `kotlin-reflect`; therefore, I recommend avoiding this approach because it might impact your code's performance.

There are many kinds of references. Some developers like using them, while others avoid them. Anyway, it is good to know how function references look and behave. It is worth practicing them as they can help make our code more elegant in applications where functional programming concepts are widely used.

Exercise: Inferred function types

Consider the following code:

```
class Centimeter(val value: Double) {
    fun plus(other: Centimeter): Centimeter =
        Centimeter(value + other.value)

    fun times(other: Double): Centimeter =
        Centimeter(value * other)

    override fun toString(): String = "$value cm"
}

val Int.cm get() = Centimeter(this.toDouble())

fun distance(from: Centimeter, to: Centimeter): Centimeter =
    Centimeter(abs(to.value - from.value))
```

For the below function references, predict what will be the result function type:
- `Centimeter::plus`
- `Centimeter::times`
- `Centimeter::value`
- `Centimeter::toString`
- `Centimeter(1.0)::plus`
- `Centimeter(2.0)::times`
- `Centimeter(3.0)::value`
- `Centimeter(4.0)::toString`
- `Int::cm`
- `123::cm`
- `::distance`

Exercise: Function references

This is a continuation of the exercise “Function types and literals”. This time, your task is to implement the following classes:

- `FunctionReference` - that defines properties `printNum`, `triple` and `produceName` with explicit function types, and define their values using references to functions from the Kotlin standard library or from `Name` class.
- `FunctionMemberReference` that defines properties `printNum`, `triple`, `produceName` and `longestOf` with explicit function types, and define their values using references to methods from its own body.
- `BoundedFunctionReference` that defines properties `printNum`, `triple`, `produceName` and `longestOf` with explicit function types, and define their values using references to methods `classic` object.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/base/References.kt`. You can clone this project and solve this exercise locally.

SAM Interface support in Kotlin

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Support for Java SAM interfaces in Kotlin

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Functional interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Inline functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Inline functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Inline functions with functional parameters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Non-local return

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Crossinline and noinline

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Reified type parameters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Inline properties

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Costs of the inline modifier

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Using inline functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Inline functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Collection processing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`forEach` and `onEach`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`filter`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`map`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`mapNotNull`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`flatMap`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Implement map

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Optimize collection processing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`fold`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`reduce`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`sum`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`withIndex` and indexed variants

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`take`, `takeLast`, `drop`, `dropLast` and `subList`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Adding element at position

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Getting elements at certain positions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Finding an element

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Counting elements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`any`, `all` and `none`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Implement shop functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`partition`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`groupBy`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Associating to a map

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`distinct` and `distinctBy`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Prime access list

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Sorting: `sorted`, `sortedBy` and `sortedWith`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Sorting mutable collections

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Maximum and minimum

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`shuffled` and `random`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Top articles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Refactor collection processing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`zip` and `zipWithNext`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Windowing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`joinToString`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

`Map`, `Set` and `String` processing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Passing students list

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Best students list

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Functional Quick Sort

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Powerset

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: All possible partitions of a set

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Sequences

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

What is a sequence?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Order is important

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Sequences do the minimum number of operations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Sequences can be infinite

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Sequences do not create collections at every processing step

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

When aren’t sequences faster?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

What about Java streams?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Kotlin Sequence debugging

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Understanding sequences

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Type Safe DSL Builders

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

A function type with a receiver

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Simple DSL builders

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Using `apply`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Simple DSL-like builders

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Multi-level DSLs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

DslMarker

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

A more complex example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

When should we use DSLs?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: HTML table DSL

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Creating user table row

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Scope functions

There is a group of minimalistic but useful inline functions from the standard library called *scope functions*. This group typically includes `let`, `apply`, `also`, `run` and `with`. Some developers also include `takeIf` and `takeUnless` in this group. They are all extensions on any generic type[12]. All scope functions are just a few lines long. Let's discuss their usages and how they work, starting with the functions I find most useful.

`let`

```
// let implementation without contract
inline fun <T, R> T.let(block: (T) -> R): R = block(this)
```

`let` is a simple function, yet it is used in many Kotlin idioms. It can be compared to the `map` function but for a single object: it transforms an object using a lambda expression.

```
fun main() {
    println(listOf("a", "b").map { it.uppercase() }) // [A, B]
    println("a".let { it.uppercase() }) // A
}
```

Let's see its common use cases.

Mapping a single object

To understand how `let` is used, let's imagine that you need to read a zip file with buffering, unpack it, and read an object from the result. On JVM, we use input streams for such operations. We first create a `FileInputStream` to read a file, and then we decorate it with classes that add the capabilities we need.

[12]Except for `with`, which is not an extension function.

```
val fis = FileInputStream("someFile.gz")
val bis = BufferedInputStream(fis)
val gis = ZipInputStream(bis)
val ois = ObjectInputStream(gis)
val someObject = ois.readObject()
```

This pattern is not very readable because we create plenty of variables that are used only once. We can easily make a mistake, for instance by using an incorrect variable at any step. How can we improve it? By using the `let` function! We can first create `FileInputStream`, and then decorate it using `let`:

```
val someObject = FileInputStream("someFile.gz")
    .let { BufferedInputStream(it) }
    .let { ZipInputStream(it) }
    .let { ObjectInputStream(it) }
    .readObject()
```

If you prefer, you can also use constructor references[13]:

```
val someObject = FileInputStream("someFile.gz")
    .let(::BufferedInputStream)
    .let(::ZipInputStream)
    .let(::ObjectInputStream)
    .readObject()
```

Using `let`, we can form a nice flow of how an element is transformed. What is more, if a nullability is introduced at any step, we can use `let` conditionally with a safe call. To see this in practice, let's imagine that we are implementing a service that, based on a user token, responds with this user's active courses.

```
class CoursesService(
    private val userRepository: UserRepository,
    private val coursesRepository: CoursesRepository,
    private val userCoursesFactory: UserCoursesFactory,
) {
    // Imperative approach, without let
    fun getActiveCourses(token: String): UserCourses? {
        val user = userRepository
            .getUser(token) ?: return null
        val activeCourses = coursesRepository
            .getActiveCourses(user.id) ?: return null
```

[13]Constructor references were explained in the chapter *Function references*.

```
        return userCoursesFactory.produce(activeCourses)
    }

    // Functional approach, using let
    fun getActiveCourses(token: String): UserCourses? =
        userRepository.getUser(token)
            ?.let { coursesRepository.getActiveCourses(it.id) }
            ?.let(userCoursesFactory::produce)
}
```

In these cases, `let` is not necessary, but it's very convenient. I see similar usage quite often, especially on backend applications. It makes our functions form a nice flow of data, and it lets us easily control the scope of each variable. It also has downsides, such as the fact that debugging is harder, so you need to decide yourself whether to use this approach in your applications.

Here is another practical example, coming from AnkiMarkdown library, that is using let to update notes, and get the result of this operation:

```
val noteContent = api.getNotesInDeck(deckName)
    .map(parser::apiNoteToNote)
    .let(parser::writeNotes)
```

Here is an example from the same project, where `let` is used to decorate a string with a prefix and a postfix:

```
fun headerToText(headerConfig: HeaderConfig): String =
    headerConfig.toYamlString()
        .let { "---\n$it\n---\n\n" }
```

The problem with member extension functions

At this point, it is worth mentioning that there is an ongoing discussion about transforming objects from one class to another. Let's say that we need to transform from `UserCreationRequest` to `UserDto`. The typical Kotlin way is to define a `toUserDto` or `toDomain` method (either a member function or an extension function).

```
class UserCreationRequest(
    val id: String,
    val name: String,
    val surname: String,
)

class UserDto(
    val userId: String,
    val firstName: String,
    val lastName: String,
)

fun UserCreationRequest.toUserDto() = UserDto(
    userId = this.id,
    firstName = this.name,
    lastName = this.surname,
)
```

The problem arises when the transformation function needs to use some external services. It needs to be defined in a class, and defining member extension functions is an anti-pattern[14].

```
class UserCreationRequest(
    val name: String,
    val surname: String,
)

class UserDto(
    val userId: String,
    val firstName: String,
    val lastName: String,
)

class UserCreationService(
    private val userRepository: UserRepository,
    private val idGenerator: IdGenerator,
) {
    fun addUser(request: UserCreationRequest): User =
        request.toUserDto()
            .also { userRepository.addUser(it) }
            .toUser()
```

[14]For details, see *Effective Kotlin, Item 46: Avoid member extensions*.

```
    // Anti-pattern! Avoid using member extensions
    private fun UserCreationRequest.toUserDto() = UserDto(
        userId = idGenerator.generate(),
        firstName = this.name,
        lastName = this.surname,
    )
}
```

`also` function will be explained next.

A good solution to this problem is defining transformation functions as regular functions in such cases, and if we want to call them "on an object", just use `let`.

```
class UserCreationRequest(
    val name: String,
    val surname: String,
)

class UserDto(
    val userId: String,
    val firstName: String,
    val lastName: String,
)

class UserCreationService(
    private val userRepository: UserRepository,
    private val idGenerator: IdGenerator,
) {
    fun addUser(request: UserCreationRequest): User =
        request.let { createUserDto(it) }
        // or request.let(::createUserDto)
            .also { userRepository.addUser(it) }
            .toUser()

    private fun createUserDto(request: UserCreationRequest) =
        UserDto(
            userId = idGenerator.generate(),
            firstName = request.name,
            lastName = request.surname,
        )
}
```

This approach works just as well when object creation is extracted into a class, like `UserDtoFactory`.

```kotlin
class UserCreationService(
    private val userRepository: UserRepository,
    private val userDtoFactory: UserDtoFactory,
) {
    fun addUser(request: UserCreationRequest): User =
        request.let { userDtoFactory.produce(it) }
            .also { userRepository.addUser(it) }
            .toUser()

//  or
//  fun addUser(request: UserCreationRequest): User =
//      request.let(userDtoFactory::produce)
//          .also(userRepository::addUser)
//          .toUser()
}
```

Moving an operation to the end of processing

The second typical use case for `let` is when we want to move an operation to the end of processing. Let's get back to our example, where we were reading an object from a zip file, but this time we will assume that we need to do something with that object in the end. For simplification, we might be printing it. Again, we face the same problem: we either need to introduce a variable or wrap the processing with a misplaced `print` call.

```kotlin
// Not good, not terrible
val someObject = FileInputStream("/someFile.gz")
    .let(::BufferedInputStream)
    .let(::ZipInputStream)
    .let(::ObjectInputStream)
    .readObject()
println(someObject)

// Terrible
print(
    FileInputStream("/someFile.gz")
        .let(::BufferedInputStream)
        .let(::ZipInputStream)
        .let(::ObjectInputStream)
        .readObject()
)
```

The solution to this problem is to use `let` (or another scope function) to invoke `print` "on" the result.

```
FileInputStream("/someFile.gz")
    .let(::BufferedInputStream)
    .let(::ZipInputStream)
    .let(::ObjectInputStream)
    .readObject()
    .let(::print)
```

This approach allows us to use safe-calls and call operations only on non-null objects.

```
FileInputStream("/someFile.gz")
    .let(::BufferedInputStream)
    .let(::ZipInputStream)
    .let(::ObjectInputStream)
    .readObject()
    ?.let(::print)
```

In real-life applications it is typically calling some other function than `print`. For instance, it can be sending a message:

```
fun sendMessage() {
    FileInputStream("message.gz")
        .let(::BufferedInputStream)
        .let(::ZipInputStream)
        .let(::ObjectInputStream)
        .readObject()
        ?.let(sender::send)
}
```

> Some developers will argue that in such cases one should use `also` instead of `let`. The reasoning is that `let` is a transformation function and should therefore have no side effects, while `also` is dedicated to use for side effects. On the other hand, using `let` in such cases is popular.

Dealing with nullability

The `let` function (and nearly all other scope functions) is called on an object, so it can be called with a safe call. We've already seen a few examples of how this capability helped us in the previous use cases. But it goes even further: `let` is often called just to help with nullability. To see this, let's consider the following example, where we want to print the user name if the `user` is not `null`. Smart casting does not work for variables because they can be modified by another thread. The easiest solution uses `let`.

```kotlin
class User(val name: String)

var user: User? = null

fun showUserNameIfPresent() {
    // will not work, because cannot smart-cast a property
    // if (user != null) {
    //     println(user.name)
    // }

    // works
    // val u = user
    // if (u != null) {
    //     println(u.name)
    // }

    // perfect
    user?.let { println(it.name) }
}
```

In this solution, if `user` is null, `let` is not called (due to the safe call used), and nothing happens. If `user` is not-null, `let` is called, so it calls `println` with the user name. This solution is fully thread-safe even in extreme cases: if `user` is not null during the safe call, and it then changes to `null` straight after that, printing the name will work fine because `it` is the reference to the user that was used at the time of the nullability check.

Here is a practical example, coming from my script for generating solutions from this book as a website:

```kotlin
// How webpage with exercise solutions is generated
val file = File(articlesDir, "all_solutions.md")
file.appendText("# Solutions")
exercises
    .groupByBooks()
    .forEach { (book, articles) ->
        file.appendText("\n\n## Exercises from ${book.title}\n\n")
        articles.forEach { article ->
            file.appendText("### ${article.title}\n\n")
            article.solutionDescription?.let { desc ->
                file.appendText(desc + "\n\n")
            }
            generatePlaygroundSolution(article)?.let { code ->
                file.appendText(code + "\n\n")
            }
```

```
        }
    }
```

> Some developers will again argue that in such cases one should use `also` instead of `let`; again, using `let` for null checks is popular.

These are the key cases where `let` is used. As you can see, it is pretty useful but there are other scope functions with similar characteristics. Let's see these, starting from the one mentioned a few times already: `also`.

also

```
// also implementation without contract
inline fun <T> T.also(block: (T) -> Unit): T {
    block(this)
    return this
}
```

We have mentioned the use of `also` already, so let's discuss it. It is pretty similar to `let`, but instead of returning the result of its lambda expression, it returns the object it is invoked on. So, if `let` is like `map` for a single object, then `also` can be considered an `onEach` for a single object, as `also` returns the object as it is.

`also` is used to invoke an operation on an object. Such operations typically include some side effects. We've used it already to add a user to our database.

```
fun addUser(request: UserCreationRequest): User =
    request.toUserDto()
        .also { userRepository.addUser(it) }
        .toUser()
```

It can be also used for all kinds of additional operations, like printing logs or storing a value in a cache.

```
fun addUser(request: UserCreationRequest): User =
    request.toUserDto()
        .also { userRepository.addUser(it) }
        .also { log("User created: $it") }
        .toUser()

class CachingDatabaseFactory(
    private val databaseFactory: DatabaseFactory,
) : DatabaseFactory {
    private var cache: Database? = null

    override fun createDatabase(): Database = cache
        ?: databaseFactory.createDatabase()
            .also { cache = it }
}
```

As mentioned already, `also` can also be used instead of `let` to unpack a nullable object or move an operation to the end.

```
class User(val name: String)

var user: User? = null

fun showUserNameIfPresent() {
    user?.also { println(it.name) }
}

fun readAndPrint() {
    FileInputStream("/someFile.gz")
        .let(::BufferedInputStream)
        .let(::ZipInputStream)
        .let(::ObjectInputStream)
        .readObject()
        ?.also(::print)
}
```

Here is a function that after referencing a directory, deletes it and then creates is again:

```
val generatedDir = File(sourcesDir, "generated")
    .also { it.deleteRecursively() }
    .also { it.mkdirs() }
```

Here is a function from an Android project, that uses `also` to cache `MoviesDatabase` instance in a variable (if we do not need to change `instance` in any other way, `lazy` can be used instead).

```
private var instance: MoviesDatabase? = null

fun getMoviesDatabase(context: Context): MoviesDatabase = instance
    ?: synchronized(this) {
        buildMoviesDatabase()
            .also { instance = it }
    }
```

takeIf and takeUnless

```
// takeIf implementation without contract
inline fun <T> T.takeIf(predicate: (T) -> Boolean): T? {
    return if (predicate(this)) this else null
}

// takeUnless implementation without contract
inline fun <T> T.takeUnless(predicate: (T) -> Boolean): T? {
    return if (!predicate(this)) this else null
}
```

We already know that `let` is like a `map` for a single object. We know that `also` is like an `onEach` for a single object. So, now it's time to learn about `takeIf` and `takeUnless`, which are like `filter` and `filterNot` for a single object.

Depending on what their predicates return, these functions either return the object they were invoked on, or `null`. `takeIf` returns an untouched object if the predicate returned `true`, and it returns `null` if the predicate returned `false`. `takeUnless` is like `takeIf` with a reversed predicate result (so `takeUnless(pred)` is like `takeIf { !pred(it) }`).

We use these functions to filter out incorrect objects. For instance, if you want to read a file only if it exists.

```
val lines = File("SomeFile")
    .takeIf { it.exists() }
    ?.readLines()
```

We use such checks for safety. For example, if a file does not exist, `readLines` throws an exception. Replacing incorrect objects with `null` helps us handle them safely. It also helps us drop incorrect results, or just replace some values with `null`.

```kotlin
class UserCreationService(
    private val userRepository: UserRepository,
) {
    fun readUser(token: String): User? =
        userRepository.findUser(token)
            .takeIf { it.isValid() }
            ?.toUser()
}
```

Here is a simple example of extracting an article title from the first line:

```kotlin
val title = originalContent.substringBefore("\n")
    .takeIf { it.startsWith("## ") }
    ?.substringAfter("## ")
```

```kotlin
override fun ankiNoteToCard(apiNote: ApiNote): Basic = Basic(
    apiNote.noteId,
    apiNote.field(FRONT_FIELD),
    apiNote.field(BACK_FIELD),
    apiNote.fieldOrNull(EXTRA_FIELD).takeUnless { it.isNullOrBlank() },
)
```

Example showing `takeUnless` *usage in AnkiMarkdown library. It transforms field value to* `null` *if it is empty.*

```kotlin
override fun ankiNoteToCard(apiNote: ApiNote): ListDeletion = ListDeletion(
    id = apiNote.noteId,
    type = API_NOTE_TO_TYPE[apiNote.modelName] ?: error("Unsupported model name " + apiNote.modelName),
    title = apiNote.field("Title"),
    items = (1 ≤ .. ≤ 20).mapNotNull { i ->
        val value = apiNote.fieldOrNull("$i").takeUnless { it.isNullOrBlank() } ?: return@mapNotNull null
        val comment = apiNote.fieldOrNull("$i comment").orEmpty()
        ListDeletion.Item(value, comment)
    },
    extra = apiNote.field("Extra")
)
```

Example showing `takeUnless` *usage in AnkiMarkdown library. Mapping uses* `takeUnless` *to ignore items whose field is empty.*

apply

```
// apply implementation without contract
inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

Moving into a slightly different kind of scope function, it's time to present `apply`, which we already used in the DSL chapter. It works like `also` in that it is called on an object and it returns it, but it introduces an essential change: its parameter is not a regular function type but a function type with a receiver.

This means that if you take `also` and replace it with `apply`, and you replace the argument (typically `it`) with a receiver (`this`) inside the lambda, the resulting code will be the same as before. However, this small change is actually really important. As we learned in the DSL chapter, changing receivers can be both a big convenience and a big danger. This is why we should not change receivers thoughtlessly, and we should restrict `apply` to concrete use cases. These use cases mainly include setting up an object after its creation and defining DSL function definitions.

```
fun createDialog() = Dialog().apply {
    title = "Some dialog"
    message = "Just accept it, ok?"
    // ...
}

fun showUsers(users: List<User>) {
    listView.apply {
        adapter = UsersListAdapter(users)
        layoutManager = LinearLayoutManager(context)
    }
}
```

The dangers of careless receiver overloading

The `this` receiver can be used implicitly, which is both convenient and potentially dangerous. It is not a good situation when we don't know which receiver is being used. In some languages, like JavaScript, this is a common source of mistakes. In Kotlin, we have more control over the receiver, but we can still easily fool ourselves. To see an example, try to guess what the result of the following snippet will be:

```kotlin
class Node(val name: String) {

    fun makeChild(childName: String) =
        create("$name.$childName")
            .apply { print("Created $name") }

    fun create(name: String): Node? = Node(name)
}

fun main() {
    val node = Node("parent")
    node.makeChild("child")
}
```

The intuitive answer is “Created parent.child”, but the actual answer is “Created parent”. Why? Notice that the `create` function declares a nullable result type, so the receiver inside `apply` is `Node?`. Can you call `name` on `Node?` type? No, you need to unpack it first. However, Kotlin will automatically (without any warning) use the outer scope, and that is why “Created parent” will be printed. We fooled ourselves. The solution is to avoid unnecessary receivers (for name resolution). This is not a case in which we should use `apply`: it is a clear case for `also`, for which Kotlin would force us to use the argument value safely if we used it.

```kotlin
class Node(val name: String) {

    fun makeChild(childName: String) =
        create("$name.$childName")
            .also { print("Created ${it?.name}") }

    fun create(name: String): Node? = Node(name)
}

fun main() {
    val node = Node("parent")
    node.makeChild("child") // Created child
}
```

with

```
// with implementation without contract
inline fun <T, R> with(receiver: T, block: T.() -> R): R =
    receiver.block()
```

As you can see, changing a receiver is not a small deal, so it is good to make it visible. `apply` is perfect for object initialization; for most other cases, a very popular option is `with`. **We use `with` to explicitly turn an argument into a receiver.**

In contrast to other scope functions, `with` is a top-level function whose first argument is used as its lambda expression receiver. This makes the new receiver definition really visible.

Typical use cases for `with` include explicit scope changing in Kotlin Coroutines, or specifying multiple assertions on a single object in tests.

```
// explicit scope changing in Kotlin Coroutines
val scope = CoroutineScope(SupervisorJob())
with(scope) {
    launch {
        // ...
    }
    launch {
        // ...
    }
}

// unit-test assertions
with(user) {
    assertEquals(aName, name)
    assertEquals(aSurname, surname)
    assertEquals(aWebsite, socialMedia?.websiteUrl)
    assertEquals(aTwitter, socialMedia?.twitter)
    assertEquals(aLinkedIn, socialMedia?.linkedin)
    assertEquals(aGithub, socialMedia?.github)
}
```

`with` returns the result of its `block` argument, so it can be used as a transformation function; however, this fact is rarely used, and I would suggest using `with` as if it is returning `Unit`.

run

```
// run implementation without contract
inline fun <R> run(block: () -> R): R = block()

// run implementation without contract
inline fun <T, R> T.run(block: T.() -> R): R = block()
```

We have already encountered a top-level `run` function in the *Lambda expressions* chapter. It just invokes a lambda expression. Its only advantage over an immediately invoked lambda expression (`{ /*...*/ }()`) is that it is inline. A plain `run` function is used to form a scope. This is not a common need, but it can be useful from time to time.

```
val locationWatcher = run {
    val positionListener = createPositionListener()
    val streetListener = createStreetListener()
    LocationWatcher(positionListener, streetListener)
}
```

Another variant of the `run` function is invoked on an object. Such an object becomes a receiver inside the `run` lambda expression. However, I do not know any good use cases for this function. Some developers use `run` for certain use cases, but nowadays, I rarely see `run` used in commercial projects. Personally, I avoid using it[15].

Using scope functions

In this chapter, we have learned about many small but useful functions, called **scope functions**. Most of them have clear use cases. Some compete with each other for use cases (especially `let` and `apply`, or `apply` and `with`). Nevertheless, knowing all these functions well and using them in suitable situations is a recipe for nicer and cleaner code. Just please use them only where they make sense; don't use them just to use them.

A simplified comparison between key scope functions is presented in the following table:

[15] Email me if you have some good use cases where you think that `run` clearly fits better than the other scope functions. My email is *marcinmoskala@gmail.com*.

Reference to receiver \ Returns	Receiver	Results of lambda
it	also	let
this	apply	run/with

Exercise: Using scope functions

See the below implementation of `StudentService`. Modify it to use scope functions. As a result, all the methods should be single-expression functions.

```
class StudentService(
    private val studentRepository: StudentRepository,
    private val studentFactory: StudentFactory,
    private val logger: Logger,
) {
    fun addStudent(addStudentRequest: AddStudentRequest):Student?{
        val student = studentFactory
            .produceStudent(addStudentRequest)
            ?: return null
        studentRepository.addStudent(student)
        return student
    }

    fun getStudent(studentId: String): ExposedStudent? {
        val student = studentRepository.getStudent(studentId)
            ?: return null

        logger.log("Student found: $student")
        return studentFactory.produceExposed(student)
    }

    fun getStudents(semester: String): List<ExposedStudent> {
        val request = produceGetStudentsRequest(semester)
```

```
        val students = studentRepository.getStudents(request)
        logger.log("${students.size} students in $semester")
        return students
            .map { studentFactory.produceExposed(it) }
    }

    private fun produceGetStudentsRequest(
        semester: String,
    ): GetStudentsRequest {
        val request = GetStudentsRequest()
        request.expectedSemester = semester
        request.minResult = 3.0
        return request
    }
}
```

> Beware! The form after transformation is shorter but not necessarily better. It is less readable for less experienced Kotlin developers, and it is harder to debug. Outside this exercise, use scope functions with caution.

Starting code and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/scope/Scope.kt`. You can clone this project and solve this exercise locally.

Exercise: orThrow

In my everyday practice, I've noticed that I sometimes need a function that in the middle of a multiline expression can be used to throw an exception if a value is `null`. So I defined it and called it `orThrow`. This is how its usage looks like:

```
fun getUser(userId: String) = userRepository
    .getUser(userId)
    .orThrow { UserNotFoundException(userId) }
    .also { log("Found user: $it") }
    .toUserJson()
```

Your task is to implement `orThrow` function. It should throw the exception specified in its lambda argument if the value is `null`. Otherwise, it should return the value typed as non-nullable.

Unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `functional/scope/orThrow.kt`. You can clone this project and solve this exercise locally.

Context parameters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Extension function problems

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Introducing context parameters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Use cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Concerns

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Named context parameters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise: Logger

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

A birds-eye view of Arrow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Functions and Arrow Core

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Memoization

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Testing higher-order functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Working with nullable types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Working with Result

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Working with Either

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Data Immutability with Arrow Optics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Final words

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Final Project: UserService

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Exercise solutions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Function types and literals

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Observable value

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Inferred function types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Function references

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Inline functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Implement map

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Optimize collection processing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Adding element at position

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Implement shop functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Prime access list

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Top articles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Refactor collection processing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Passing students list

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Best students list

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Functional Quick Sort

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Powerset

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: All possible partitions of a set

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Understanding sequences

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: HTML table DSL

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Creating user table row

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Using scope functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: orThrow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: Logger

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.

Solution: UserService

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_functional.