Kotlin: Programming with Lambdas with examples

12 FEBRUARY 2021 | HITS: 2006

Vovan_ST (/en/component/authorlist/author/15:vovan_st)
ИТ специалист со стажем. Автор статьи. Профиль (/social/1130-vovan-st)

Ratings ★★★★★ (1)

This article covers the following topics:

- Lambda expressions and member references
- Working with collections in a functional style

- Sequences: performing collection operations lazily
- Using Java functional interfaces in Kotlin
- Using lambdas with receivers

Table of contents [Show]

Lambda expressions, or simply *lambdas*, are essentially small chunks of code that can be passed to other functions. With lambdas, you can easily extract common code structures into library functions, and the Kotlin standard library makes heavy use of them. One of the most common uses for lambdas is working with collections, and in this chapter you'll see many examples of replacing common collection access patterns with lambdas passed to standard library functions. You'll also see how lambdas can be used with Java libraries—even those that weren't originally designed with lambdas in mind. Finally, we'll look at lambdas with receivers—a special kind of lambdas where the body is executed in a different context than the surrounding code.

1. Lambda expressions and member references

The introduction of lambdas to Java 8 was one of the longest-awaited changes in the evolution of the language. Why was it such a big deal? In this section, you'll find out why lambdas are so useful and what the syntax of lambda expressions in Kotlin looks like.

1.1. Introduction to lambdas: blocks of code as function parameters

Passing and storing pieces of behavior in your code is a frequent task. For example, you often need to express ideas like "When an event happens, run this handler" or "Apply this operation to all elements in a data structure." In older versions of Java, you could accomplish this through anonymous inner classes. This technique works but requires verbose syntax.

Functional programming offers you another approach to solve this problem: the ability to treat functions as values. Instead of declaring a class and passing an instance of that class to a function, you can pass a function directly. With lambda expressions, the code is even more concise. You don't need to declare a function: instead, you can, effectively, pass a block of code directly as a function parameter.

Let's look at an example. Imagine that you need to define a behavior for clicking a button. You add a listener responsible for handling the click. This listener implements the corresponding `OnClickListener` interface with one method, `onClick`.

Listing 1. Implementing a listener with an anonymous inner class

```
/* Java */
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View view) {
        /* actions on click */
    }
});
```

The verbosity required to declare an anonymous inner class becomes irritating when repeated many times. The notation to express just the behavior—what should be done on clicking—helps eliminate redundant code. In Kotlin, as in Java 8, you can use a lambda.

Listing 2. Implementing a listener with a lambda

```
button.setOnClickListener { /* actions on click */ }
button.setOnClickListener { /* actions on click */ }
```

This Kotlin code does the same thing as an anonymous class in Java but is more concise and readable. We'll discuss the details of this example later in this section.

You saw how a lambda can be used as an alternative to an anonymous object with only one method. Let's now continue with another classical use of lambda expressions: working with collections.

1.2. Lambdas and collections

One of the main tenets of good programming style is to avoid any duplication in your code. Most of the tasks we perform with collections follow a few common patterns, so the code that implements them should live in a library. But without lambdas, it's difficult to provide a good, convenient library for working with collections. Thus if you wrote your code in Java (prior to Java 8), you most likely have a habit of implementing everything on your own. This habit must be changed with Kotlin!

Let's look at an example. You'll use the Person class that contains information about a person's name and age.

```
data class Person(val name: String, val age: Int)
```

Suppose you have a list of people, and you need to find the oldest of them. If you had no experience with lambdas, you might rush to implement the search manually. You'd introduce two intermediate variables—one to hold the maximum age and another to store the first found person of this age—and then iterate over the list, updating these variables.

Listing 3. Searching through a collection manually

```
fun findTheOldest(people: List<Person>) {
    var maxAge = 0                          // Stores the maximum age
    var theOldest: Person? = null           // Stores a person of the maximum age
    for (person in people) {
        if (person.age > maxAge) {          // If the next person is older than the current oldest person, changes the maximum
            maxAge = person.age
            theOldest = person
        }
    }
    println(theOldest)
}
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> findTheOldest(people)
Person(name=Bob, age=31)
```

With enough experience, you can bang out such loops pretty quickly. But there's quite a lot of code here, and it's easy to make mistakes. For example, you might get the comparison wrong and find the minimum element instead of the maximum.

In Kotlin, there's a better way. You can use a library function, as shown next.

Listing 4. Searching through a collection using a lambda

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })      // Finds the maximum by comparing the ages
Person(name=Bob, age=31)
```

The maxBy function can be called on any collection and takes one argument: the function that specifies what values should be compared to find the maximum element. The code in curly braces { it.age } is a lambda implementing that logic. It receives a collection element as an argument (referred to using it) and returns a value to compare. In this example, the collection element is a Person object, and the value to compare is its age, stored in the age property.

If a lambda just delegates to a function or property, it can be replaced by a member reference.

Listing Searching using a member reference

```
people.maxBy(Person::age)
```

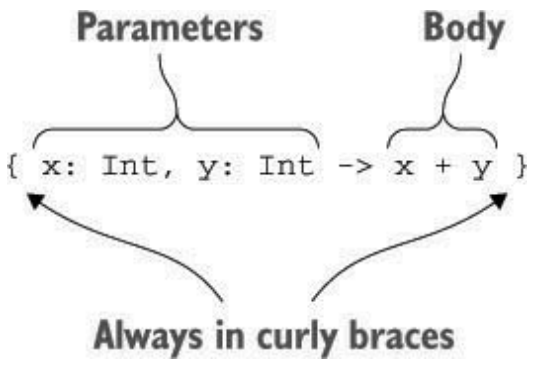
This code means the same thing as listing 5. Section 1.5 will cover the details.

Most of the things we typically do with collections in Java (prior to Java 8) can be better expressed with library functions taking lambdas or member references. The resulting code is much shorter and easier to understand. To help you start getting used to it, let's look at the syntax for lambda expressions.

1.3. Syntax for lambda expressions

As we've mentioned, a lambda encodes a small piece of behavior that you can pass around as a value. It can be declared independently and stored in a variable. But more frequently, it's declared directly when passed to a function. Figure 1 shows the syntax for declaring lambda expressions.

Figure 1. Lambda expression syntax



A lambda expression in Kotlin is always surrounded by curly braces. Note that there are no parentheses around the arguments. The arrow separates the argument list from the body of the lambda.

You can store a lambda expression in a variable and then treat this variable like a normal function (call it with the corresponding arguments):

```
>>> val sum = { x: Int, y: Int -> x + y }
>>> println(sum(1, 2))
3
```

Calls the lambda stored in a variable

If you want to, you can call the lambda expression directly:

```
>>> { println(42) }()
42
```

But such syntax isn't readable and doesn't make much sense (it's equivalent to executing the lambda body directly). If you need to enclose a piece of code in a block, you can use the library function run that executes the lambda passed to it:

```
>>> run { println(42) }
42
```

Runs the code in the lambda

Let's return to listing 4, which finds the oldest person in a list:

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })
Person(name=Bob, age=31)
```

If you rewrite this example without using any syntax shortcuts, you get the following:

```
people.maxBy({ p: Person -> p.age })
```

It should be clear what happens here: the piece of code in curly braces is a lambda expression, and you pass it as an argument to the function. The lambda expression takes one argument of type Person and returns its age.

But this code is verbose. First, there's too much punctuation, which hurts readability. Second, the type can be inferred from the context and therefore omitted. Last, you don't need to assign a name to the lambda argument in this case.

Let's make these improvements, starting with braces. In Kotlin, a syntactic convention lets you move a lambda expression out of parentheses if it's the last argument in a function call. In this example, the lambda is the only argument, so it can be placed after the parentheses:

```
people.maxBy() { p: Person -> p.age }
```

When the lambda is the only argument to a function, you can also remove the empty parentheses from the call:

```
people.maxBy { p: Person -> p.age }
```

All three syntactic forms mean the same thing, but the last one is the easiest to read. If a lambda is the only argument, you'll definitely want to write it without the parentheses. When you have several arguments, you can emphasize that the lambda is an argument by leaving it inside the parentheses, or you can put it outside of them—both options are valid. If you want to pass two or more lambdas, you can't move more than one out, so it's usually better to pass them using the regular syntax.

It's also defined in the Kotlin standard library, with the difference that the standard library version takes a function as an additional parameter. This function can be used to convert an element to a string differently than the toString function. Here's how you can use it to print names only.

Listing 6. Passing a lambda as a named argument

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> val names = people.joinToString(separator = " ",
...                         transform = { p: Person -> p.name })
>>> println(names)
Alice Bob
```

And here's how you can rewrite that call with the lambda outside the parentheses.

Listing 7. Passing a lambda outside of parentheses

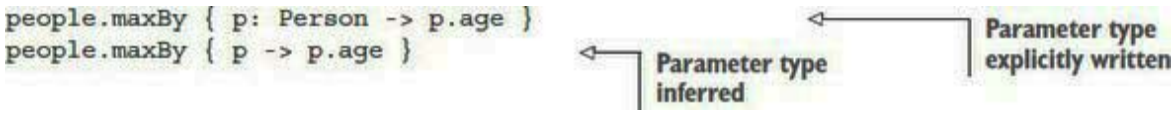
```
people.joinToString(" ") { p: Person -> p.name }
```

Listing 7 uses a named argument to pass the lambda, making it clear what the lambda is used for. Listing 8 is more concise, but it doesn't express explicitly what the lambda is used for, so it may be harder to understand for people not familiar with the function being called.

IntelliJ IDEA tip
To convert one syntactic form to the other, you can use the actions: "Move lambda expression out of parentheses" and "Move lambda expression into parentheses."

Let's move on with simplifying the syntax and get rid of the parameter type.

Listing 8. Omitting lambda parameter type



As with local variables, if the type of a lambda parameter can be inferred, you don't need to specify it explicitly. With the `maxBy` function, the parameter type is always the same as the collection element type. The compiler knows you're calling maxBy on a collection of `Person` objects, so it can understand that the lambda parameter will also be of type `Person`.

There are cases when the compiler can't infer the lambda parameter type, but we won't discuss them here. The simple rule you can follow is to always start without the types; if the compiler complains, specify them.

You can specify only some of the argument types while leaving others with just names. Doing so may be convenient if the compiler can't infer one of the types or if an explicit type improves readability.

The last simplification you can make in this example is to replace a parameter with the default parameter name: it. This name is generated if the context expects a lambda with only one argument, and its type can be inferred.

Listing 9. Using the default parameter name

```
people.maxBy { it.age }
```

"it" is an autogenerated parameter name.

This default name is generated only if you don't specify the argument name explicitly.

> Note
> The it convention is great for shortening your code, but you shouldn't abuse it. In particular, in the case of nested lambdas, it's better to declare the parameter of each lambda explicitly; otherwise it's difficult to understand which value the it refers to. It's useful also to declare parameters explicitly if the meaning or the type of the parameter isn't clear from the context.

If you store a lambda in a variable, there's no context from which to infer the parameter types, so you have to specify them explicitly:

```
>>> val getAge = { p: Person -> p.age }
>>> people.maxBy(getAge)
```

So far, you've only seen examples with lambdas that consist of one expression or statement. But lambdas aren't constrained to such a small size and can contain multiple statements. In this case, the last expression is the result:

```
>>> val sum = { x: Int, y: Int ->
...     println("Computing the sum of $x and $y...")
...     x + y
... }
>>> println(sum(1, 2))
Computing the sum of 1 and 2...
3
```

Next, let's talk about a concept that often goes side-by-side with lambda expressions: capturing variables from the context.

1.4. Accessing variables in scope

You know that when you declare an anonymous inner class in a function, you can refer to parameters and local variables of that function from inside the class. With lambdas, you can do exactly the same thing. If you use a lambda in a function, you can access the parameters of that function as well as the local variables declared before the lambda.

To demonstrate this, let's use the forEach standard library function. It's one of the most basic collection-manipulation functions; all it does is call the given lambda on every element in the collection. The forEach function is somewhat more concise than a regular for loop, but it doesn't have many other advantages, so you needn't rush to convert all your loops to lambdas.

The following listing takes a list of messages and prints each message with the same prefix.

Listing 10. Using function parameters in a lambda

```
fun printMessagesWithPrefix(messages: Collection<String>, prefix: String) {
    messages.forEach {
        println("$prefix $it")
    }
}
```

Accesses the "prefix" parameter in the lambda

Takes as an argument a lambda specifying what to do with each element

```
>>> val errors = listOf("403 Forbidden", "404 Not Found")
>>> printMessagesWithPrefix(errors, "Error:")
Error: 403 Forbidden
Error: 404 Not Found
```

One important difference between Kotlin and Java is that in Kotlin, you aren't restricted to accessing final variables. You can also modify variables from within a lambda. The next listing counts the number of client and server errors in the given set of response status codes.

Listing 11. Changing local variables from a lambda

```
fun printProblemCounts(responses: Collection<String>) {
    var clientErrors = 0                                  // Declares variables that will be accessed from the lambda
    var serverErrors = 0
    responses.forEach {
        if (it.startsWith("4")) {
            clientErrors++                                // Modifies variables in the lambda
        } else if (it.startsWith("5")) {
            serverErrors++
        }
    }
    println("$clientErrors client errors, $serverErrors server errors")
}

>>> val responses = listOf("200 OK", "418 I'm a teapot",
...                        "500 Internal Server Error")
>>> printProblemCounts(responses)
1 client errors, 1 server errors
```

Kotlin, unlike Java, allows you to access non-final variables and even modify them in a lambda. External variables accessed from a lambda, such as `prefix`, `clientErrors`, and `serverErrors` in these examples, are said to be *captured* by the lambda.

Note that, by default, the lifetime of a local variable is constrained by the function in which the variable is declared. But if it's captured by the lambda, the code that uses this variable can be stored and executed later. You may ask how this works. When you capture a final variable, its value is stored together with the lambda code that uses it. For non-final variables, the value is enclosed in a special wrapper that lets you change it, and the reference to the wrapper is stored together with the lambda.

Capturing a mutable variable: implementation details

Java allows you to capture only final variables. When you want to capture a mutable variable, you can use one of the following tricks: either declare an array of one element in which to store the mutable value, or create an instance of a wrapper class that stores the reference that can be changed. If you used this technique explicitly in Kotlin, the code would be as follows:

```
class Ref<T>(var value: T)                                // Class used to simulate capturing a mutable variable
>>> val counter = Ref(0)
>>> val inc = { counter.value++ }                         // Formally, an immutable variable is captured; but the actual value is stored in a field and can be changed.
```

In real code, you don't need to create such wrappers. Instead, you can mutate the variable directly:

```
var counter = 0
val inc = { counter++ }
```

How does it work? The first example shows how the second example works under the hood. Any time you capture a final variable (val), its value is copied, as in Java. When you capture a mutable variable (var), its value is stored as an instance of a Ref class. The Ref variable is final and can be easily captured, whereas the actual value is stored in a field and can be changed from the lambda.

An important caveat is that, if a lambda is used as an event handler or is otherwise executed asynchronously, the modifications to local variables will occur only when the lambda is executed. For example, the following code isn't a correct way to count button clicks:

```
fun tryToCountButtonClicks(button: Button): Int {
    var clicks = 0
    button.onClick { clicks++ }
    return clicks
}
```

This function will always return `0`. Even though the `onClick` handler will modify the value of clicks, you won't be able to observe the modification, because the `onClick` handler will be called after the function returns. A correct implementation of the function would need to store the click count not in a local variable, but in a location that remains accessible outside of the function—for example, in a property of a class.

We've discussed the syntax for declaring lambdas and how variables are captured in lambdas. Now let's talk about member references, a feature that lets you easily pass references to existing functions.

1.5. Member references

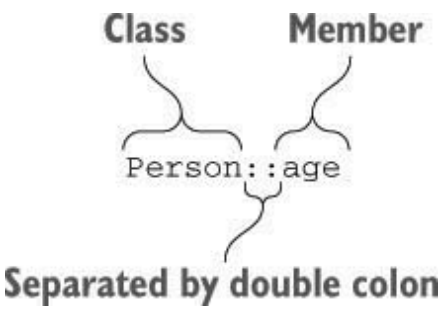
You've seen how lambdas allow you to pass a block of code as a parameter to a function. But what if the code that you need to pass as a parameter is already defined as a function? Of course, you can pass a lambda that calls that function, but doing so is somewhat redundant. Can you pass the function directly?

In Kotlin, just like in Java 8, you can do so if you convert the function to a value. You use the `::` operator for that:

```
val getAge = Person::age
```

This expression is called *member reference*, and it provides a short syntax for creating a function value that calls exactly one method or accesses a property. A double colon separates the name of a class from the name of the member you need to reference (a method or property), as shown in figure 2.

Figure 2. Member reference syntax



This is a more concise expression of a lambda that does the same thing:

```
val getAge = { person: Person -> person.age }
```

Note that, regardless of whether you're referencing a function or a property, you shouldn't put parentheses after its name in a member reference.

A member reference has the same type as a lambda that calls that function, so you can use the two interchangeably:

```
people.maxBy(Person::age)
```

You can have a reference to a function that's declared at the top level (and isn't a member of a class), as well:

```
fun salute() = println("Salute!")
>>> run(::salute)
Salute!
```

Reference to the top-level function

In this case, you omit the class name and start with `::`. The member reference `::salute` is passed as an argument to the library function run, which calls the corresponding function.

It's convenient to provide a member reference instead of a lambda that delegates to a function taking several parameters:

```
val action = { person: Person, message: String ->
    sendEmail(person, message)
}
val nextAction = ::sendEmail
```

This lambda delegates [illegible] function.

You can reference instead.

You can store or postpone the action of creating an instance of a class using a *constructor reference*. The constructor reference is formed by specifying the class name after the double colons:

```
data class Person(val name: String, val age: Int)

>>> val createPerson = ::Person
>>> val p = createPerson("Alice", 29)
>>> println(p)
Person(name=Alice, age=29)
```

An action of creating an instance of "Person" is saved as a value.

Note that you can also reference extension functions the same way:

```
fun Person.isAdult() = age >= 21
val predicate = Person::isAdult
```

Although `isAdult` isn't a member of the Person class, you can access it via reference, just as you can access it as a member on an instance: `person.isAdult()`.

Bound references
In Kotlin 1.0, when you take a reference to a method or property of a class, you always need to provide an instance of that class when you call the reference. Support for bound member references, which allow you to use the member-reference syntax to capture a reference to the method on a specific object instance, is planned for Kotlin 1.1:

```
>>> val p = Person("Dmitry", 34)
>>> val personsAgeFunction = Person::age
>>> println(personsAgeFunction(p))
34
>>> val dmitrysAgeFunction = p::age
>>> println(dmitrysAgeFunction())
34
```

A bound member reference that you can use in Kotlin 1.1

Note that `personsAgeFunction` is a one-argument function (it returns the age of a given person), whereas `dmitrysAgeFunction` is a zero-argument function (it returns the age of a specific person). Before Kotlin 1.1, you needed to write the lambda `{ p.age }` explicitly instead of using the bound member reference `p::age`.

In the following section, we'll look at many library functions that work great with lambda expressions, as well as member references.

2. Functional APIs for collections

Functional style provides many benefits when it comes to manipulating collections. You can use library functions for the majority of tasks and simplify your code. In this section, we'll discuss some of the functions in the Kotlin standard library for working with collections. We'll start with staples like `filter` and `map` and the concepts behind them. We'll also cover other useful functions and give you tips about how not to overuse them and how to write clear and comprehensible code.

Note that none of these functions were invented by the designers of Kotlin. These or similar functions are available for all languages that support lambdas, including C#, Groovy, and Scala. If you're already familiar with these concepts, you can quickly look through the following examples and skip the explanations.

2.1. Essentials: filter and map

The `filter` and `map` functions form the basis for manipulating collections. Many collection operations can be expressed with their help.

For each function, we'll provide one example with numbers and one using the familiar `Person` class:

```
data class Person(val name: String, val age: Int)
```

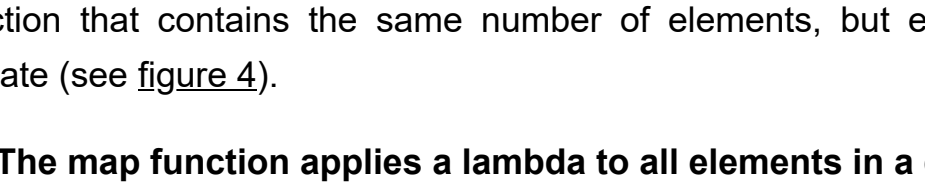
The `filter` function goes through a collection and selects the elements for which the given lambda returns `true`:

```
>>> val list = listOf(1, 2, 3, 4)
>>> println(list.filter { it % 2 == 0 })
[2, 4]
```

Only even numbers remain.

The result is a new collection that contains only the elements from the input collection that satisfy the predicate, as illustrated in figure 3.

Figure 3. The filter function selects elements matching given predicate



If you want to keep only people older than 30, you can use `filter`:

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.filter { it.age > 30 })
[Person(name=Bob, age=31)]
```

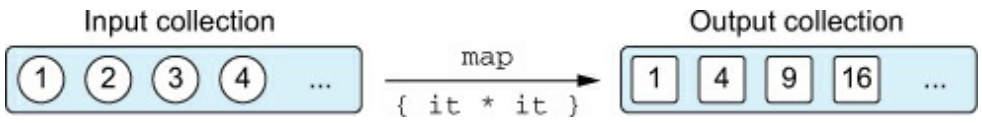
The `filter` function can remove unwanted elements from a collection, but it doesn't change the elements. Transforming elements is where map comes into play.

The map function applies the given function to each element in the collection and collects the results into a new collection. You can transform a list of numbers into a list of their squares, for example:

```
>>> val list = listOf(1, 2, 3, 4)
>>> println(list.map { it * it })
[1, 4, 9, 16]
```

The result is a new collection that contains the same number of elements, but each element is transformed according to the given predicate (see figure 4).

Figure 4. The map function applies a lambda to all elements in a collection



If you want to print just a list of names, not a list of people, you can transform the list using `map`:

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.map { it.name })
[Alice, Bob]
```

Note that this example can be nicely rewritten using member references:

```
people.map(Person::name)
```

You can easily chain several calls like that. For example, let's print the names of people older than 30:

```
>>> people.filter { it.age > 30 }.map(Person::name)
[Bob]
```

Now, let's say you need the names of the oldest people in the group. You can find the maximum age of the people in the group and return everyone who is that age. It's easy to write such code using lambdas:

```
people.filter { it.age == people.maxBy(Person::age).age }
```

But note that this code repeats the process of finding the maximum age for every person, so if there are 100 people in the collection, the search for the maximum age will be performed 100 times!

The following solution improves on that and calculates the maximum age only once:

```
val maxAge = people.maxBy(Person::age).age
people.filter { it.age == maxAge }
```

Don’t repeat a calculation if you don’t need to! Simple-looking code using lambda expressions can sometimes obscure the complexity of the underlying operations. Always keep in mind what is happening in the code you write.

You can also apply the `filter` and transformation functions to maps:

```
>>> val numbers = mapOf(0 to "zero", 1 to "one")
>>> println(numbers.mapValues { it.value.toUpperCase() })
{0=ZERO, 1=ONE}
```

There are separate functions to handle keys and values. `filterKeys` and `mapKeys` filter and transform the keys of a map, respectively, where as `filterValues` and `mapValues` filter and transform the corresponding values.

2.2. “all”, “any”, “count”, and “find”: applying a predicate to a collection

Another common task is checking whether all elements in a collection match a certain condition (or, as a variation, whether any elements match). In Kotlin, this is expressed through the all and any functions. The count function checks how many elements satisfy the predicate, and the find function returns the first matching element.

To demonstrate those functions, let’s define the predicate `canBeInClub27` to check whether a person is 27 or younger:

```
val canBeInClub27 = { p: Person -> p.age <= 27 }
```

If you’re interested in whether all the elements satisfy this predicate, you use the all function:

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.all(canBeInClub27))
false
```

If you need to check whether there’s at least one matching element, use `any`:

```
>>> println(people.any(canBeInClub27))
true
```

Note that !all (“not all”) with a condition can be replaced with any with a negation of that condition, and vice versa. To make your code easier to understand, you should choose a function that doesn’t require you to put a negation sign before it:

```
>>> val list = listOf(1, 2, 3)
>>> println(!list.all { it == 3 })
true
>>> println(list.any { it != 3 })
true
```

The negation ! isn’t noticeable, so it’s better to use “any” in this case.

The condition in the argument has changed to its opposite.

The first check ensures that not all elements are equal to 3. That’s the same as having at least one non-3, which is what you check using any on the second line.

If you want to know how many elements satisfy this predicate, use `count`:

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.count(canBeInClub27))
1
```

Using the right function for the job: “count” vs. “size”

It’s easy to forget about `count` and implement it by filtering the collection and getting its size:

```
>>> println(people.filter(canBeInClub27).size)
1
```

But in this case, an intermediate collection is created to store all the elements that satisfy the predicate. On the other hand, the count method tracks only the number of matching elements, not the elements themselves, and is therefore more efficient.

As a general rule, try to find the most appropriate operation that suits your needs.

To find an element that satisfies the predicate, use the `find` function:

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.find(canBeInClub27))
Person(name=Alice, age=27)
```

This returns the first matching element if there are many or `null` if nothing satisfies the predicate. A synonym of `find` is `firstOrNull`, which you can use if it expresses the idea more clearly for you.

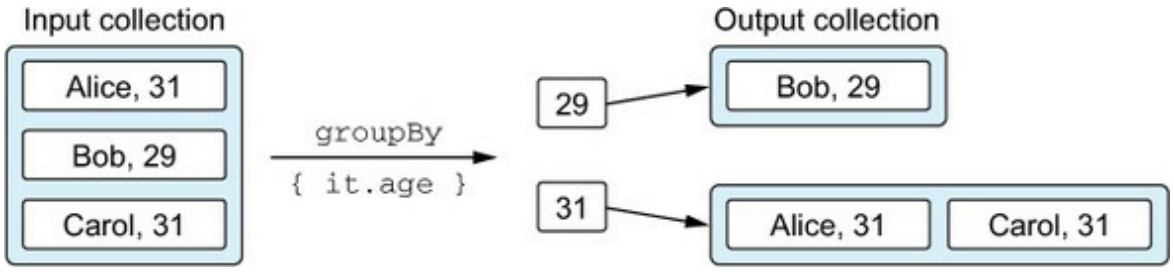
2.3. groupBy: converting a list to a map of groups

Imagine that you need to divide all elements into different groups according to some quality. For example, you want to group people of the same age. It’s convenient to pass this quality directly as a parameter. The `groupBy` function can do this for you:

```
>>> val people = listOf(Person("Alice", 31),
...         Person("Bob", 29), Person("Carol", 31))
>>> println(people.groupBy { it.age })
```

The result of this operation is a map from the key by which the elements are grouped (age, in this case) to the groups of elements (persons); see figure 5.

Figure 5. The result of applying the groupBy function



For this example, the output is as follows:

```
{29=[Person(name=Bob, age=29)],
 31=[Person(name=Alice, age=31), Person(name=Carol, age=31)]}
```

Each group is stored in a list, so the result type is Map `<Int, List<Person>>`. You can do further modifications with this map, using functions such as mapKeys and `mapValues`.

As another example, let’s see how to group strings by their first character using member references:

```
>>> val list = listOf("a", "ab", "b")
>>> println(list.groupBy(String::first))
{a=[a, ab], b=[b]}
```

Note that `first` here isn't a member of the `String` class, it's an extension. Nevertheless, you can access it as a member reference.

2.4. flatMap and flatten: processing elements in nested collections

Now let's put aside our discussion of people and switch to books. Suppose you have a storage of books, represented by the class `Book`:

```
class Book(val title: String, val authors: List<String>)
```

Each book was written by one or more authors. You can compute the set of all the authors in your library:

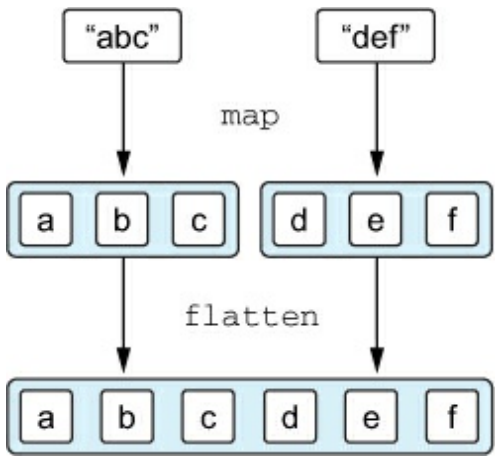
```
books.flatMap { it.authors }.toSet()
```

← Set of all authors who wrote books in the "books" collection

The flatMap function does two things: At first it transforms (or *maps*) each element to a collection according to the function given as an argument, and then it combines (or *flattens*) several lists into one. An example with strings illustrates this concept well (see figure 6):

```
>>> val strings = listOf("abc", "def")
>>> println(strings.flatMap { it.toList() })
[a, b, c, d, e, f]
```

Figure 6. The result of applying the flatMap function



The `toList` function on a string converts it into a list of characters. If you used the `map` function together with `toList`, you'd get a list of lists of characters, as shown in the second row in the figure. The `flatMap` function does the following step as well, and returns one list consisting of all the elements.

Let's return to the authors:

```
>>> val books = listOf(Book("Thursday Next", listOf("Jasper Fforde")),
...                    Book("Mort", listOf("Terry Pratchett")),
...                    Book("Good Omens", listOf("Terry Pratchett",
...                                              "Neil Gaiman")))
>>> println(books.flatMap { it.authors }.toSet())
[Jasper Fforde, Terry Pratchett, Neil Gaiman]
```

Each book can be written by multiple authors, and the `book.authors` property stores the collection of authors. The `flatMap` function combines the authors of all the books in a single, flat list. The `toSet` call removes duplicates from the resulting collection—so, in this example, Terry Pratchett is listed only once in the output.

You may think of `flatMap` when you're stuck with a collection of collections of elements that have to be combined into one. Note that if you don't need to transform anything and just need to flatten such a collection, you can use the `flatten` function: `listOfLists.flatten()`.

We've highlighted a few of the collection operation functions in the Kotlin standard library, but there are many more. We won't cover them all, for reasons of space, and also because showing a long list of functions is boring. Our general advice when you write code that works with collections is to think of how the operation could be expressed as a general transformation, and to look for a library function that performs such a transformation. It's likely that you'll be able to find one and use it to solve your problem more quickly than with a manual implementation.

Now let's take a closer look at the performance of code that chains collection operations. In the next section, you'll see the different ways in which such operations can be executed.

3. Lazy collection operations: sequences

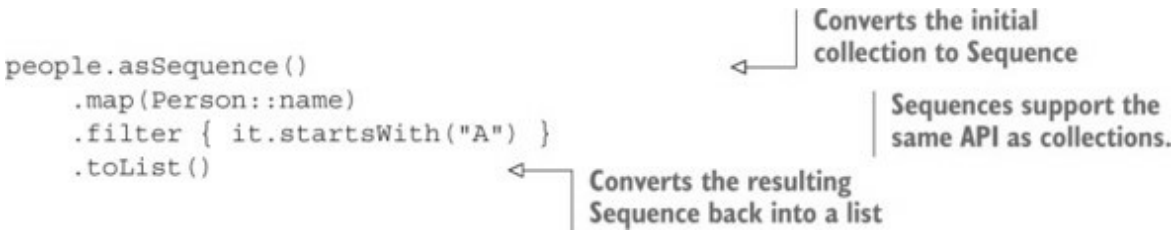
In the previous section, you saw several examples of chained collection functions, such as `map` and `filter`. These functions create intermediate collections *eagerly*, meaning the intermediate result of each step is stored in a temporary list. *Sequences* give you an alternative way to perform such computations that avoids the creation of intermediate temporary objects.

Here's an example:

```
people.map(Person::name).filter { it.startsWith("A") }
```

The Kotlin standard library reference says that both `filter` and `map` return a list. That means this chain of calls will create two lists: one to hold the results of the `filter` function and another for the results of `map`. This isn't a problem when the source list contains two elements, but it becomes much less efficient if you have a million.

To make this more efficient, you can convert the operation so it uses sequences instead of using collections directly:



The result of applying this operation is the same as in the previous example: a list of people's names that start with the letter *A*. But in the second example, no intermediate collections to store the elements are created, so performance for a large number of elements will be noticeably better.

The entry point for lazy collection operations in Kotlin is the Sequence interface. The interface represents just that: a sequence of elements that can be enumerated one by one. Sequence provides only one method, iterator, that you can use to obtain the values from the sequence.

The strength of the Sequence interface is in the way operations on it are implemented. The elements in a sequence are evaluated lazily. Therefore, you can use sequences to efficiently perform chains of operations on elements of a collection without creating collections to hold intermediate results of the processing.

You can convert any collection to a sequence by calling the extension function asSequence. You call toList for backward conversion.

Why do you need to convert the sequence back to a collection? Wouldn't it be more convenient to use sequences instead of collections, if they're so much better? The answer is: sometimes. If you only need to iterate over the elements in a sequence, you can use the sequence directly. If you need to use other API methods, such as accessing the elements by index, then you need to convert the sequence to a list.

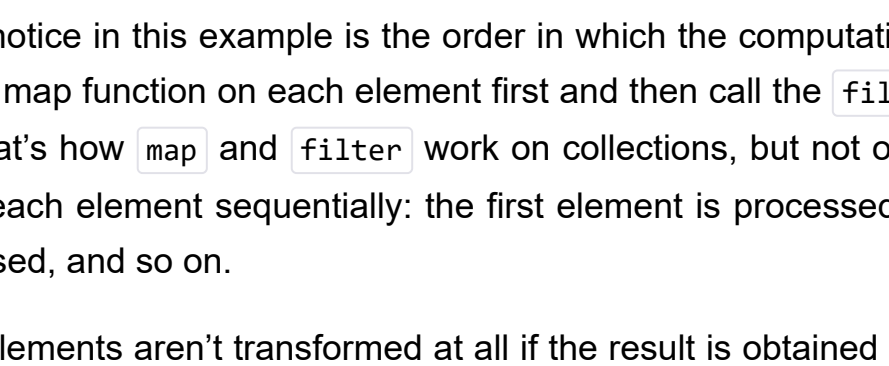
> Note
> As a rule, use a sequence whenever you have a chain of operations on a *large* collection. But if the collection contains a large number of elements, the intermediate rearranging of elements costs a lot, so lazy evaluation is preferable.

Because operations on a sequence are lazy, in order to perform them, you need to iterate over the sequence's elements directly or by converting it to a collection. The next section explains that.

3.1. Executing sequence operations: intermediate and terminal operations

Operations on a sequence are divided into two categories: intermediate and terminal. An *intermediate operation* returns another sequence, which knows how to transform the elements of the original sequence. A *terminal operation* returns a result, which may be a collection, an element, a number, or any other object that's somehow obtained by the sequence of transformations of the initial collection (see figure 7).

Figure 7. Intermediate and terminal operations on sequences



Intermediate operations are always lazy. Look at this example, where the terminal operation is missing:

Executing this code snippet prints nothing to the console. That means the `map` and `filter` transformations are postponed and will be applied only when the result is obtained (that is, when the terminal operation is called):

```
>>> listOf(1, 2, 3, 4).asSequence()
...         .map { print("map($it) "); it * it }
...         .filter { print("filter($it) "); it % 2 == 0 }
...         .toList()
map(1) filter(1) map(2) filter(4) map(3) filter(9) map(4) filter(16)
```

The terminal operation causes all the postponed computations to be performed.

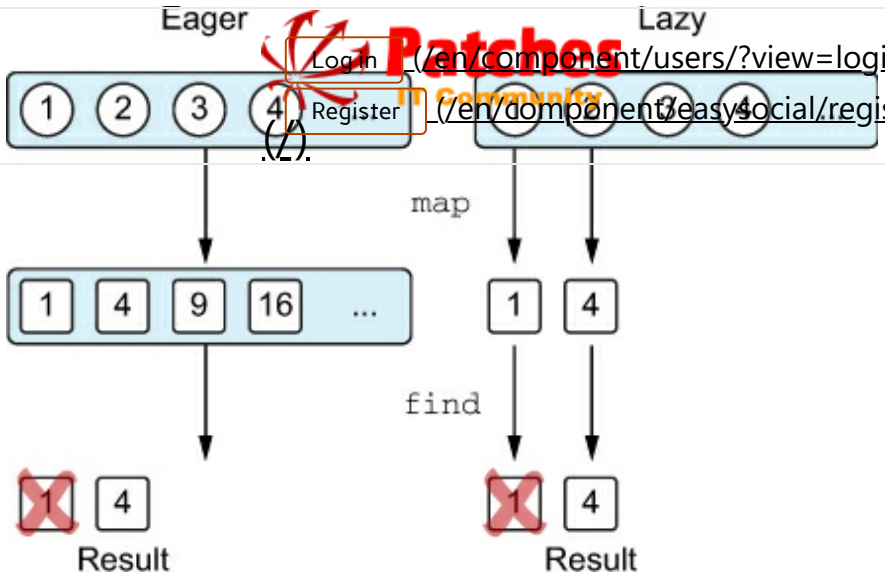
One more important thing to notice in this example is the order in which the computations are performed. The naive approach would be to call the map function on each element first and then call the `filter` function on each element of the resulting sequence. That's how `map` and `filter` work on collections, but not on sequences. For sequences, all operations are applied to each element sequentially: the first element is processed (mapped, then filtered), then the second element is processed, and so on.

This approach means some elements aren't transformed at all if the result is obtained before they are reached. Let's look at an example with `map` and `find` operations. First you map a number to its square, and then you find the first item that's greater than 3:

```
>>> println(listOf(1, 2, 3, 4).asSequence()
                               .map { it * it }.find { it > 3 })
4
```

If the same operations are applied to a collection instead of a sequence, then the result of map is evaluated first, transforming all elements in the initial collection. In the second step, an element satisfying the predicate is found in the intermediate collection. With sequences, the lazy approach means you can skip processing some of the elements. Figure 8 illustrates the difference between evaluating this code in an eager (using collections) and lazy (using sequences) manner.

Figure 8. Eager evaluation runs each operation on the entire collection; lazy evaluation processes elements one by one



In the first case, when you work with collections, the list is transformed into another list, so the map transformation is applied to each element, including 3 and 4. Afterward, the first element satisfying the predicate is found: the square of 2.

In the second case, the find call begins processing the elements one by one. You take a number from the original sequence, transform it with map, and then check whether it matches the predicate passed to `find`. When you reach 2, you see that its square is greater than 3 and return it as the result of the `find` operation. You don't need to look at 3 and 4, because the result was found before you reached them.

The order of the operations you perform on a collection can affect performance as well. Imagine that you have a collection of people, and you want to print their names if they're shorter than a certain limit. You need to do two things: map each person to their name, and then filter out those names that aren't short enough. You can apply map and `filter` operations in any order in this case. Both approaches give the same result, but they differ in the total number of transformations that should be performed (see figure 9).

Figure 9. Applying filter first helps to reduce the total number of transformations

If map goes first, each element is transformed. If you apply filter first, inappropriate elements are filtered out as soon as possible and aren't transformed.

> Streams vs. sequences
> If you're familiar with Java 8 streams, you'll see that sequences are exactly the same concept. Kotlin provides its own version of the same concept because Java 8 streams aren't available on platforms built on older versions of Java, such as Android. If you're targeting Java 8, streams give you one big feature that isn't currently implemented for Kotlin collections and sequences: the ability to run a stream operation (such as `map` or `filter`) on multiple CPUs in parallel. You can choose between streams and sequences based on the Java versions you target and your specific requirements.

3.2. Creating sequences

The previous examples used the same method to create a sequence: you called as-Sequence() on a collection. Another possibility is to use the generateSequence function. This function calculates the next element in a sequence given the previous one. For example, here's how you can use generateSequence to calculate the sum of all natural

numbers up to 100.

Listing 12. Generating and using a sequence of natural numbers (Z).

Note that naturalNumbers and numbersTo100 in this example are both sequences with postponed computation. The actual numbers in those sequences won't be evaluated until you call the terminal operation (sum in this case).

Another common use case is a sequence of parents. If an element has parents of its own type (such as a human being or a Java file), you may be interested in qualities of the sequence of all of its ancestors. In the following example, you inquire whether the file is located in a hidden directory by generating a sequence of its parent directories and checking this attribute on each of the directories.

Listing 13. Generating and using a sequence of parent directories

```
fun File.isInsideHiddenDirectory() =
        generateSequence(this) { it.parentFile }.any { it.isHidden }

>>> val file = File("/Users/svtk/.HiddenDir/a.txt")
>>> println(file.isInsideHiddenDirectory())
true
```

Once again, you generate a sequence by providing the first element and a way to get each subsequent element. By replacing any with `find`, you'll get the desired directory. Note that using sequences allows you to stop traversing the parents as soon as you find the required directory.

We've thoroughly discussed a frequently used application of lambda expressions: using them to simplify manipulating collections. Now let's continue with another important topic: using lambdas with an existing Java API.

4. Using Java functional interfaces

Using lambdas with Kotlin libraries is nice, but the majority of APIs that you work with are probably written in Java, not Kotlin. The good news is that Kotlin lambdas are fully interoperable with Java APIs; in this section, you'll see exactly how this works.

At the beginning of the chapter, you saw an example of passing a lambda to a Java method:

The Button class sets a new listener to a button via an `setOnClickListener` method that takes an argument of type `OnClickListener`:

```
/* Java */
public class Button {
    public void setOnClickListener(OnClickListener l) { ... }
}
The OnClickListener interface declares one method, onClick:

/* Java */
public interface OnClickListener {
    void onClick(View v);
}
```

In Java (prior to Java 8), you have to create a new instance of an anonymous class to pass it as an argument to the `setOnClickListener` method:

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        ...
    }
}
```

In Kotlin, you can pass a lambda instead:

```
button.setOnClickListener { view -> ... }
```

The lambda used to implement `OnClickListener` has one parameter of type View, as in the `onClick` method. The mapping is illustrated in figure 10.

Figure 10. Parameters of the lambda correspond to method parameters.

is works because the `OnClickListener` interface has only one abstract method. Such interfaces are called *functional interfaces*, or *SAM interfaces*, where SAM stands for *single abstract method*. The Java API is full of functional interfaces like `Runnable` and `Callable`, as well as methods working with them. Kotlin allows you to use lambdas when calling Java methods that take functional interfaces as parameters, ensuring that your Kotlin code remains clean and idiomatic.

> Note
> Unlike Java, Kotlin has proper function types. Because of that, Kotlin functions that need to take lambdas as parameters should use function types, not functional interface types, as the types of those parameters. Automatic conversion of lambdas to objects implementing Kotlin interfaces isn't supported.

Let's look in detail at what happens when you pass a lambda to a method that expects an argument of a functional interface type.

4.1. Passing a lambda as a parameter to a Java method

You can pass a lambda to any Java method that expects a functional interface. For example, consider this method, which has a parameter of type `Runnable`:

```
/* Java */
void postponeComputation(int delay, Runnable computation);
```

In Kotlin, you can invoke it and pass a lambda as an argument. The compiler will automatically convert it into an instance of `Runnable`:

```
postponeComputation(1000) { println(42) }
```

Note that when we say "an instance of Runnable," what we mean is "an instance of an anonymous class implementing Runnable." The compiler will create that for you and will use the lambda as the body of the single abstract method—the run method, in this case.

You can achieve the same effect by creating an anonymous object that implements Runnable explicitly:

But there's a difference. When you explicitly declare an object, a new instance is created on each invocation. With a lambda, the situation is different: if the lambda doesn't access any variables from the function where it's defined, the corresponding anonymous class instance is reused between calls:

Therefore, the equivalent implementation with an explicit object declaration is the following snippet, which stores the Runnable instance in a variable and uses it for every invocation:

If the lambda captures variables from the surrounding scope, it's no longer possible to reuse the same instance for every invocation. In that case, the compiler creates a new object for every call and stores the values of the captured variables in that object. For example, in the following function, every invocation uses a new Runnable instance, storing the id value as a field:

Lambda implementation details

As of Kotlin 1.0, every lambda expression is compiled into an anonymous class, unless it's an inline lambda. Support for generating Java 8 bytecode is planned for later versions of Kotlin. Once implemented, it will allow the compiler to avoid generating a separate .class file for every lambda expression.

If a lambda captures variables, the anonymous class will have a field for each captured variable, and a new instance of that class will be created for every invocation.

Otherwise, a single instance will be created. The name of the class is derived by adding a suffix from the name of the function in which the lambda is declared: `HandleComputation$1`, for this example.

Here's what you'll see if you decompile the code of the previous lambda expression:

As you can see, the compiler generates a field and a constructor parameter for each captured variable.

> Note that the discussion of creating an anonymous class and an instance of this class for a lambda is valid for Java methods expecting functional interfaces, but does not apply to working with collections using Kotlin extension methods. If you pass a lambda to the Kotlin function that's marked `inline`, no anonymous classes are created. And most of the library functions are marked `inline`.

As you've seen, in most cases the conversion of a lambda to an instance of a functional interface happens automatically, without any effort on your part. But there are cases when you need to perform the conversion explicitly. Let's see how to do that.

4.2. SAM constructors: explicit conversion of lambdas to functional interfaces

A *SAM constructor* is a compiler-generated function that lets you perform an explicit conversion of a lambda into an instance of a functional interface. You can use it in contexts when the compiler doesn't apply the conversion automatically. For instance, if you have a method that returns an instance of a functional interface, you can't return a lambda directly; you need to wrap it into a SAM constructor. Here's a simple example.

Listing 14. Using a SAM constructor to return a value

```
fun createAllDoneRunnable(): Runnable {
    return Runnable { println("All done!") }
}

>>> createAllDoneRunnable().run()
All done!
```

The name of the SAM constructor is the same as the name of the underlying functional interface. The SAM constructor takes a single argument—a lambda that will be used as the body of the single abstract method in the functional interface—and returns an instance of the class implementing the interface.

In addition to returning values, SAM constructors are used when you need to store a functional interface instance generated from a lambda in a variable. Suppose you want to reuse one listener for several buttons, as in the following listing (in an Android application, this code can be a part of the `Activity.onCreate method`).

Listing 15. Using a SAM constructor to reuse a listener instance

`listener` checks which button was the source of the click and behaves accordingly. You could define a listener by using an object declaration that implements `OnClickListener`, but SAM constructors give you a more concise option.

> Lambdas and adding/removing listeners
> Note that there's no this in a lambda as there is in an anonymous object: there's no way to refer to the anonymous class instance into which the lambda is converted. From the compiler's point of view, the lambda is a block of code, not an object, and you can't refer to it as an object. The this reference in a lambda refers to a surrounding class.
> If your event listener needs to unsubscribe itself while handling an event, you can't use a lambda for that. Use an anonymous object to implement a listener, instead. In an anonymous object, the this keyword refers to the instance of that object, and you can pass it to the API that removes the listener.

Also, even though SAM conversion in method calls typically happens automatically, there are cases when the compiler can't choose the right overload when you pass a lambda as an argument to an overloaded method. In those cases, applying an explicit SAM constructor is a good way to resolve the compilation error.

To finish our discussion of lambda syntax and usage, let's look at lambdas with receivers and how they're used to define convenient library functions that look like built-in constructs.

5. Lambdas with receivers: "with" and "apply"

This section demonstrates the with and apply functions from the Kotlin standard library. These functions are convenient, and you'll find many uses for them even without understanding how they're declared. The explanations in this section, however, help you become familiar with a unique feature of Kotlin's lambdas that isn't available with Java: the ability to call methods of a different object in the body of a lambda without any additional qualifiers. Such lambdas are called *lambdas with receivers*. Let's begin by looking at the with function, which uses a lambda with a receiver.

5.1. The "with" function

Many languages have special statements you can use to perform multiple operations on the same object without repeating its name. Kotlin also has this facility, but it's provided as a library function called with, not as a special language construct.

To see how it can be useful, consider the following example, which you'll then refactor using with.

Listing 16. Building the alphabet


```
fun alphabet(): String {
    val result = StringBuilder()
    for (letter in 'A'..'Z') {
         result.append(letter)
    }
    result.append("\nNow I know the alphabet!")
    return result.toString()
}
>>> println(alphabet())
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Now I know the alphabet!
```

In this example, you call several different methods on the `result` instance and repeating the result name in each call. This isn't too bad, but what if the expression you were using was longer or repeated more often?

Here's how you can rewrite the code using `with`.

Listing 17. Using with to build the alphabet

The with structure looks like a special construct, but it's a function that takes two arguments: `stringBuilder`, in this case, and a lambda. The convention of putting the lambda outside of the parentheses works here, and the entire invocation looks like a built-in feature of the language. Alternatively, you could write this as `with(stringBuilder, { ... })`, but it's less readable.

The `with` function converts its first argument into a *receiver* of the lambda that's passed as a second argument. You can access this receiver via an explicit `this` reference. Alternatively, as usual for a `this` reference, you can omit it and access methods or properties of this value without any additional qualifiers.

In listing 17, `this` refers to `stringBuilder`, which is passed to `with` as the first argument. You can access methods on stringBuilder via explicit `this` references, as in `this.append(letter);` or directly, as in `append("\nNow...")`.

Lambdas with receiver and extension functions

You may recall that you saw a similar concept with this referring to the function receiver. In the body of an extension function, this refers to the instance of the type the function is extending, and it can be omitted to give you direct access to the receiver's members.

> Note that an extension function is, in a sense, a function with a receiver. The following analogy can be applied:
> 1. Regular function 2. Regular lambda
> 3. Extension function 4. Lambda with a receiver
> A lambda is a way to define behavior similar to a regular function. A lambda with a receiver is a way to define behavior similar to an extension function.

Let's refactor the initial alphabet function even further and get rid of the extra `stringBuilder` variable.

Listing 18. Using with and an expression body to build the alphabet

```
fun alphabet() = with(StringBuilder()) {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
    toString()
}
```

This function now only returns an expression, so it's rewritten using the expression-body syntax. You create a new instance of `StringBuilder` and pass it directly as an argument, and then you reference it without the explicit `this` in the lambda.

> Method-name conflicts
> What happens if the object you pass as a parameter to `with` has a method with the same name as the class in which you're using `with`? In this case, you can add an explicit label to the `this` reference to specify which method you need to call.

Imagine that the alphabet function is a method of the class `OuterClass`. If you need to refer to the `toString` method defined in the outer class instead of the one in StringBuilder, you can do so using the following syntax:

```
this@OuterClass.toStringdocument.getElementById('cloakf705a69ea754601f12b74cf75753116d').inn
```

The value that `with` returns is the result of executing the lambda code. The result is the last expression in the lambda. But sometimes you want the call to return the receiver object, not the result of executing the lambda. That's where the `apply` library function can be of use.

5.2. The "apply" function

The apply function works almost exactly the same as with; the only difference is that apply always returns the object passed to it as an argument (in other words, the receiver object). Let's refactor the alphabet function again, this time using `apply`.

Listing 19. Using apply to build the alphabet

```
fun alphabet() = StringBuilder().apply {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
}.toString()
```

The apply function is declared as an extension function. Its receiver becomes the receiver of the lambda passed as an argument. The result of executing apply is `StringBuilder`, so you call `toString` to convert it to `String` afterward.

One of many cases where this is useful is when you're creating an instance of an object and need to initialize some properties right away. In Java, this is usually accomplished through a separate `Builder` object; and in Kotlin, you can use apply on any object without any special support from the library where the object is defined.

To see how apply is used for such cases, let's look at an example that creates an Android TextView component with some custom attributes.

Listing 20. Using apply to initialize a TextView

```
fun createViewWithCustomAttributes(context: Context) =
    TextView(context).apply {
        text = "Sample Text"
        textSize = 20.0
        setPadding(10, 0, 0, 0)
    }
```

The apply function allows you to use the compact expression body style for the function. You create a new `TextView` instance and immediately pass it to apply. In the lambda passed to apply, the `TextView` instance becomes the receiver, so you can call methods and set properties on it. After the lambda is executed, apply returns that instance, which is already initialized; it becomes the result of the `createViewWithCustomAttributes` function.

The `with` and `apply` functions are basic generic examples of using lambdas with receivers. More specific functions can also use the same pattern. For example, you can simplify the `alphabet` function even further by using the `buildString` standard library function, which will take care of creating a `StringBuilder` and calling `toString`. The argument of `buildString` is a lambda with a receiver, and the receiver is always a `StringBuilder`.

Listing 21. Using buildString to build the alphabet

```
fun alphabet() = buildString {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
}
```

The `buildString` function is an elegant solution for the task of creating a `String` with the help of `StringBuilder`.

Lambdas with receivers are great tools for building DSLs; we'll show you how to use them for that purpose and how to define your own functions that call lambdas with receivers.

6. Summary

- Lambdas allow you to pass chunks of code as arguments to functions.
- Kotlin lets you pass lambdas to functions outside of parentheses and refer to a single lambda parameter as `it`.
- Code in a lambda can access and modify variables in the function containing the call to the lambda.
- You can create references to methods, constructors, and properties by prefixing the name of the function with `::`, and pass such references to functions instead of lambdas.
- Most common operations with collections can be performed without manually iterating over elements, using functions such as `filter`, `map`, `all`, `any`, and so on.
- Sequences allow you to combine multiple operations on a collection without creating collections to hold intermediate results.
- You can pass lambdas as arguments to methods that take a Java functional interface (an interface with a single abstract method, also known as a SAM interface) as a parameter.
- Lambdas with receivers are lambdas in which you can directly call methods on a special receiver object.
- The `with` standard library function allows you to call multiple methods on the same object without repeating the reference to the object. apply lets you construct and initialize any object using a builder-style API.

(/en/coding/evolution-of-java-deployment-platforms)

(/en/coding/good-programs-are-effective,-robust,-efficient-and-maintainable)

Java (/en/component/tags/tag/java)
Kotlin (/en/component/tags/tag/kotlin)
Lambda expressions (/en/component/tags/tag/lambda-expressions)

Comments (0)

There are no comments posted here yet

Posting as Guest | Login

Name (Required)

Email

Write your comment here...

I agree to the terms and condition

Submit Comment

Popular topics

Find...

Most read blogs

J2EE: poweful web application development (/en/coding/j2ee-basics)
Introduction to Oracle E-Business Suite (/en/is/introduction-to-oracle-e-business-suite)
RMAN: Checking for Corruption in Data Files and Backups (/en/databases/oracle/rman-checking-for-corruption-in-data-files-and-backups)
Oracle: Granting the Ability to Create & Execute PL/SQL Stored Programs (/en/databases/sql/oracle-granting-the-ability-to-create-execute-pl-sql-stored-programs)
Restore Oracle Database 12c from an RMAN Backup (/en/databases/oracle/restore-oracle-database-12c-from-an-rman-backup)
Troubleshooting and Tuning LOB Segment Performance in Oracle (/en/databases/oracle/troubleshooting-and-tuning-lob-segment-performance-in-oracle)
PL/SQL procedure: Setting Default Parameter Values (/en/databases/sql/pl-sql-procedure-setting-default-parameter-values)

ВАШ АККАУНТ

Username

Password

Keep me logged in

Log in (/en/component/users/?view=login&Itemid=1284)

Register (/en/component/easysocial/registration)

EN...

(/)

Login

Register for an account (/en/social/registration)

I forgot my username (/en/social/account/lostusername)

I forgot my password (/en/social/account/lostpassword)

Contact Us (https://oracle-patches.com/en/contact-us)

About the Project (/en/about)

Site Map (https://oracle-patches.com/en/site-map)

Site Rules (/en/terms)

Privacy Policy (/en/privacy)

License Agreement (/en/license)

(https://www.facebook.com/groups/databases.oracle)

(https://twitter.com/oracle_patches)

(https://vk.com/oracle.patches)