**Lambda expressions**

A lambda expression in Kotlin is always surrounded by curly braces. Note that there are no parentheses around the arguments.

**val** sum = { x: Int, y: Int -> x + y }

println(sum(1, 2))

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

**val** people = listOf(Person("Alice", 29), Person("Bob", 31))

**var** oldest = people.maxBy({ p: Person -> p.age })

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:

oldest = 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:

oldest = people.maxBy { p: Person -> p.age }

Let’s move on with simplifying the shortest syntax and get rid of the parameter type:

oldest = people.maxBy { p -> p.age }

The last simplification you can make in this example is to replace an argument with the default argument name: it

oldest = people.maxBy { it.age }

**Accessing variables in scope**

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.in Kotlin, you aren’t restricted to accessing nonfinal variables. You can also modify variables from within a lambda.

**Member references**

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.

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

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

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

**fun** salute() = println("Salute!")

run(::salute)

This lambda delegates to a sendEmail function.

**val** action = { person: Person, message: String ->

sendEmail(person, message)

}

You can use a member reference instead.

**val** nextAction = ::sendEmail

You can store or postpone the action of creating an instance of a class using a constructor reference.

**val** createPerson = ::Person

**val** p = createPerson("Alice", 29)

you can also reference extension functions the same way:

**fun** Person.isAdult() = age >= 21

**val** predicate = Person::isAdult

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.

**val** p = Person("Dmitry", 34)

**val** personsAgeFunction = Person::age

println(personsAgeFunction(p))

**val** dmitrysAgeFunction = p::age

println(dmitrysAgeFunction())

**Functional APIs for collections**

**Filter and map**

**val** list = listOf(1, 2, 3, 4)

**val** newList = list.filter { it % 2 == 0 }

**val** people = listOf(Person("Alice", 29), Person("Bob", 31))

**val** newList = people.filter { it.age > 30 }

**val** list = listOf(1, 2, 3, 4)

**val** newList = list.map { it \* it }

**val** people = listOf(Person("Alice", 29), Person("Bob", 31))

**val** newList = people.map { it.name }

**val** name = people.filter { it.age > 30 }.map(Person::name)

**val** oldest = people.filter { it.age == people.maxBy(Person::age)?.age }

**All, any, count, and find**

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

**val** people = listOf(Person("Alice", 27), Person("Bob", 31))

println(people.all(canBeInClub27))

println(people.any(canBeInClub27))

println(people.count(canBeInClub27))

println(people.find(canBeInClub27))

println(list.all { it == 3 })

**GroupBy**

println(people.groupBy { it.age })

Each group is stored in a list, so the result type is Map<Int, List<Person>> .

**Lazy collection operations: sequences**

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.

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

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

people.asSequence()

.map(Person::name)

.filter { it.startsWith("A") }

.toList()

**Creating sequences**

The previous examples used the same method to create a sequence: you called asSequence() on a collection. Another possibility is to use the generateSequence() function.

**val** naturalNumbers = generateSequence(0) { it + 1 }

**val** numbersTo100 = naturalNumbers.takeWhile { it <= 100 }

println(numbersTo100.sum())

All the delayed operations are performed when the result "sum" is obtained.

**The with function**

**fun** alphabet(): String {

**val** stringBuilder = StringBuilder()

**return** with(stringBuilder)

//Specifies the receiver value on which you’re calling the methods

{

**for** (letter **in** 'A'..'Z') {

**this**.append(letter)

//Calls a method on the receiver value though an explicit "this"

}

append("\nNow I know the alphabet!")

//Calls a method, omitting "this"

**this**.toString()

//Returns a value from the lambda

}

}

**The apply function**

**fun** alphabet() = StringBuilder().apply {

**for** (letter **in** 'A'..'Z') {

append(letter)

}

append("\nNow I know the alphabet!")

}.toString()