Kotlin Functional Programming

Download the full source codes from here
https://drive.google.com/file/d/1JyVLjUWXQbmXIkUifjKlfzh-iipPNhv4/view?usp=sharing

Kotlin is not a pure functional language but do support Lambdas and High order functions.

Lambdas - anonymous functions

Lambda expressions and anonymous functions are *function literals*. **Function literals are functions that are not declared but are passed immediately as an expression**

```
val addTwo: (Int) -> Int = { x -> x + 2 }
println(addTwo(2))
```

We created an anonymous function and save her in a val later we will send it to another function. The function that receives it is will be called an **High Order function.**

Our Lambdas can be shorter using Type inferred - Kotlin deduced it from the function - but remember pυrameters types can never be inferred.

```
val sub = { x: Int, y: Int -> x - y }
println(sub(5, 3))
```

We can use Lambdas in High order functions - functions that receives other functions as arguments
filter() - a pre-existing high order function that filtering a list using a given predicate. A Predicate is a lambda function that take a collection element and return a boolean value: true means that the given element matches the predicate, false means the opposite

```
val list = (1..100).toList()

println(list)//the whole list

println(list.filter { x -> x % 3 == 0 })//only the numbers divided by 3

println(list.filter { it % 4 == 0})//using implicit it
//default lambda parameter
```

1. Because the function receives Lambdas we can discard the () also if the lambda is the last parameter it can be written outside the parentheses
2. Implicit **it -** if the Lambda has only one parameter it can be implicitly called by **it**

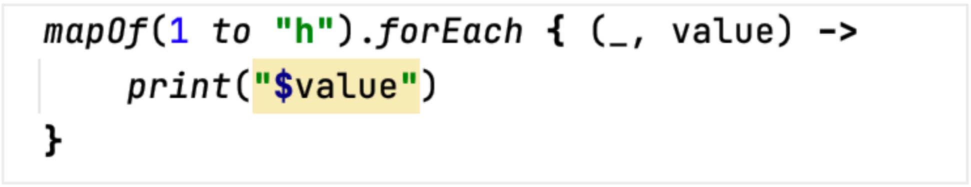
According to Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses - Such syntax is also known as ***trailing lambda*.**

When using default parameters, If the last argument after default parameters is a lambda, you can pass it either as a named argument or outside the parentheses. If the lambda is the only mandatory argument in that call, the parentheses can be omitted entirely:

```
fun foo(
    bar: Int = 0,
    baz: Int = 1,
    qux: () -> Unit,
) { /*...*/ }

foo(1) { println("hello") }     // Uses the default v
foo(qux = { println("hello") }) // Uses both default
foo { println("hello") }        // Uses both default
```

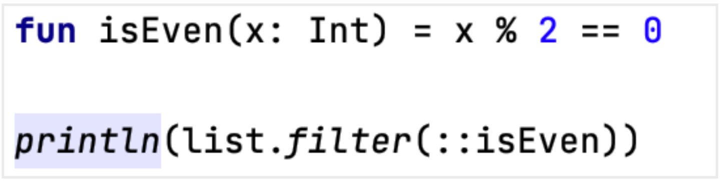
You can explicitly return a value from the lambda using the return syntax. Otherwise, the value of the last expression is implicitly returned.

If the lambda parameter is unused, you can place an underscore instead of its name:



```
mapOf(1 to "h").forEach { (_, value) ->
    print("$value")
}
```

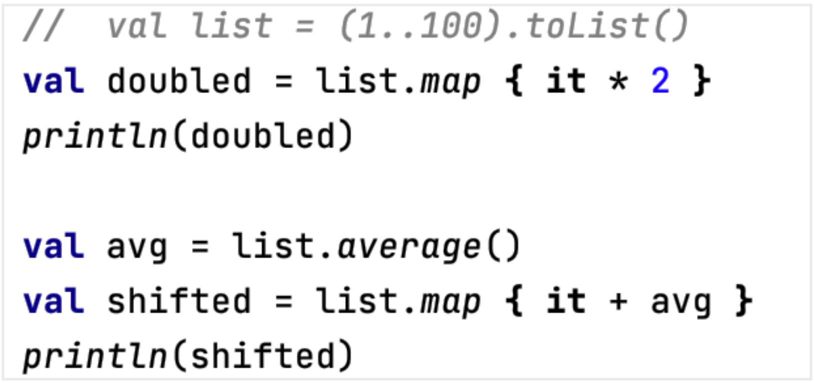
Function pointer ::
If we want to pass non-anonymous function we can use the **::** which is a function pointer



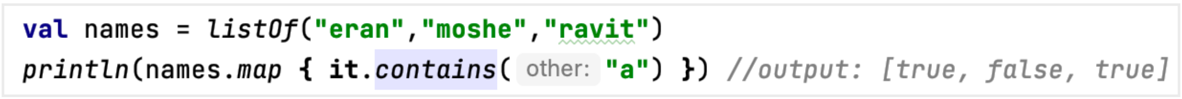
```
fun isEven(x: Int) = x % 2 == 0

println(list.filter(::isEven))
```

The filter function will call the isEven function passing it each element on the list and will filter by it's result

map
Returns a list containing the results of applying the given transform function to each element in the original array



```
//  val list = (1..100).toList()
val doubled = list.map { it * 2 }
println(doubled)

val avg = list.average()
val shifted = list.map { it + avg }
println(shifted)
```

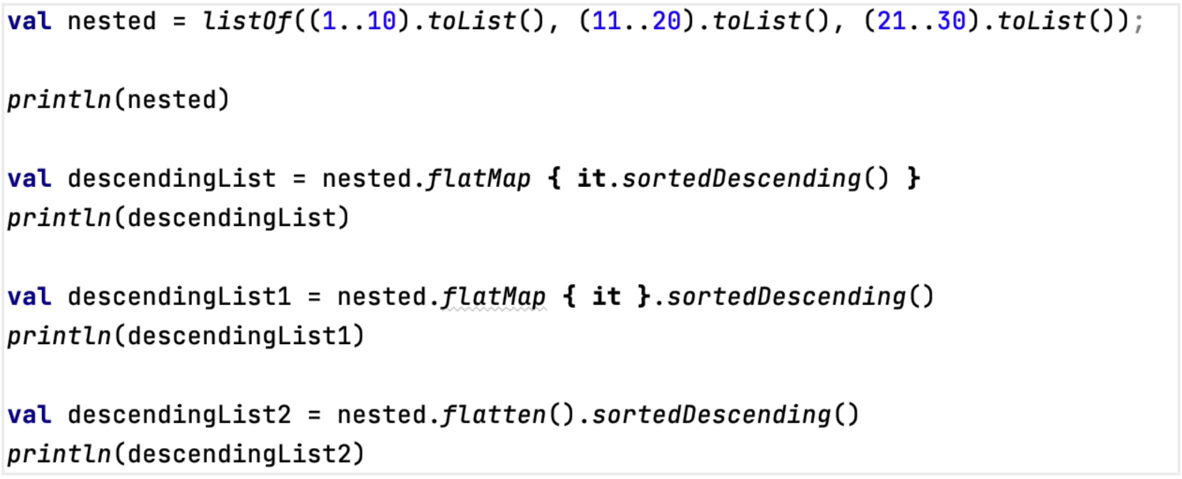


```
val names = listOf("eran","moshe","ravit")
println(names.map { it.contains( other: "a") }) //output: [true, false, true]
```

In the last example we get a list of Boolean since it is the result of applying the contains function.

flatMap
Returns a single list of all elements yielded from results of transform function being invoked on each element of original array.

In other words both function applies a function to each of the elements but flat map also "FLATTEN" it - To understand what *flattening* a stream consists in, consider a structure like [[1,2,3],[4,5,6],[7,8,9]] which has "two levels". Flattening this means transforming it in a "one level" structure :
[1,2,3,4,5,6,7,8,9]

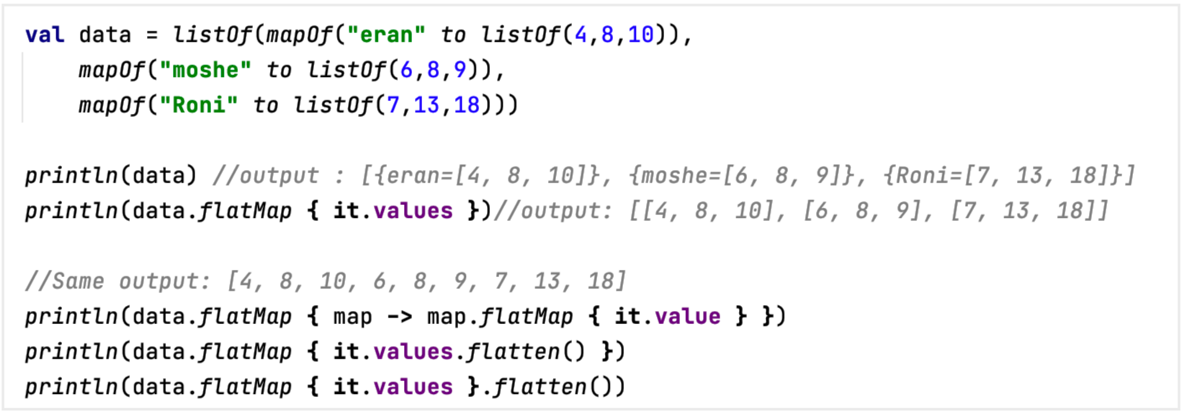


```kotlin
val nested = listOf((1..10).toList(), (11..20).toList(), (21..30).toList());

println(nested)

val descendingList = nested.flatMap { it.sortedDescending() }
println(descendingList)

val descendingList1 = nested.flatMap { it }.sortedDescending()
println(descendingList1)

val descendingList2 = nested.flatten().sortedDescending()
println(descendingList2)
```

Please note that in the above example the **it** refers to the list inside - the it the inner object

More on collection sorting here (please read and note the sortBy function that also receives a selector)
https://www.baeldung.com/kotlin/sort

Let's use complex data like JSON objects.
Lets make a data structure of parents and their children and play a little

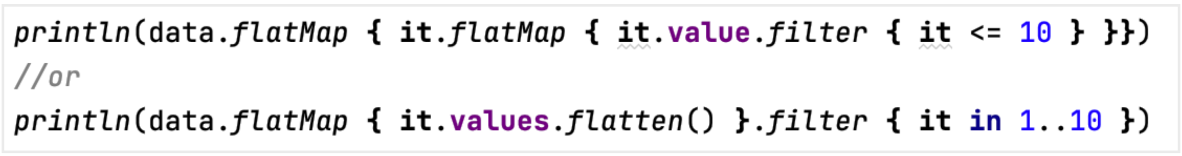


```kotlin
val data = listOf(mapOf("eran" to listOf(4,8,10)),
    mapOf("moshe" to listOf(6,8,9)),
    mapOf("Roni" to listOf(7,13,18)))

println(data) //output : [{eran=[4, 8, 10]}, {moshe=[6, 8, 9]}, {Roni=[7, 13, 18]}]
println(data.flatMap { it.values })//output: [[4, 8, 10], [6, 8, 9], [7, 13, 18]]

//Same output: [4, 8, 10, 6, 8, 9, 7, 13, 18]
println(data.flatMap { map -> map.flatMap { it.value } })
println(data.flatMap { it.values.flatten() })
println(data.flatMap { it.values }.flatten())
```

Drills

1. Using the data above Print one list that contain only the ages of 1 to 10

Solution:



```kotlin
println(data.flatMap { it.flatMap { it.value.filter { it <= 10 } }})
//or
println(data.flatMap { it.values.flatten() }.filter { it in 1..10 })
```

2. Define a map which contain each parent and his children ages (not list of maps, just one map) and put some faulty ages between the correct ones (like -6 or 150). Then print out the average children's age of each parent (take only the valid ages to the already exist Kotlin average()

function) - use map this time. Try to solve this in two ways, use both lambdas and the :: (function pointer)

Solution:

```kotlin
val faultyData = mapOf(
        "eran" to listOf(2,6,8),
        "moshe" to listOf(8,150,19,-1,16),
        "ronit" to listOf(5,-6,10,15)
)

//val isValidAge:(Int)->Boolean = { it in 0..120 }
println(faultyData.map { it.value.filter { it in 0..120 }.average() })

//or
fun isValidAge(x:Int) = x in 0..120
val avg = faultyData.map { it.value.filter(::isValidAge).average() }
println(avg)
```

3. Now print the average of all the children together

Solution:

```kotlin
println(faultyData.flatMap { it.value.filter (::isValidAge) }.average())
//or
val avg1 = faultyData.flatMap { it.value }.filter { it in 0..120 }.average()
println(avg1)
```

4. Print the names of parents with faulty ages. Hint: search the Kotlin API for function that applies boolean condition on a list

Solution:
any() - Returns true if at least one element matches the given predicate (boolean values functions)

```kotlin
 println(faultyData.filter { it.value.any {it !in 0..120 } }.keys)
```

5. Print the number of faulty ages

Solution:

```kotlin
 val numberOfFaultyAges = faultyData.flatMap { it.value }.filter { it !in 0..120 }.size
```

6. Print only the names of the parents who has no faulty data (use all())

Solution:

```
val namesWithValidAges = faultyData.filter { it.value.all { it in 0..120 } }.keys
println(namesWithValidAges)
```

take(n) - takes only the first n elements
drop(n) - leaves the first n elements and leaves the rest
first(), last() - the first and last elements of the list

takeIf()
take the list if the the given predicate gives true
Take the list if it contains 4 if not we will get null

```
println(list.takeIf { it.contains(4) })
```

Note: takeIf() and takeUnless() is part of the Kotlin Standard library and will be mentioned later on. It can be applied to any instance of any class.

zip() - create list of Pairs. each pair will consist of key from the first list and value from the second. If one list in longer then the second it's extra values will be ignored

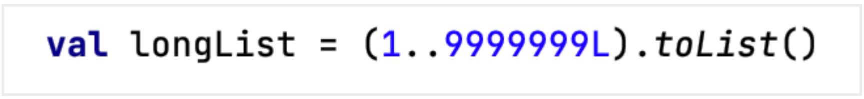
```
val names = listOf("eran", "moshe", "dave", "yael")
val numOfChildren = listOf(3, 2, 5, 3);

val zipped = names.zip(numOfChildren);
println(zipped)
```

Drill
Use zip function to create a list of pairs that each pair's key is the name and the value is true or false whether the name contains the character 'a'

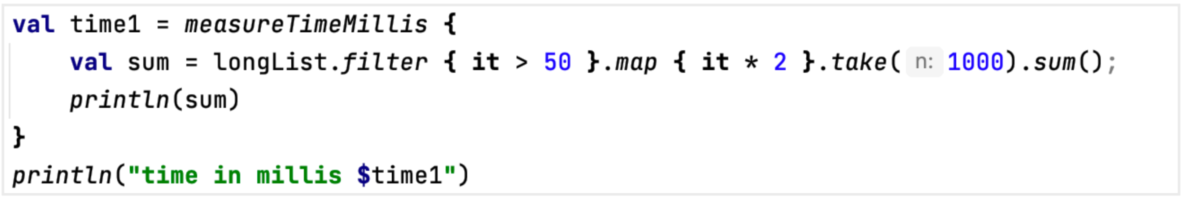
```
println(names.zip(names.map { it.contains( other: "a") }))
```

For more reading on צק and High Order functions:
https://kotlinlang.org/docs/lambdas.html#higher-order-functions

Lazy Sequence

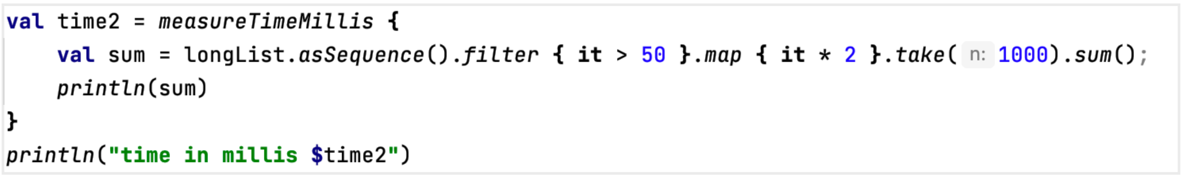


```kotlin
val longList = (1..9999999L).toList()
```



```kotlin
val time1 = measureTimeMillis {
    val sum = longList.filter { it > 50 }.map { it * 2 }.take(n: 1000).sum();
    println(sum)
}
println("time in millis $time1")
```

This is about 1-2 seconds

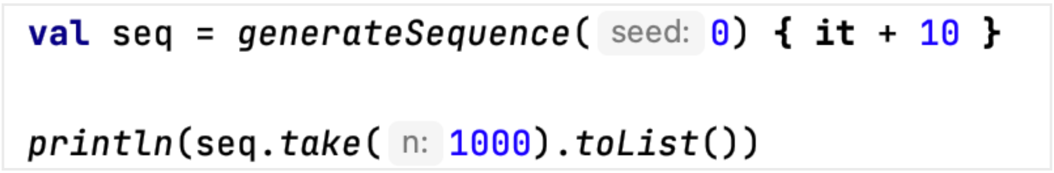


```kotlin
val time2 = measureTimeMillis {
    val sum = longList.asSequence().filter { it > 50 }.map { it * 2 }.take(n: 1000).sum();
    println(sum)
}
println("time in millis $time2")
```

This takes only 10 mili-seconds

Sequences are containers where the main difference between them and all the other collections is that the **actual computing is done only when needed** - in both examples we only needs the first 1000 not on all of the elements. In the first example the computations were made on all the list, but in the second one the computations were made on only the ones we needed - the first 1000 elements - this is the meaning of Lazy Sequences. And in general Lazy is a wide concept in Kotlin meaning that it's only done when needed.

Because of this Sequences can be infinite.



```kotlin
val seq = generateSequence(seed: 0) { it + 10 }

println(seq.take(n: 1000).toList())
```

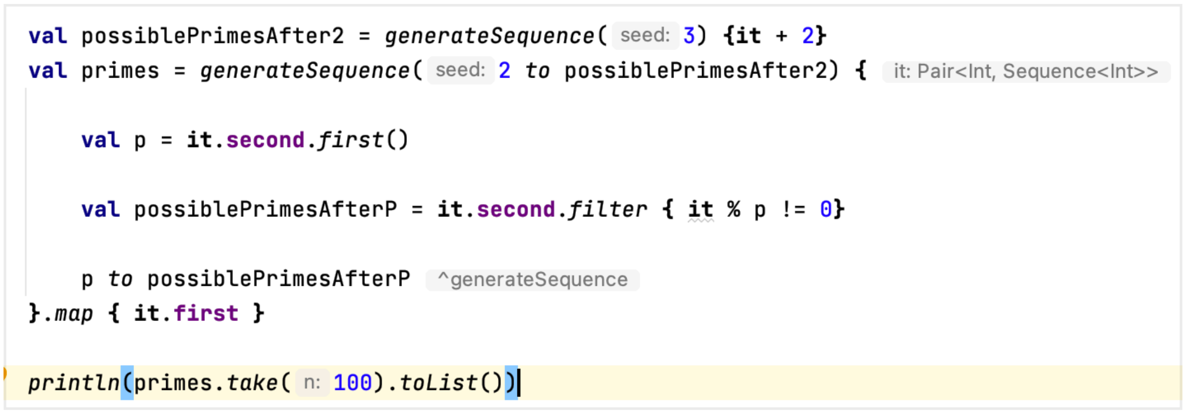
 If we don't take the first 1000 the program will go to infinite loop - he tries to turn infinite loop into list

Lazy Algorithm for finding prime numbers - Sieve of Eratosthenes

This algorithm is based on taking each number and removes all his multiplications

Drill - Advance
Build an algorithm based on the sieve of Eratosthenes that print the first n primes - use sequences

Solution:



```
val possiblePrimesAfter2 = generateSequence( seed: 3) {it + 2}
val primes = generateSequence( seed: 2 to possiblePrimesAfter2) {  it: Pair<Int, Sequence<Int>>

    val p = it.second.first()

    val possiblePrimesAfterP = it.second.filter { it % p != 0}

    p to possiblePrimesAfterP  ^generateSequence
}.map { it.first }

println(primes.take( n: 100).toList())
```

Scope functions

The Kotlin standard library contains several functions whose sole purpose is to execute a block of code within the context of an object. When you call such a function on an object with a lambda expression provided, it forms a temporary scope. In this scope, you can access the object without its name. Such functions are called *scope functions*. There are five of them: **let, run, with, apply, and also**.

Basically, these functions do the same: execute a block of code on an object. What's different is how this object becomes available inside the block and what is the result of the whole expression.
Here is a short guide for choosing scope functions depending on the intended purpose:

- Executing a lambda on non-null objects: **let**
- Introducing an expression as a variable in local scope: **let**
- Object configuration: **apply**
- Object configuration and computing the result: **run**
- Additional effects: also
- Grouping function calls on an object: with

here are two main differences between each scope function:

- The way to refer to the context object
- The return value.

The scope functions differ by the result they return:

- apply and also return the context object.
- **let, run, and with return the lambda result.**

The return value of **apply** and **also** is the context object itself. Hence, they can be included into call chains as *side steps*: you can continue chaining function calls on the same object after them. They also can be used in return statements of functions returning the context object.

let, run, and with return the lambda result. So, you can use them when assigning the result to a variable, chaining operations on the result, and so on.

let()

define a scope a variable - it applies the the variable that we opened the scope on, we can give it a name

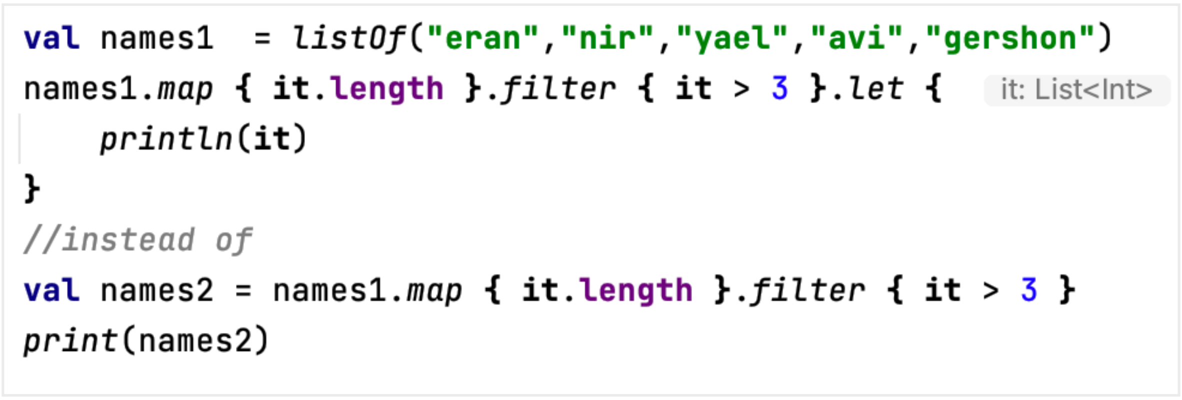
```
File( pathname: "text.txt").bufferedReader().let {reader ->

    if (reader.ready()) {

    }
}

//or

File( pathname: "text.txt").bufferedReader().let {  it: BufferedReader

    if (it.ready()) {

    }
}
```

We can use **run() and instead of the lambda argument (it) we have the lambda receiver (this):**

```
File( pathname: "text.txt").bufferedReader().run {  this: BufferedReader
    if(ready()) {

    }
}
```

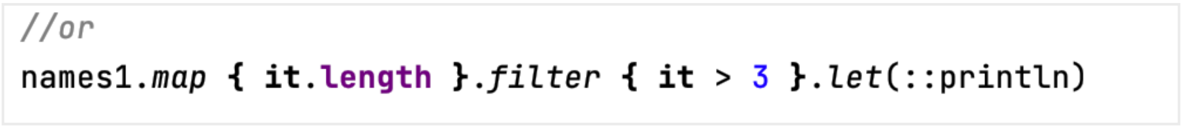
On the other hand, if this is omitted, it can be hard to distinguish between the receiver members and external objects or functions. So, having the context object as a receiver (this) is **recommended for lambdas that mainly operate on the object members: call its functions or assign properties.**

Drill
Create a list of names and print the size of only the names that are longer then 3 using let



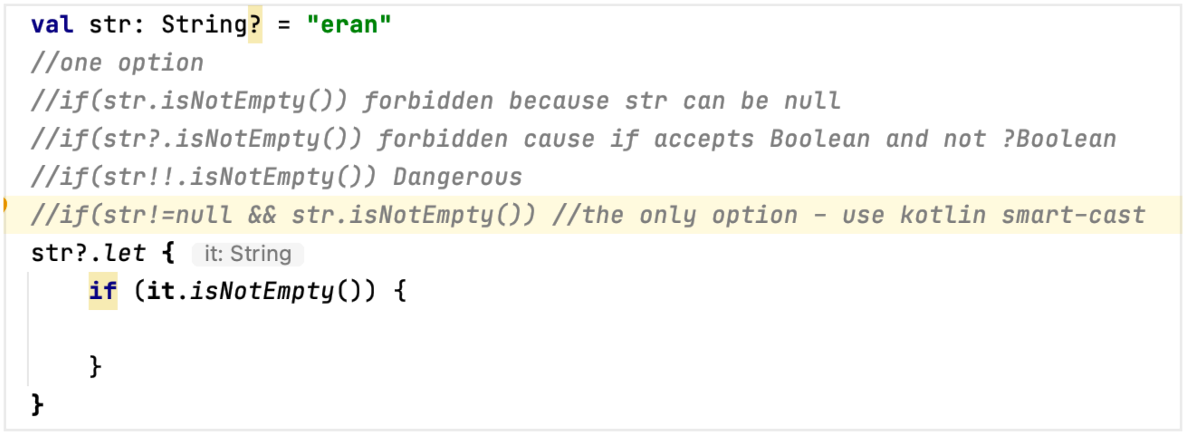
```
val names1  = listOf("eran","nir","yael","avi","gershon")
names1.map { it.length }.filter { it > 3 }.let {   it: List<Int>
    println(it)
}
//instead of
val names2 = names1.map { it.length }.filter { it > 3 }
print(names2)
```

If the code block contains a single function with it as an argument, you can use the method reference (::) instead of the lambda:



```
//or
names1.map { it.length }.filter { it > 3 }.let(::println)
```

We can use let for working with nulls - we enter the block only if it exists maybe aging str is not null



```
val str: String? = "eran"
//one option
//if(str.isNotEmpty()) forbidden because str can be null
//if(str?.isNotEmpty()) forbidden cause if accepts Boolean and not ?Boolean
//if(str!!.isNotEmpty()) Dangerous
//if(str!=null && str.isNotEmpty()) //the only option - use kotlin smart-cast
str?.let {   it: String
    if (it.isNotEmpty()) {

    }
}
```

run()

run is used to execute a block of code and return the result - if the run function invoked on an object - not mandatory - unlike let - you can refer it by this (not **it** like let), like let it returns the lambda result

Combine let and run :

```kotlin
var p : String? = null
p?.let { println("p is $p") } ?: run { println("p was null. Setting default value to: ")
    p = "Kotlin"}

println(p)
//output:
//p was null. Setting default value to:
//Kotlin
```

also()

also is very similar to let but instead of the lambda result it **return the object itself** (both have **it**)

```kotlin
data class Person(var first: String, var last : String)
val person = Person("Moshik", "Afia")

val l = person.let { it.last = "Afik" }
val al = person.also { it.last = "Afik" }

println(l) //output: kotlin.Unit - the lambda return Unit
println(al) //output: Person(first=Moshik, last=Afik)
```

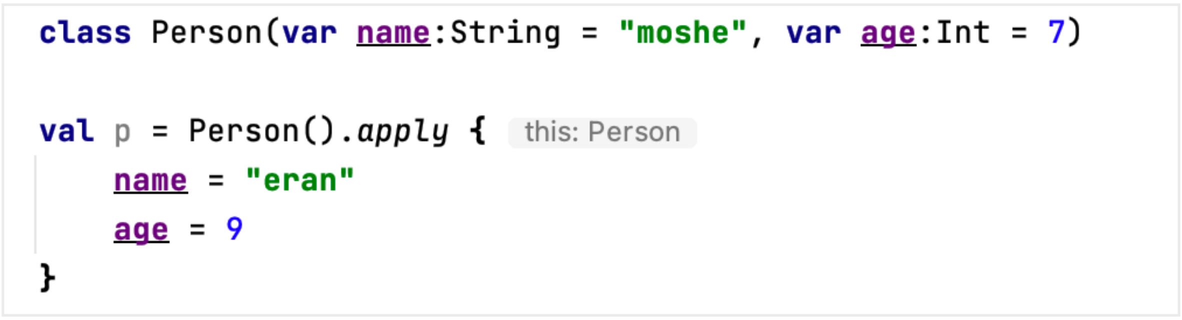
The also expression returns the data class object whereas the let expression returns nothing (Unit) as we didn't specify anything explicitly.

apply

The context object is available as a receiver (this). **The return value** is the object itself.
This is what differentiate it from also - Apply and also are almost the same apply has **this** and also **it**

```kotlin
person.apply { first = "Moshon" }

person.also { it.first = "Mushi" }
```

Use apply for code blocks that don't return a value and mainly operate on the members of the receiver object. The common case for apply is the object configuration. Such calls can be read as " *apply the following assignments to the object."*

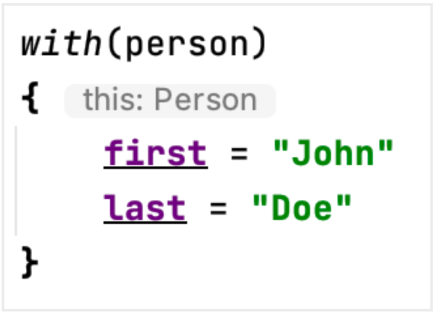


```
class Person(var name:String = "moshe", var age:Int = 7)

val p = Person().apply {  this: Person
    name = "eran"
    age = 9
}
```

On the other hand, We should use also only when we don't want to shadow this.

with()

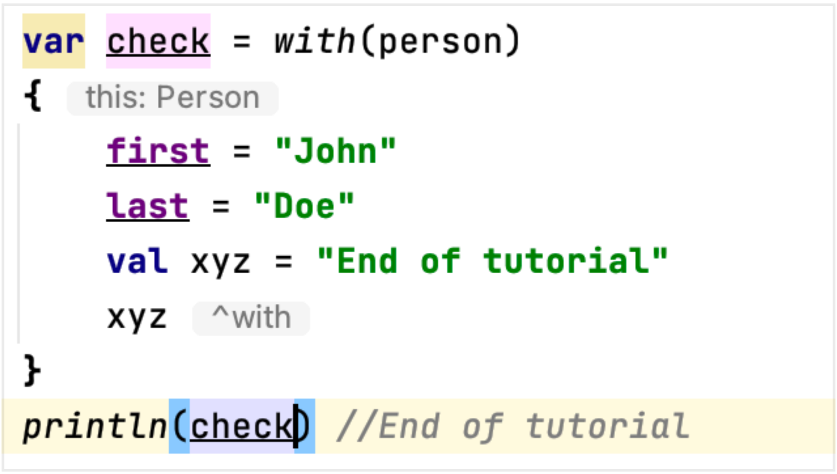
"*with this object, do the following.*" - we have **this**



```
with(person)
{  this: Person
    first = "John"
    last = "Doe"
}
```

Because we have **this** it is very similar to apply but with 3 major difference:

1. Apply must work on an instance (the receiver) in with the instance is supplied as a parameter
2. with returns the lambda's result:



```
var check = with(person)
{  this: Person
    first = "John"
    last = "Doe"
    val xyz = "End of tutorial"
    xyz  ^with
}
println(check) //End of tutorial
```

For more reading on scope functions:
https://kotlinlang.org/docs/scope-functions.html

Please Notre that all of the scoped functioned mentioned above are part of the Kotlin's standard functions
There you can also find a function we already discussed **takeIf() & takeUnless()**

Lets look closely on takeIf:

```
public inline fun <T> T.takeIf(predicate: (T) -> Boolean): T?
    = if (predicate(this)) this else null
```

From it, we notice that

- It is called from the T object itself. i.e. T.takeIf.
- The predicate function takes T object as parameter
- It returns this or null pending on the predicate evaluation.

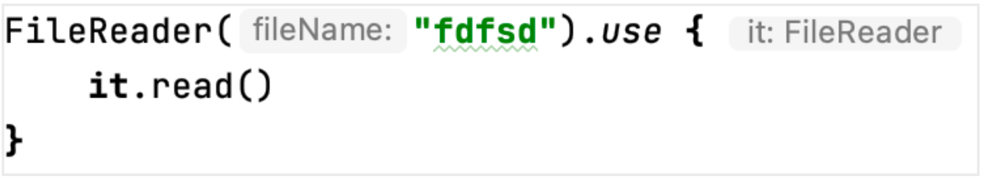
Thus it is very useful in null checks:

```
// Original code
if (someObject != null && status) {
   doThis()
}


// Improved code
someObject?.takeIf{ status }?.apply{ doThis() }
```

You can read more on takeIf() from where is example is taken from

use()

Executes the given block function on this resource and then closes it down correctly whether an exception is thrown or not.
Must be used on objects that implements the java closable interface

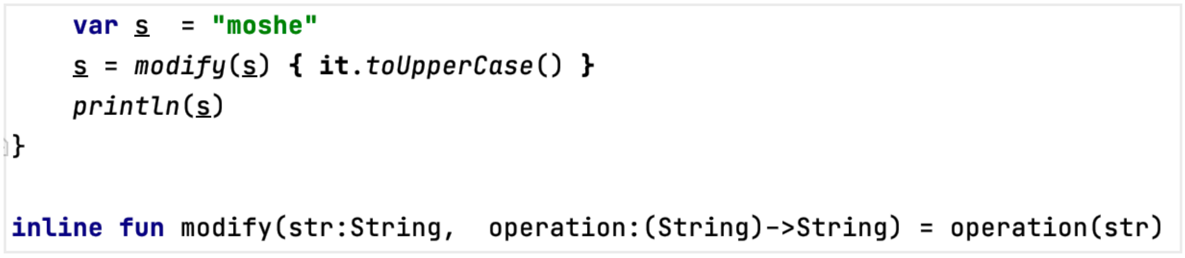


```
FileReader( fileName: "fdfsd").use {  it: FileReader
    it.read()
}
```

No need to close the FileReader

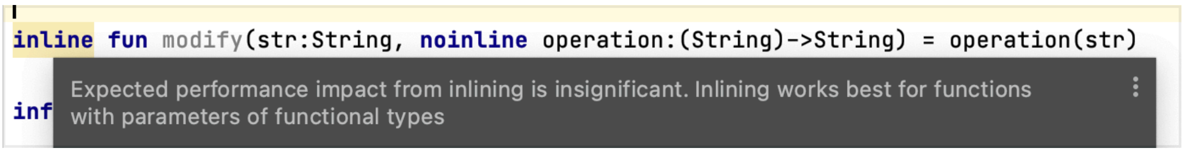
Inline function

All of our scoped functions were **inline**
Meaning the compiler copies the function code to the place where the we invoke it
When using lambda the compiler create an instance of the function each time unless we declare the function be inline and then he just copies the function code



```
    var s  = "moshe"
    s = modify(s) { it.toUpperCase() }
    println(s)
}

inline fun modify(str:String,  operation:(String)->String) = operation(str)
```

If we add the keyword **noinline** before the lambda the compiler will alert us that the inline keyword has no meaning cause it saves time when working with functions



```
inline fun modify(str:String, noinline operation:(String)->String) = operation(str)

inf
```

Expected performance impact from inlining is insignificant. Inlining works best for functions with parameters of functional types

Operator Overloading in Kotlin

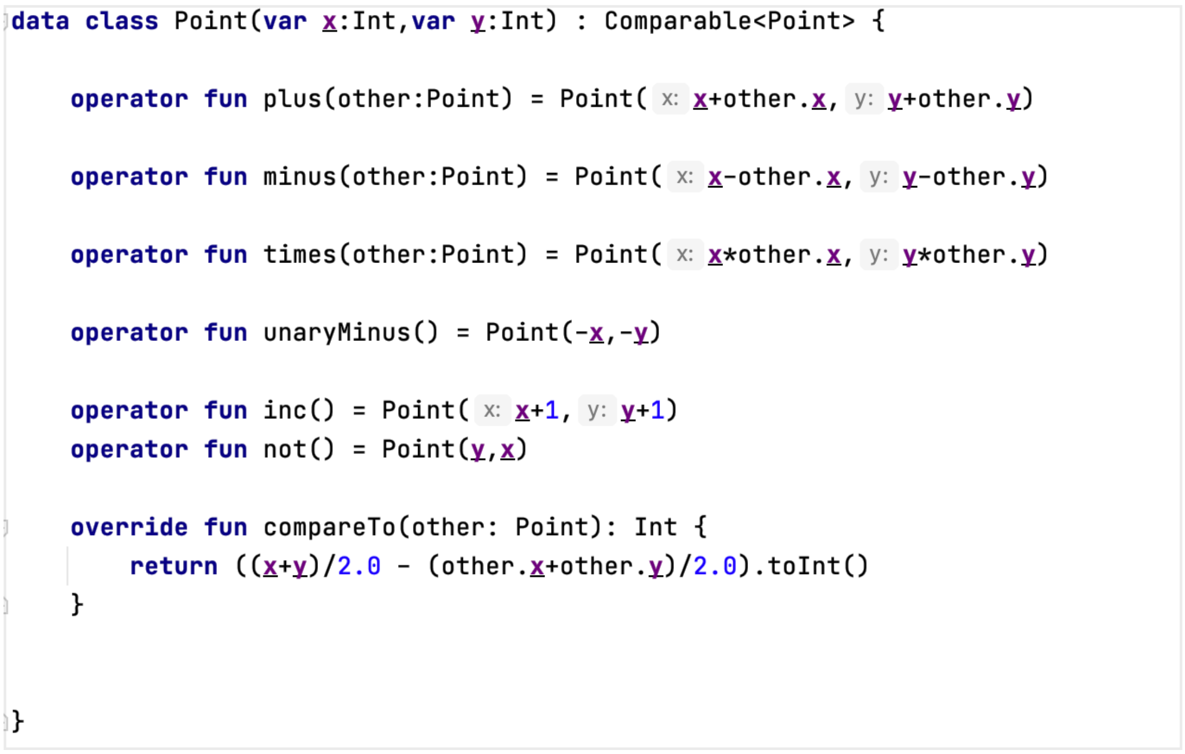
Use **operator** keyword before the fun

We already have by default the == and != when we override the Any's equals() method

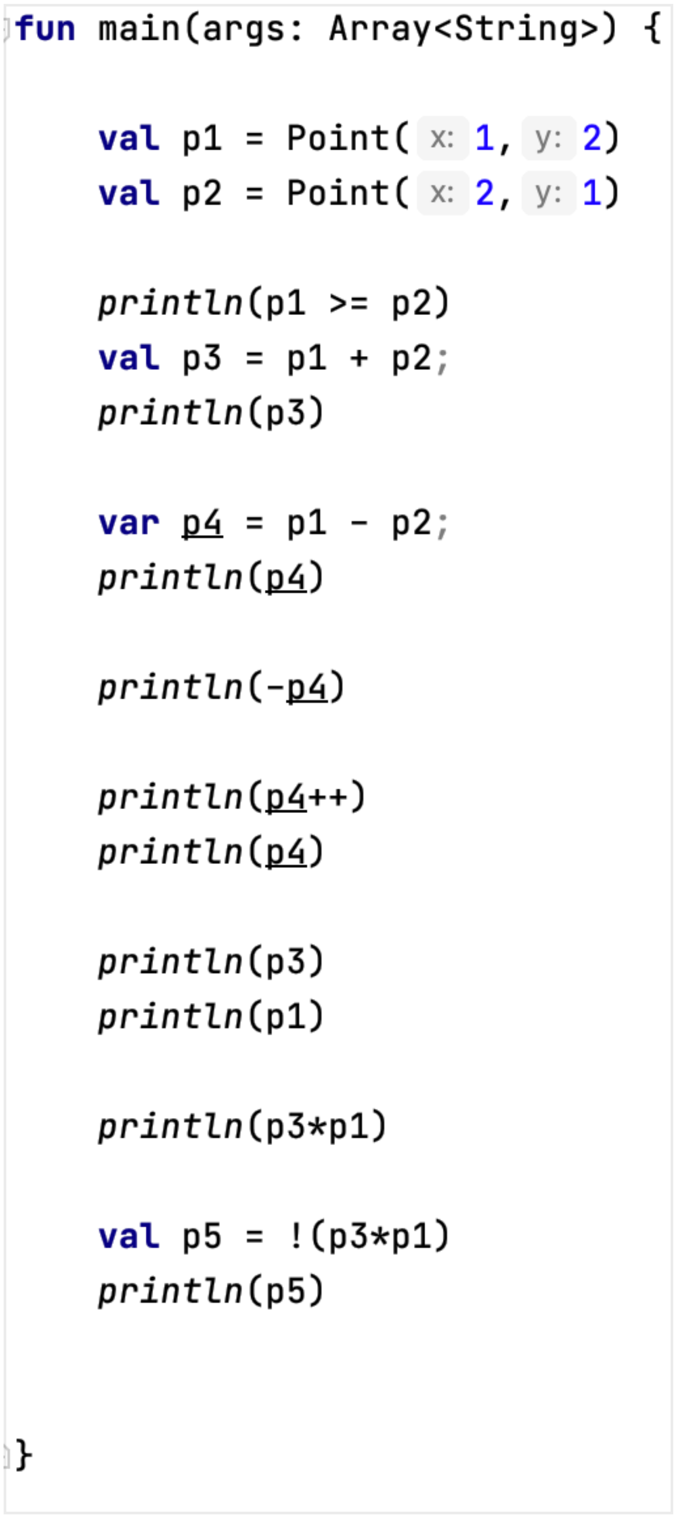
We can overload the plus, minus, times, unsayMinus, unaryPlus, inc(++), dec(—), and not

Kotlin has a special treatment of Java's Comparable.
Simply put, we can call the compareTo method in the Comparable interface by a few Kotlin conventions. In fact, any comparisons made by "<", "<=", ">", or ">=" would be translated to a compareTo function call.

In other words, If we want to overload the > , >= and < , <= we need to implement the Comparable interface



```kotlin
data class Point(var x:Int,var y:Int) : Comparable<Point> {

    operator fun plus(other:Point) = Point( x: x+other.x, y: y+other.y)

    operator fun minus(other:Point) = Point( x: x-other.x, y: y-other.y)

    operator fun times(other:Point) = Point( x: x*other.x, y: y*other.y)

    operator fun unaryMinus() = Point(-x,-y)

    operator fun inc() = Point( x: x+1, y: y+1)
    operator fun not() = Point(y,x)

    override fun compareTo(other: Point): Int {
        return ((x+y)/2.0 - (other.x+other.y)/2.0).toInt()
    }


}
```



```
fun main(args: Array<String>) {

    val p1 = Point( x: 1, y: 2)
    val p2 = Point( x: 2, y: 1)

    println(p1 >= p2)
    val p3 = p1 + p2;
    println(p3)

    var p4 = p1 - p2;
    println(p4)

    println(-p4)

    println(p4++)
    println(p4)

    println(p3)
    println(p1)

    println(p3*p1)

    val p5 = !(p3*p1)
    println(p5)


}
```