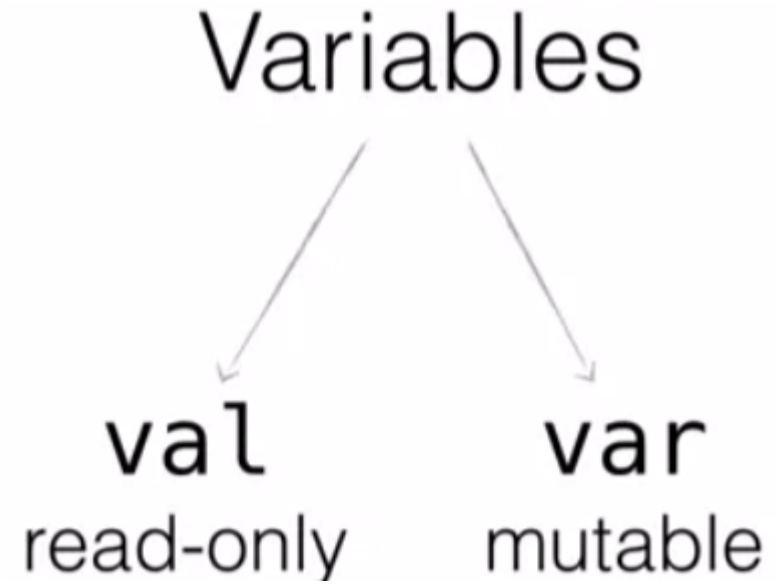
Kotlin for Java Developers

Basics

Variables



Readonly variables: val

```
val question: String =
        "life, the universe, " +
                "and everything"
println("$question?")

question = "sure?"

    Compiler error:
val cannot be reassigned
```

corresponds to a `final` variable in Java

Mutable variables: var

```
var answer = 0
answer = 42
println(answer)          // 42
```

Local type Inference

```
              : String
val greeting = "Hi!"

              : Int
var number = 0
```

String and Int types are inferred

Lists: mutable & read-only

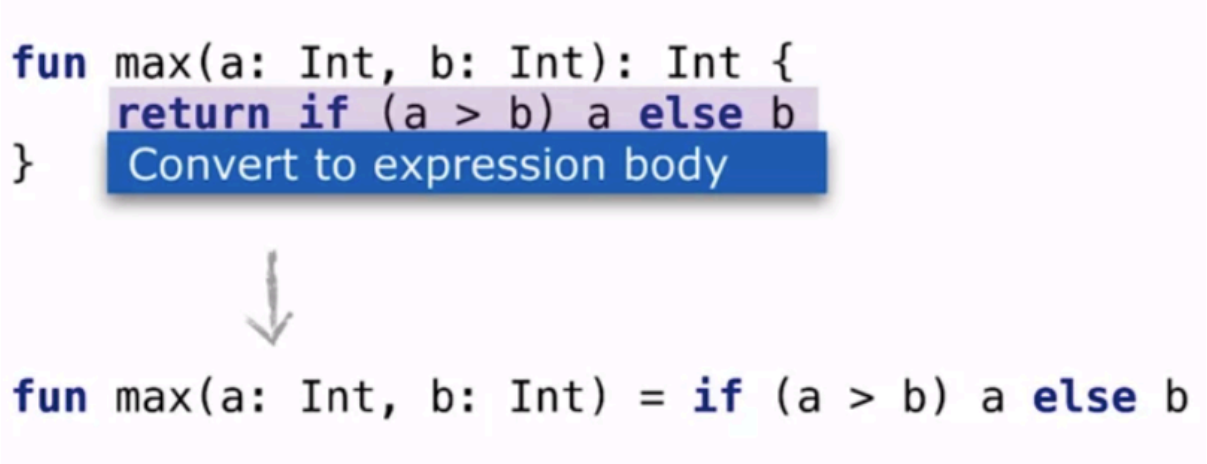
```
val mutableList = mutableListOf("Java")
mutableList.add("Kotlin")
```

```
val readOnlyList = listOf("Java")
readOnlyList.add("Kotlin")
```

Read-only list lacks mutating methods

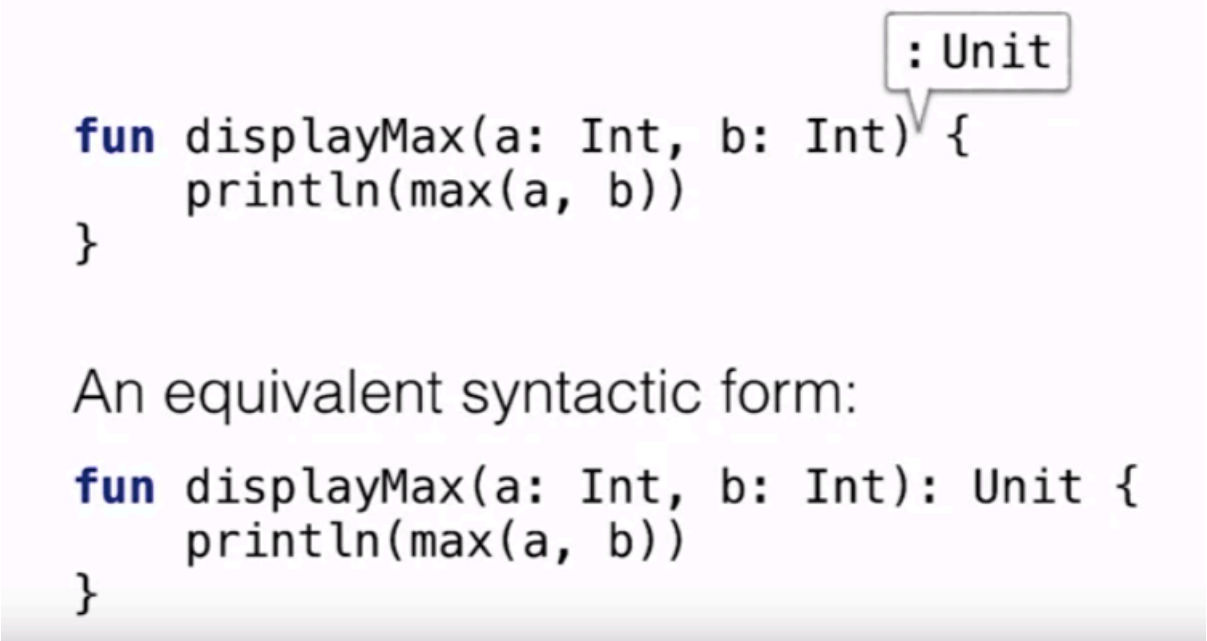
Prefer vals to vars

Functions

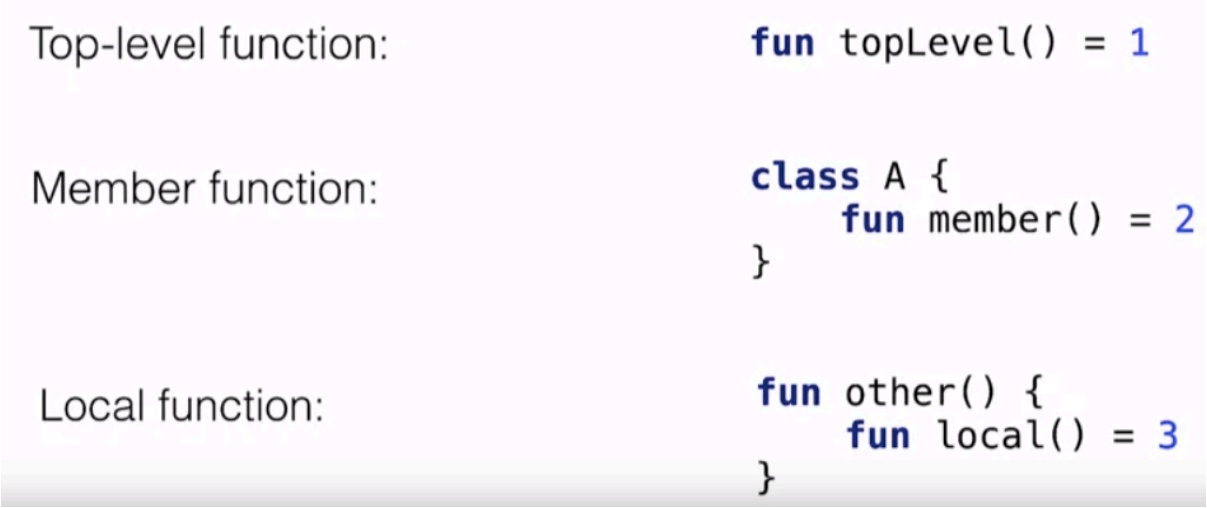


Function returning Unit

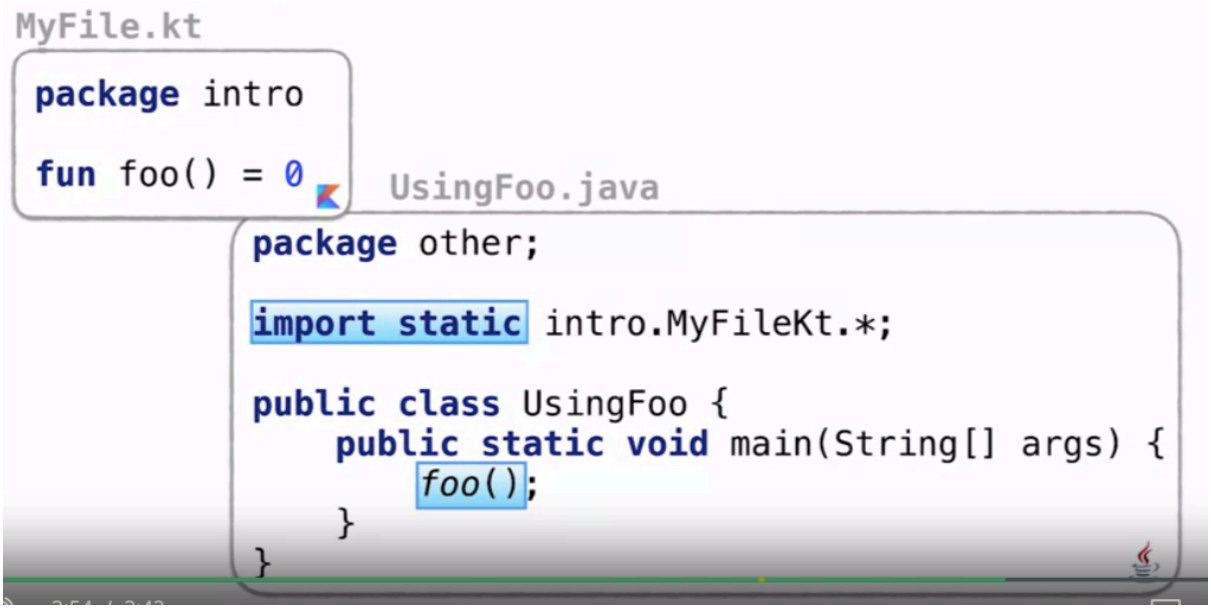
Unit is an analog of void



Top Level - Member - Local

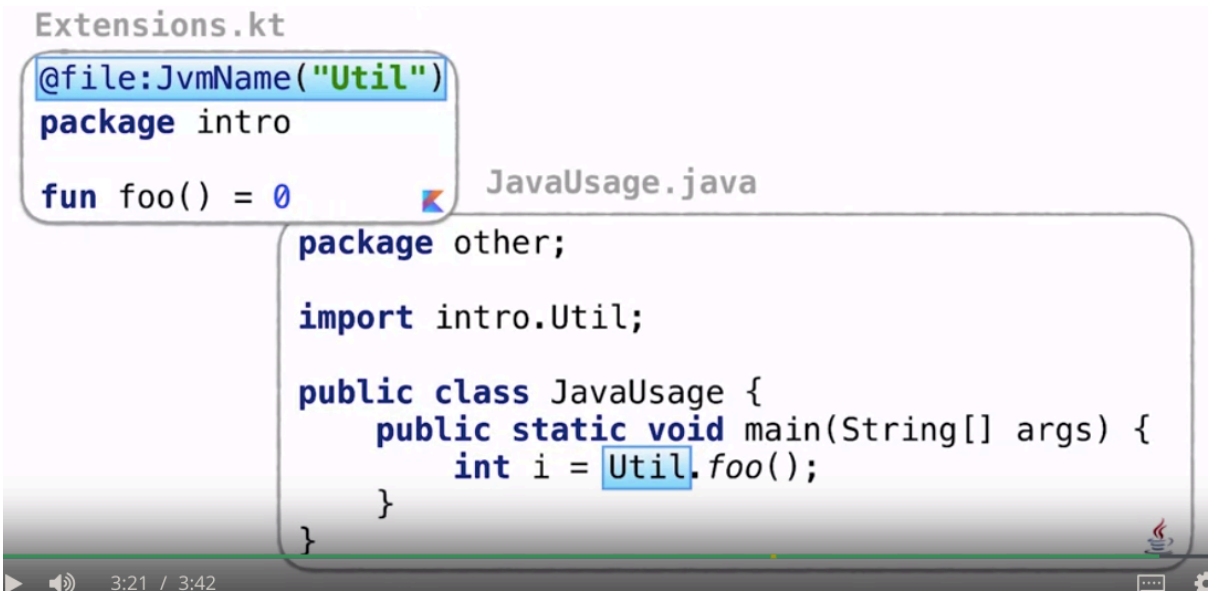


Calling Top-Level function from Java



@JvmName

Changes the name of JVM name of the class containing top-level functions.



Named & default arguments

Functions: default values

```
fun displaySeparator(character: Char = '*', size: Int = 10) {
    repeat(size) {
        print(character)
    }
}


    displaySeparator('#', 5)      // #####
    displaySeparator('#')         // ##########
    displaySeparator()            // **********
```

Control Structures

if & when

if is an expression

```
val max = if (a > b) a else b
```

No ternary operator in Kotlin:

```
(a > b) ? a : b
```

when as switch

```
enum class Color {
    BLUE, ORANGE, RED
}

fun getDescription(color: Color): String =
    when (color) {
        BLUE -> "cold"
        ORANGE -> "mild"
        RED -> "hot"
    }
```

No break is needed

```
switch (color) {
  case BLUE:
    System.out.println("cold");
    break;
  case ORANGE:
    System.out.println("mild");
    break;
  default:
    System.out.println("hot");
}
```

```
when (color) {
    BLUE -> println("cold")
    ORANGE -> println("mild")
    else -> println("hot")
}
```

Checking values

check several values at once

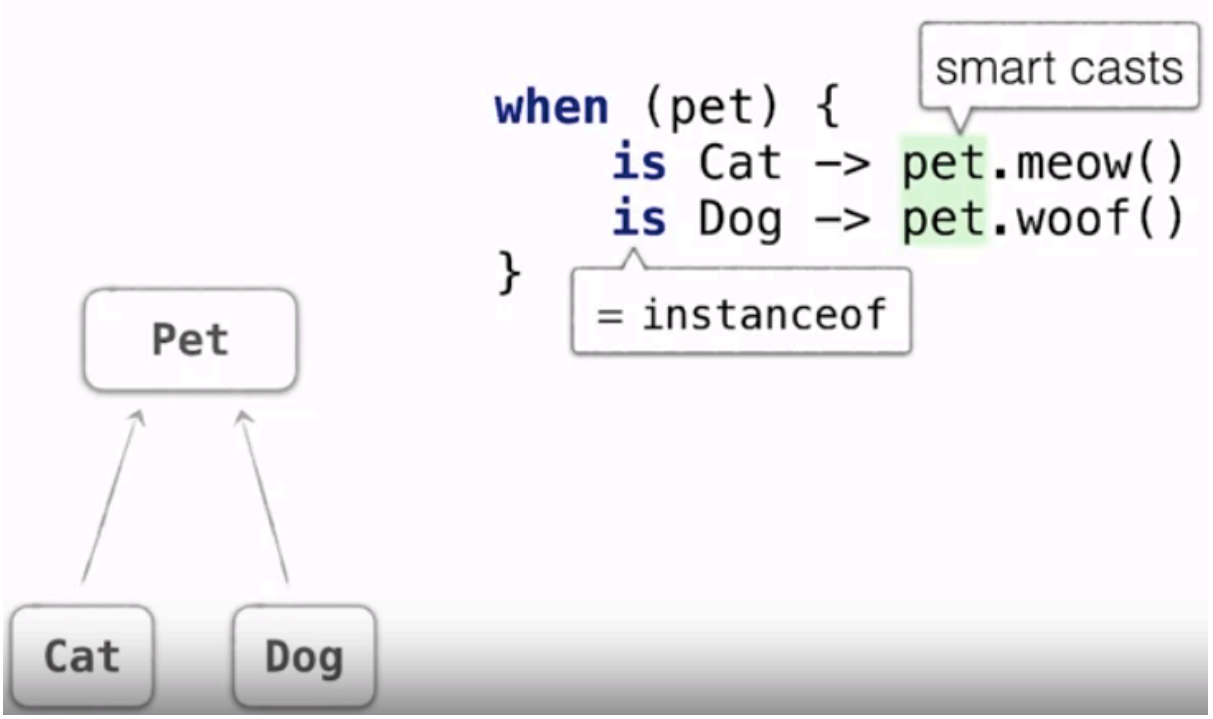
```
fun respondToInput(input: String) = when (input) {
    "y", "yes" -> "I'm glad you agree"
    "n", "no" -> "Sorry to hear that"
    else -> "I don't understand you"
}
```

Any expression can be used as a branch condition

```
fun mix(c1: Color, c2: Color) =
        when (setOf(c1, c2)) {
            setOf(RED, YELLOW) -> ORANGE
            setOf(YELLOW, BLUE) -> GREEN
            setOf(BLUE, VIOLET) -> INDIGO
            else -> throw Exception("Dirty color")
        }
```

the argument is checked for equality with the branch conditions

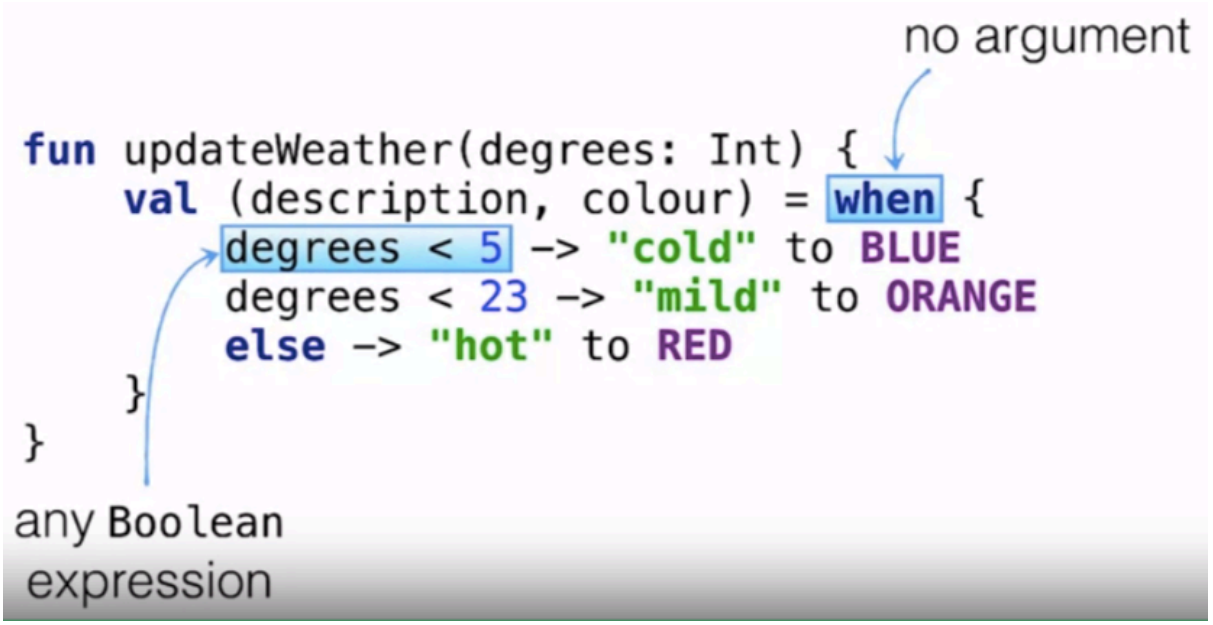
Checking types

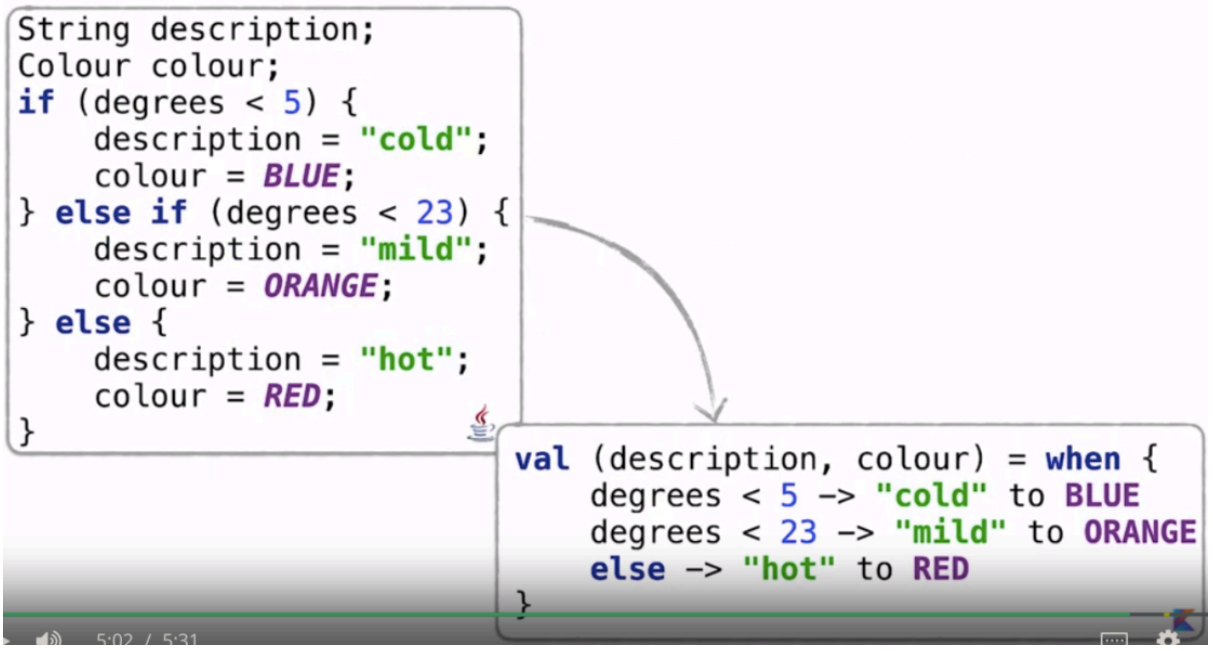


Capturing when subject in a variable

```
when (val pet = getMyFavouritePet()) {
    is Cat -> pet.meow()
    is Dog -> pet.woof()
}
```

Checking conditions: when without parameters





Loops

while

```
while (condition) {
    /*...*/
}
```

do-while

```
do {
    /*...*/
} while (condition)
```

for

```
val list = listOf("a", "b", "c")
for (s in list) {
    print(s)
}
```

```
abc
```

Iterating over a map

```
val map = mapOf(1 to "one",
                2 to "two",
                3 to "three")

for ((key, value) in map) {
    println("$key = $value")
}
```

```
1 = one
2 = two
3 = three
```

Iterating with index

```
val list = listOf("a", "b", "c")
for ((index, element) in list.withIndex()) {
    println("$index: $element")
}
```

```
0: a
1: b
2: c
```

Iterating over a range

including upper bound

```
for (i in 1..9) {
    print(i)
}
```

123456789

excluding upper bound

```
for (i in 1 until 9) {
    print(i)
}
```

12345678

Iterating with a step

```
for (i in 9 downTo 1 step 2) {
    print(i)
}
```

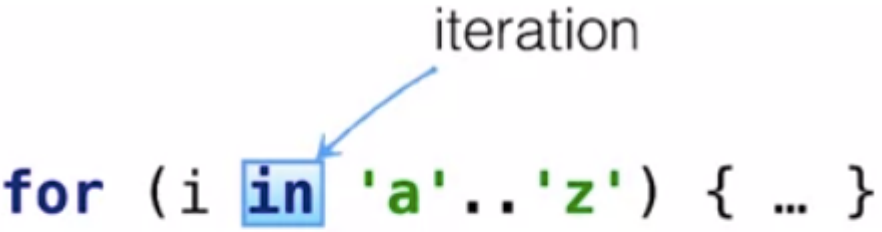
97531

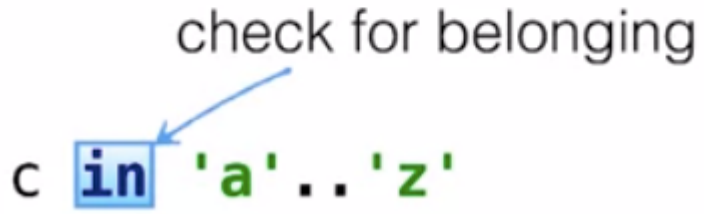
Iterating over String

```
for (ch in "abc") {
    print(ch + 1)
}
```

'in' checks & ranges

in: two uses cases





not in range

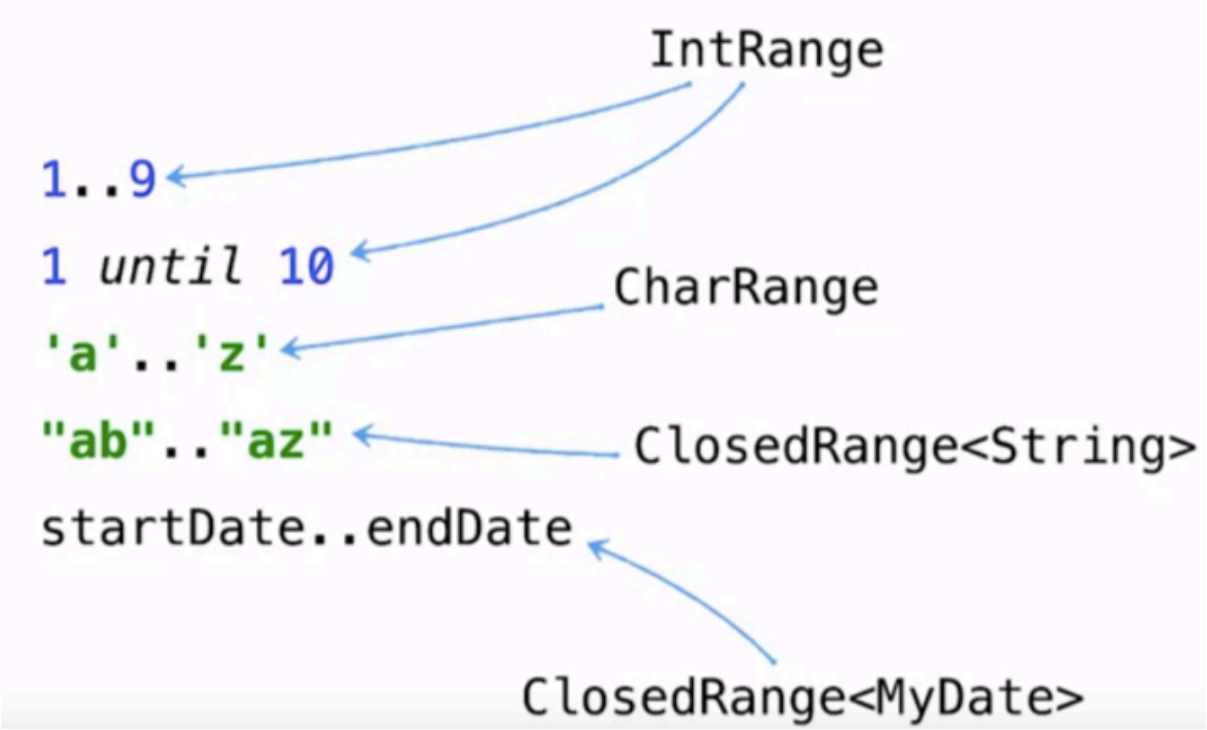
```
fun isNotDigit(c: Char) = c !in '0'..'9'
```

```
isNotDigit('x')   // true
```

in as when condition

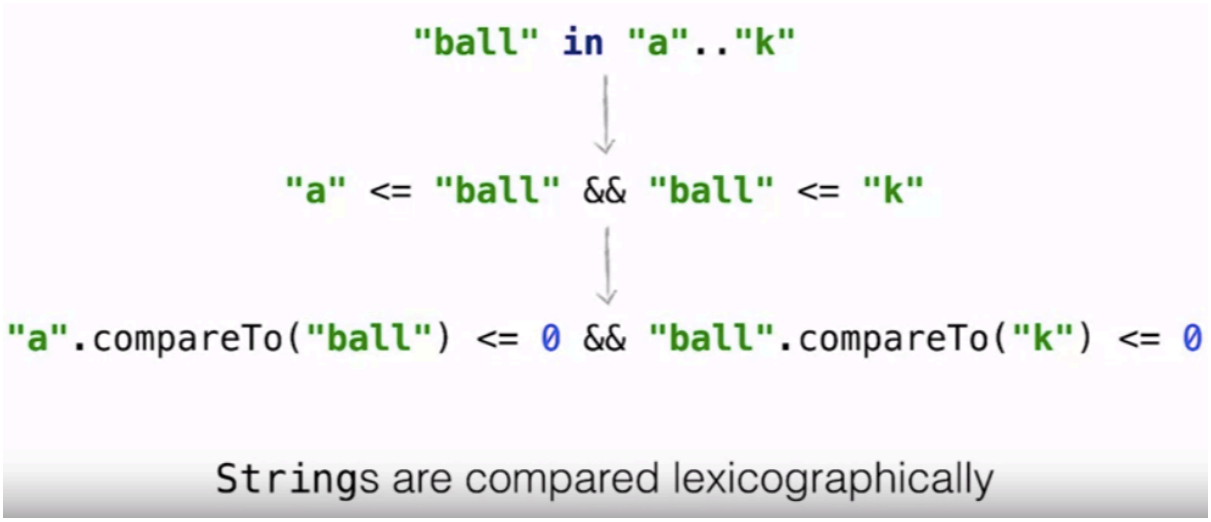
```
fun recognize(c: Char) = when (c) {
    in '0'..'9' -> "It's a digit!"
    in 'a'..'z', in 'A'..'Z' -> "It's a letter!"
    else -> "I don't know…"
}


    recognize('$')     // I don't know…
```

Different ranges



Comparing Strings

Strings are compared alphabetically




```
"ball" in "a".."k"      // true
"zoo" in "a".."k"       // false

"Kotlin" in "Java".."Scala"     // true
```

Range of Custom Types

You must implement **Comparable<T>** and then you can use **>= <=** operators or Range operator "**..**".

```
class MyDate : Comparable<MyDate>

if (myDate.compareTo(startDate) >= 0 &&
        myDate.compareTo(endDate) <= 0) { ... }

if (myDate >= startDate && myDate <= endDate) { ... }

if (myDate in startDate..endDate) { ... }
```

Belonging to collection

```
if (element in list) { ... }
```

the same as:

```
if (list.contains(element)) { ... }
```

Exceptions

No difference between checked and unchecked exceptions.

throw is an expression

```
val percentage =
    if (number in 0..100)
        number
    else
        throw IllegalArgumentException(
            "A percentage value must be" +
             "between 0 and 100: $number")
```

try is an expression

That means we can assign the result of try to a variable. Here we assign the result of parseInt function to a number variable if everything is okay. We return from the outer function if the exception was throw. Instead of completing the function entirely from the catch block, we can assign now to a number variable as a result.

```
val number = try {
    Integer.parseInt(string)
} catch (e: NumberFormatException) {
    null
}
```

Extension Functions

Extension function extends the class. It is defined outside of the class but can be called as a regular member to this class.

```
fun String.lastChar() = this.get(this.length - 1)
```

```
val c: Char = "abc".l
                λ lastChar() for String in com.example.ut
                λ last() for CharSequence in kotlin.text
```

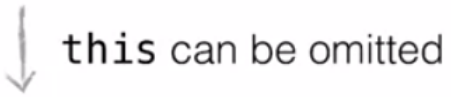
The type that the function extends is called a Receiver. Here, string is the receiver of the lastChar function. In the body of this function, we can access the receiver by this reference. In our example, this refers to string. As usual for this reference, we can omit it. We can call members of the receiver inside an extension function without explicit this dot something specification. An important thing to note here is that you can't define an extension and use it everywhere. You have to import it explicitly. If you define an extension function lastChar somewhere and need to use it, you have to import it either as one function, or inside the whole contents of the package

Receiver

```
fun String.lastChar() = this.get(this.length - 1)
```

Member access

```
fun String.lastChar() = this.get(this.length - 1)
```




```
fun String.lastChar() = get(length - 1)
```

Accessing private members

- In Java you can't call a private member from a static function *of another class*
- Kotlin extension functions are regular static functions defined in *a separate auxiliary class*
- You can't call private members from extensions

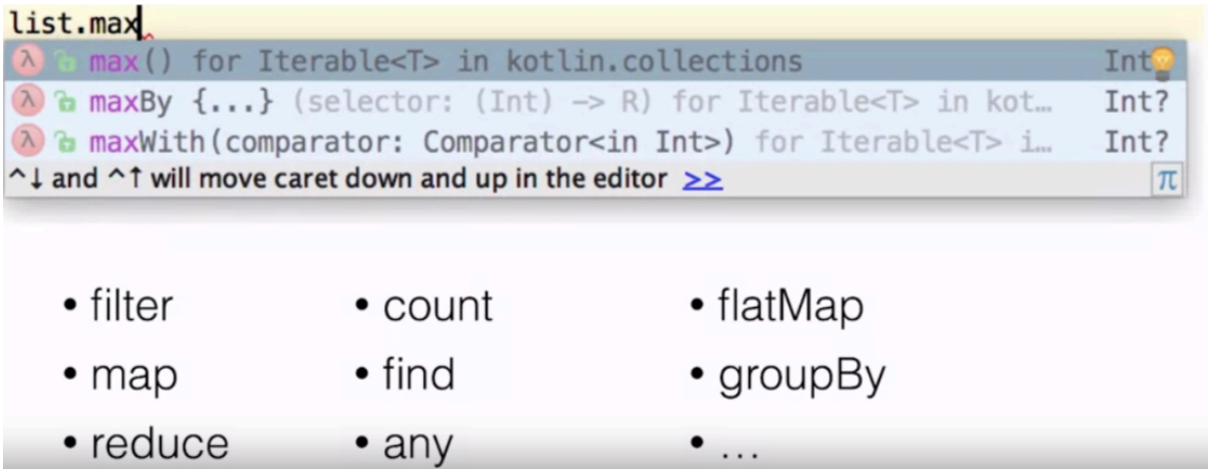
Importing extension

```
fun String.lastChar() = get(length - 1)
```

```
import com.example.util.*
val c: Char = "abc".lastChar()
```

Examples from the Standard Library

Kotlin library: extension on collections



joinToString

```
println(listOf('a', 'b', 'c').joinToString(
        separator = "", prefix = "(", postfix = ")"))
```

```
(abc)
```

```
fun <T> Iterable<T>.joinToString(
        separator: CharSequence = ", ",
        prefix: CharSequence = "",
        postfix: CharSequence = ""
): String
```

getOrNull

```
val list = listOf("abc")
println(list.getOrNull(0))  // abc
println(list.getOrNull(1))  // null
```

```
fun <T> List<T>.getOrNull(index: Int) =
    if (index in 0 until size) this[index] else null
```

withIndex

```
val list = listOf("a", "b", "c")
for ((index, element) in list.withIndex()) {
    println("$index $element")
}
```

```
fun <T> Iterable<T>.withIndex(): List<IndexedValue<T>> { … }
```

until

```
infix fun Int.until(to: Int): IntRange
```

```
1.until(10)
```

```
1 until 10
```

to

```
infix fun <A, B> A.to(that: B) = Pair(this, that)
```

```
"ANSWER".to(42)
```

```
"hot" to RED
```

```
mapOf(0 to "zero", 1 to "one")
```

Extension on Strings

Formatting multiline Strings

```
val q = """To code,
    or not to code?.."""


println(q)      To code,
                ____or not to code?..
```

trimMargin

```
val q = """To code,
    |or not to code?..""".trimMargin()


println(q)      To code,
                or not to code?..
```

marginPrefix

```
val q = """To code,
    #or not to code?..""".trimMargin(marginPrefix = "#")


println(q)      To code,
                or not to code?..
```

trimIntent

```
val q = """To code,
    |or not to code?..""".trimMargin()
val a = """
    Keep calm
    and learn Kotlin""".trimIndent()

println(q)      To code,
                or not to code?..

println(a)      Keep calm
                and learn Kotlin
```

Using Regular Expressions

```
val regex = """\d{2}\.\d{2}\.\d{4}""".toRegex()
regex.matches("15.02.2016")          // true
regex.matches("15.02.16")            // false
```

Conversion to Numbers

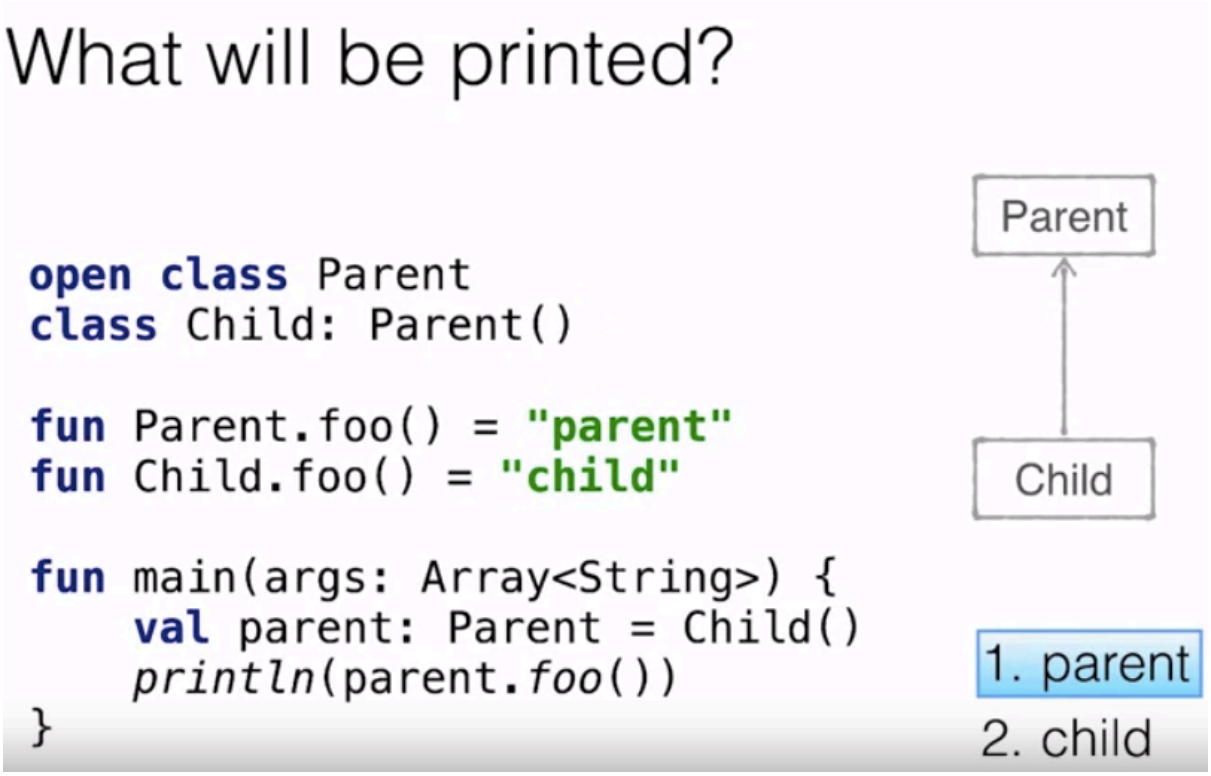
```
"123".toInt()          // 123
"1e-10".toDouble()     // 1.0E-10


"xx".toInt()           // NumberFormatException


"123".toIntOrNull()    // 123
"xx".toIntOrNull()     // null
```

Example of extension function (eq)

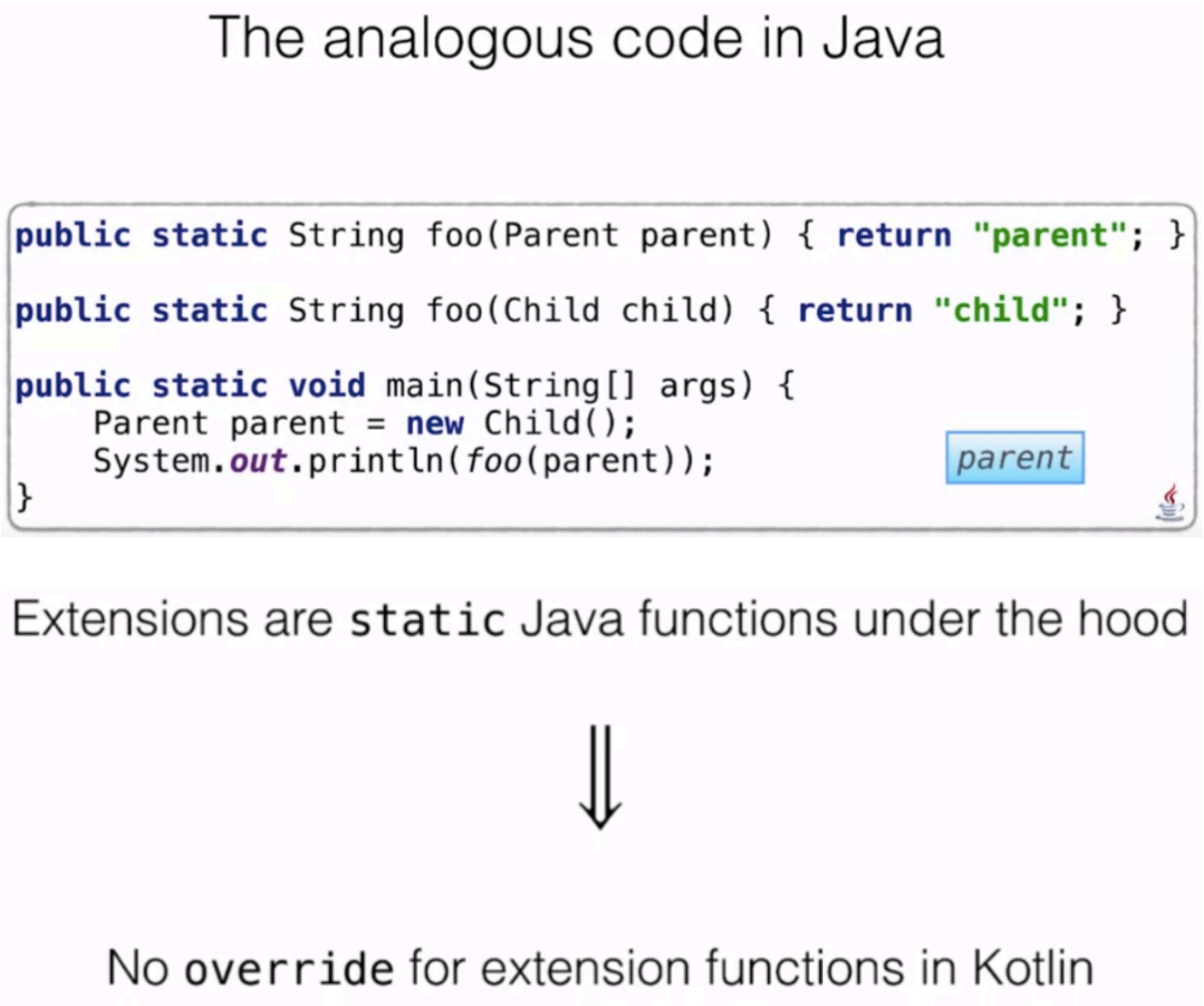
```
infix fun <T> T.eq(other: T) {
    if (this == other) println("OK")
    else println("Error: $this != $other")
}


fun getAnswer() = 42

getAnswer() eq 42   // OK
getAnswer() eq 43   // Error: 42 != 43
```

Calling Extensions (Inheritance)

Let's say we have two classes, parent and child and child extends parent. We can define two similar extension functions. The first one to parent, the second one to child. We create an instance of a child and store it in a reference of the parent type. The question for you is which foo function will be called in this case?

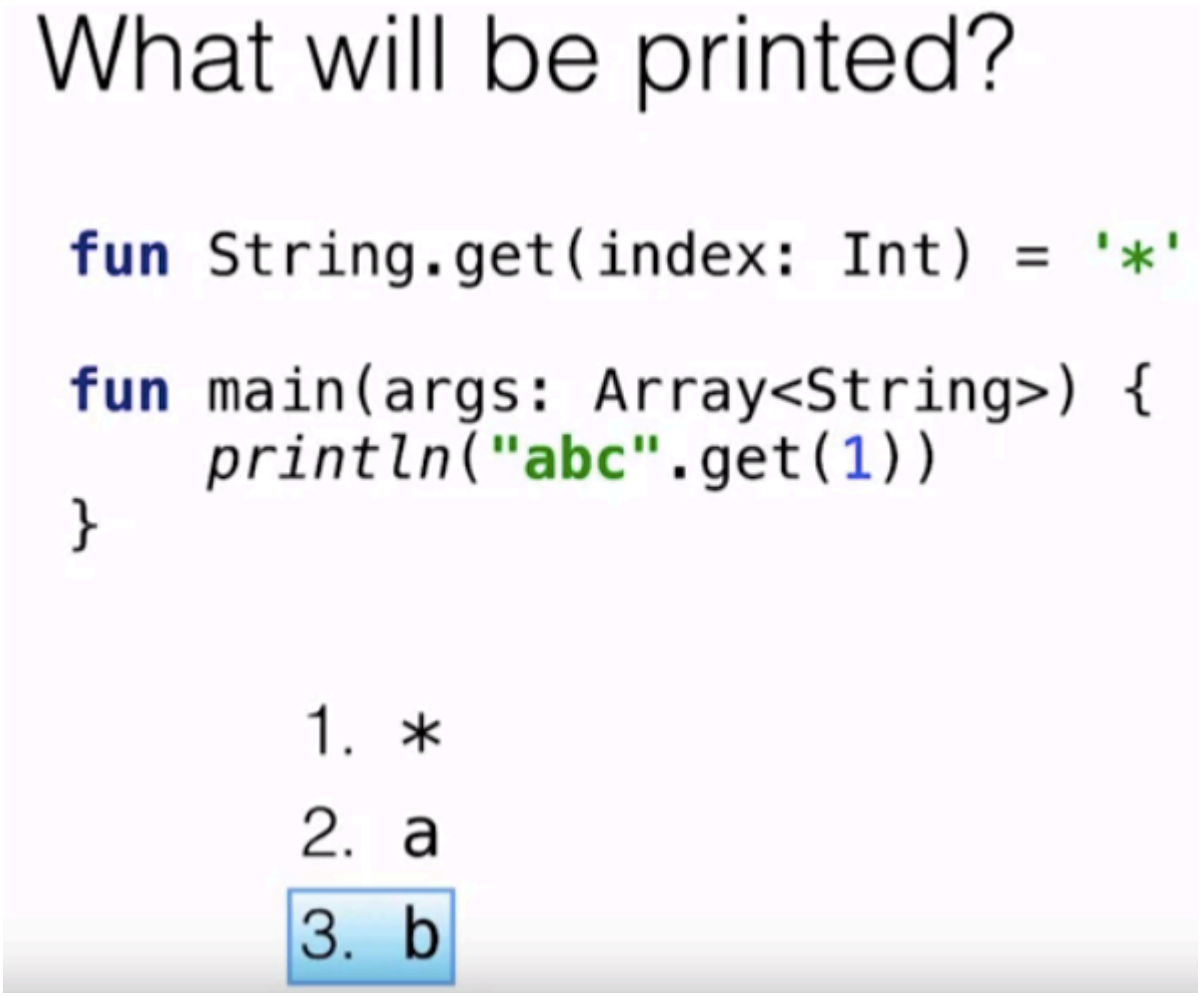


The resolution of which function should be called works in a similar manner as for Java static functions. You can convert this example into Java and answer the same question for static functions. Under the hood, these extension functions are compiled to Java static functions. The first one **takes parent as a parameter**, the second one **takes child as a parameter**. When extension functions are compiled, **the type of the receiver is transformed to the type of the first additional parameter**. Java resolves static function statically. It finds the right function to be called during the compilation. It only uses the type of the argument to choose the right function, thus the parent function is chosen here since the parent variable has the parent type.



Members vs Extensions

Now, the interesting question is what happens if we try to define an extension which duplicates a member? For instance, the string class has a member function get, which simply returns a character by its index. We tried to define an extension with the same signature. What do you think will be printed here?



The right answer is b, member always wins.

Overload members

However, you can overload a member. If you define an extension with the different signature, different parameter types or the different number of parameters, your new function will be called if it fits better.

```
class A {
    fun foo() = "member"
}

fun A.foo(i: Int) = "extension($i)"


A().foo(2)    // extension(2)
```

Nullability

Nullable types

```
val s1: String = "always not null"

val s2: String? = null


s1.length    ✓

s2.length    ✗
```

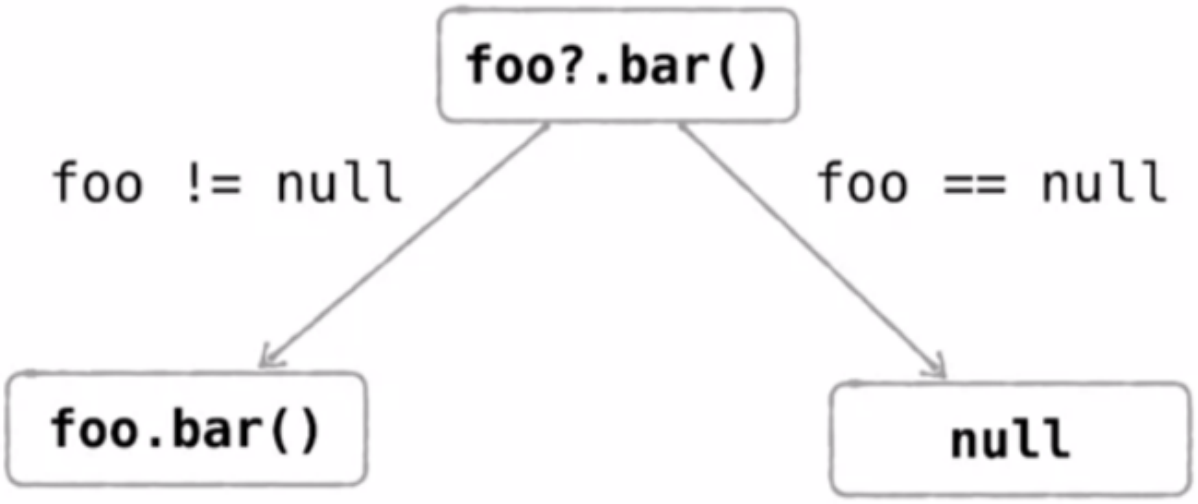
Dealing with nullable types

```
val s: String?

if (s != null) {
    s.
}
```

Replace 'if' expression with safe access expression

```
s?.length
```

Safe access



Nullability operators

```
val s: String?

val length: Int? = if (s != null) s.length else null
```

```
val length: Int? = s?.length
```

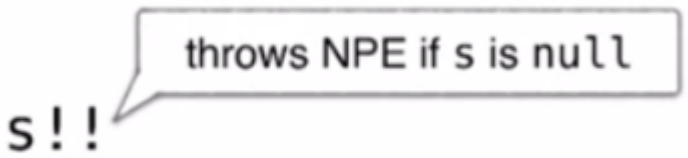
Convert to Nonnull Int

```
val s: String?

val length: Int = if (s != null) s.length else 0
```

```
val length: Int = s?.length ?: 0
```

Making NPE Explicitly

```
val s: String?
```




```
s.length
```

Bad exmample of non-null assertion

```
person.company!!.address!!.country
```

Prefer

?. ?: if-checks

to

!!

Operator Precedence

Precedence	Title	Symbols
Highest	Postfix	++, --, ., ?., ?
	Prefix	-, +, ++, --, !, labelDefinition
	Type RHS	:, as, as?
	Multiplicative	*, /, %
	Additive	+, -
	Range	..
	Infix function	SimpleName
	Elvis	?:
	Named checks	in, !in, is, !is
	Comparison	<, >, <=, >=
	Equality	==, \!==
	Conjunction	&&
	Disjunction	||
Lowest	Assignment	=, +=, -=, *=, /=, %=

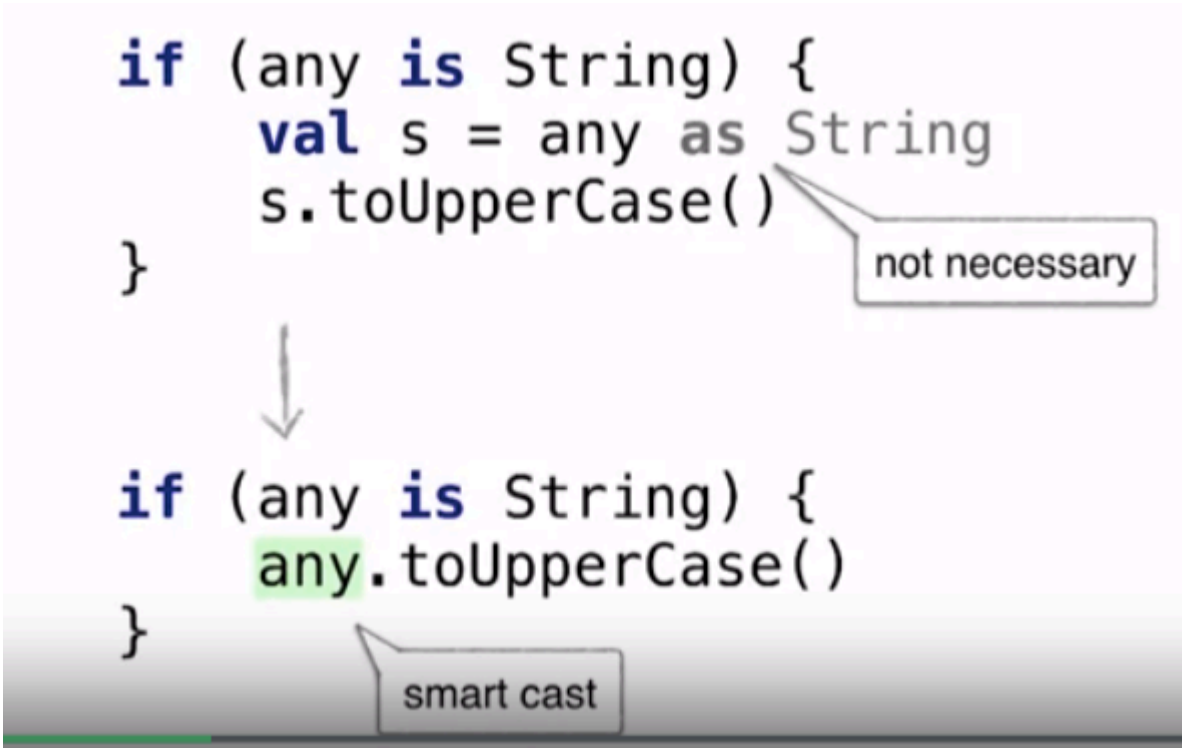
Prefer the parentheses

```
val x: Int? = 1

val y: Int = 2

val s1 = x ?: 0 + y       // 1

val s2 = x ?: (0 + y)     // 1

val s3 = (x ?: 0) + y     // 3
```

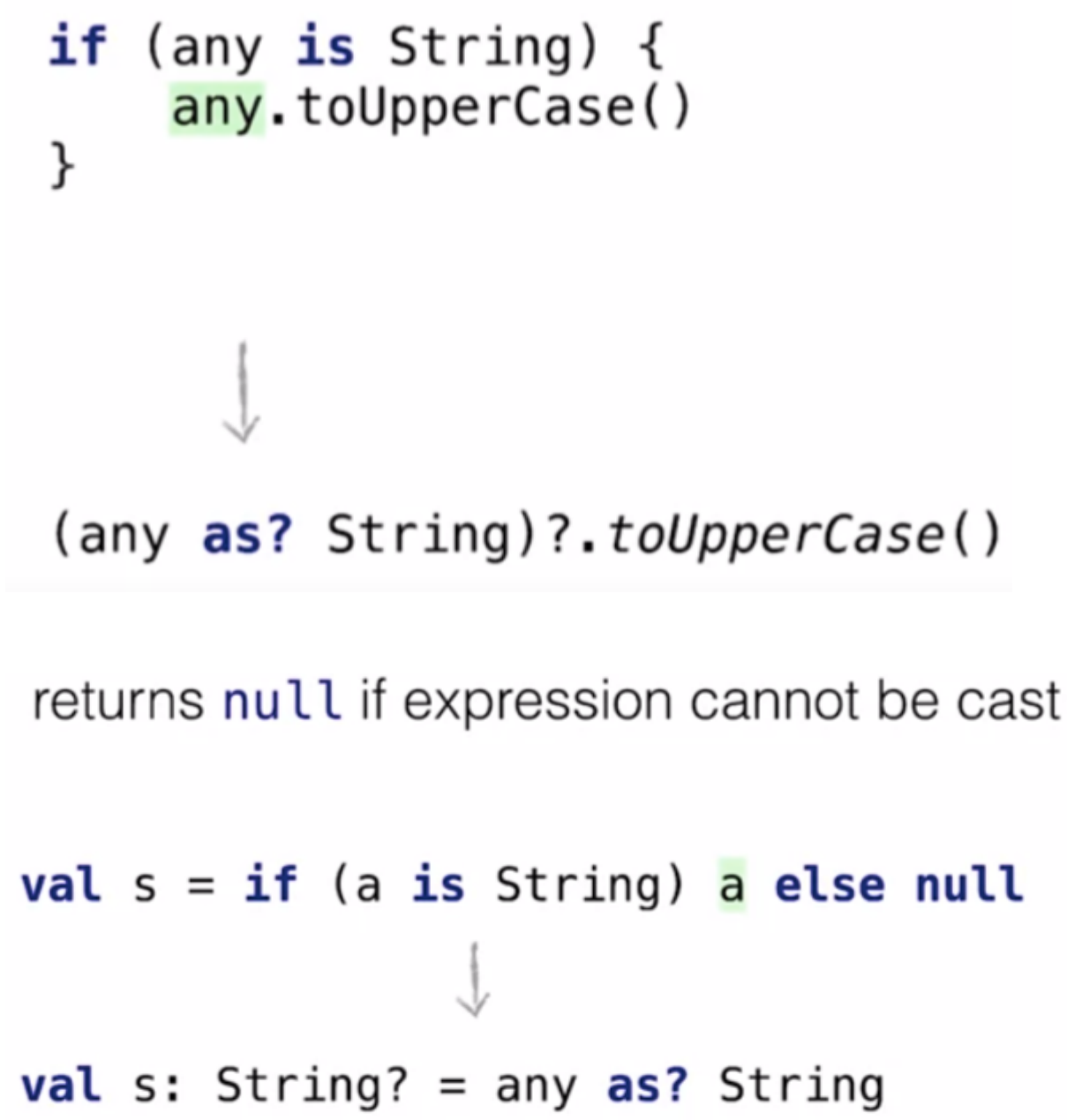
Safe casts

A safe way to cast an expression to a type.

Type cast: as



Safe cast: as?



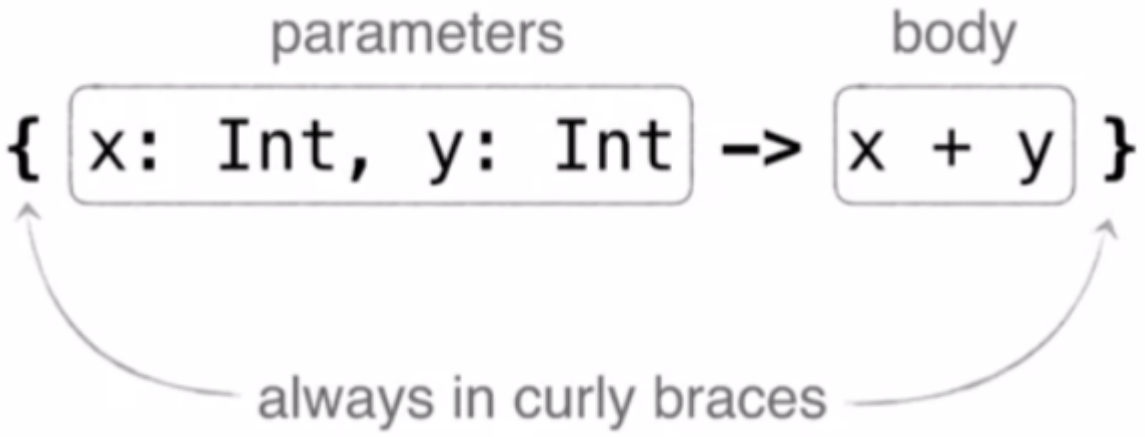
Functional Programming

Lambdas

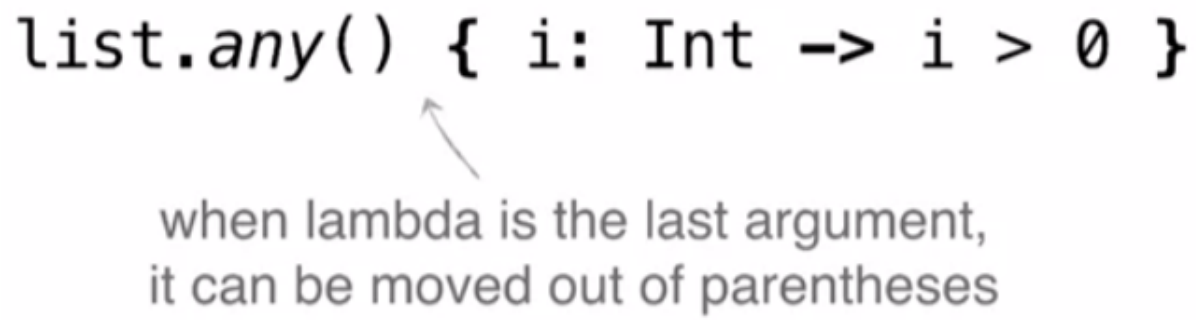
Lambda is an anonymous function that can be used as an expression, for instance, passed as an argument to another function vacation.

```
button.addActionListener { println("Hi") }
```

Syntax



If you pass a Lambda as an argument, you can put the whole Lambda inside the parentheses. However, there is a better way to express that. **You can move Lambda out with parentheses, if the Lambda is the last argument, and if the parentheses are empty, you can omit them.**



If the type of the argument can be inferred, if it's clear from the context, it can be omitted. If your Lambda has their own argument, you can replace its name with it.

```
list.any { it > 0 }
```

`it` denotes the argument if it's only one

Multi-line lambda

```
list.any {
    println("processing $it")
    it > 0
}
```

the last expression is the result

Destructing declarations

```
map.mapValues { (key, value) -> "$key -> $value!" }
```

destructuring declarations syntax

Omit unused parameters

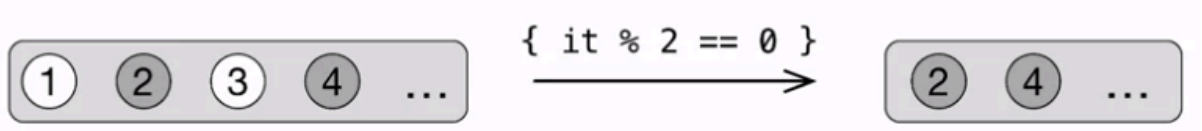
```
map.mapValues { (_, value) -> "$value!" }
```

omit the parameter name if the parameter is unused

Common Operations on collections

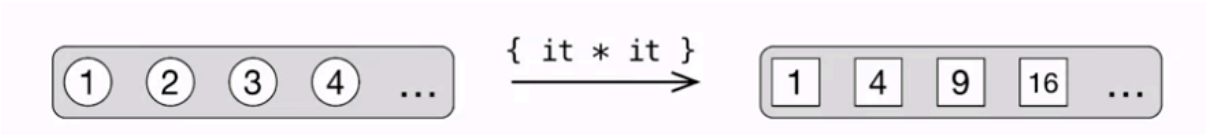
Filter

It filters out the content of the list and keeps only their elements that satisfy the given predicate



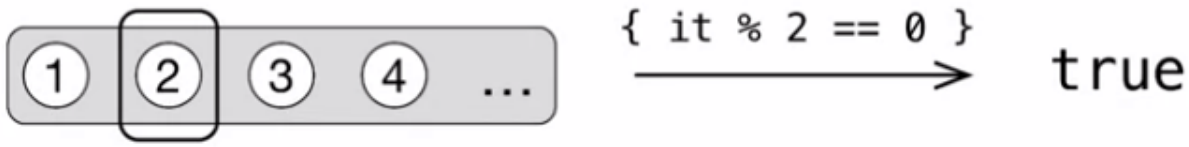
Map

Map transform each element in a collection and stores all the resulting elements in a new list.



Any, All, None

There are several predicates checking whether the given facts about the elements are true. For instance, **any** checks that there is at least one element satisfying the given predicate, here, we check whether there is at least one even number in the least and the result is true. **All** checks whether all elements satisfy the predicate and **none** makes sure that none of the elements satisfies the given predicate.



Find

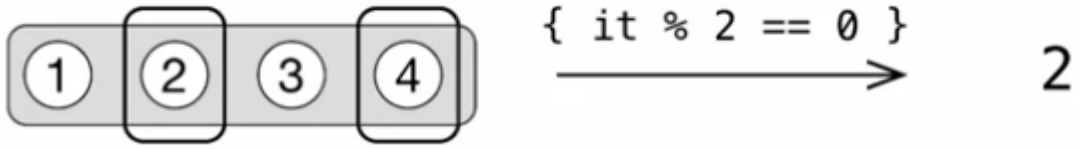
Find finds an element satisfying the given predicate and returns it as a result. If there is no required element, find returns null.

First / FirstOrNull

FirstOrNull does the same as find, it returns you an element or null as a result. First takes a predicate and throws an exception if no elements satisfying the predicate was found.

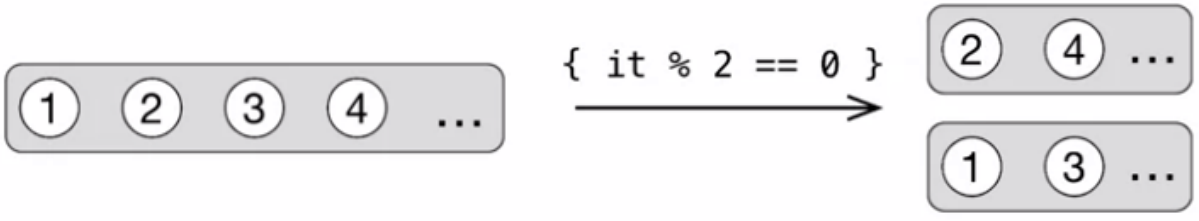
Count

Count counts the number of elements that satisfy the given predicate.



Partition

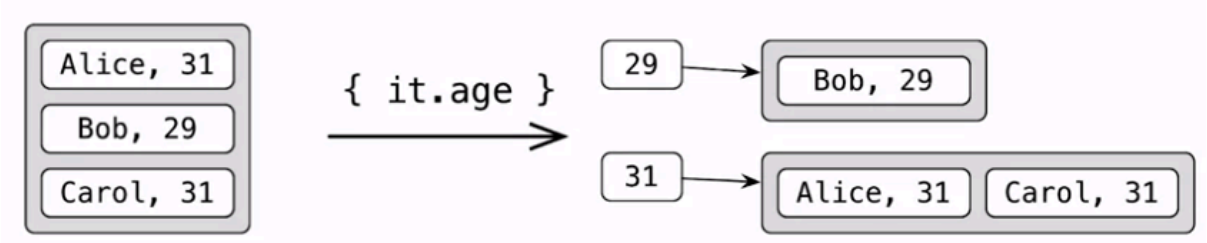
Partition divides the collection into two collections. It returns two collections, for the good elements and the remaining ones



GroupBy

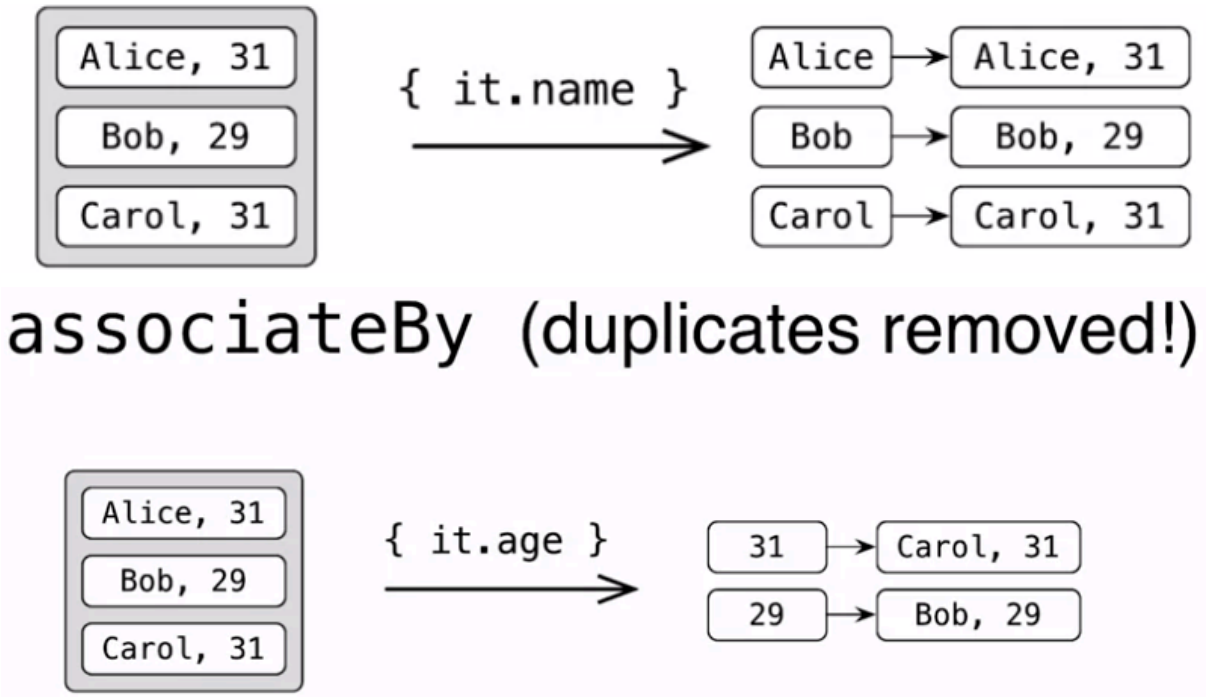
If you need to divide your collection into more than two groups, you can use GroupBy. As an argument, you provide the way how to group the elements. What should be the grouping

key? For instance, here we group personal elements by their age. The result is, map from the given key to a list of elements that satisfy this key.



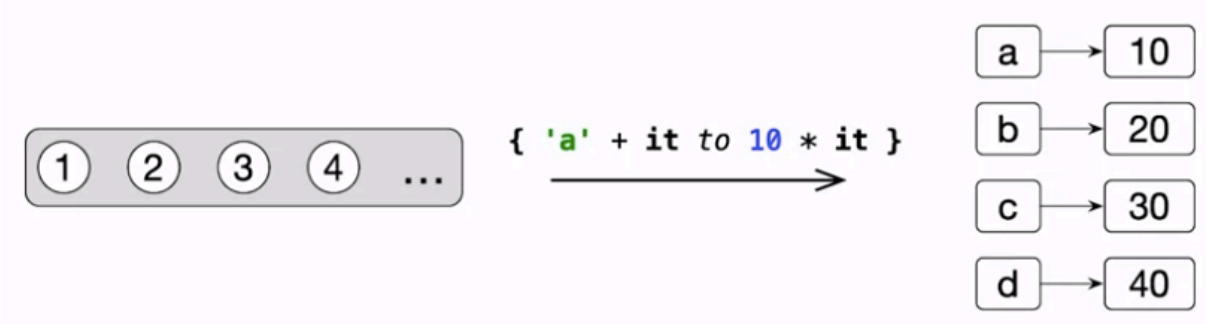
AssociateBy

If the key is unique, then it's more useful to have a map of the key to this unique element as a result. That's what they associateBy function does for you. It also performs groping, but it returns you one element as the map value. Note that, **associateBy should be used to run the keys unique**. If it's not, pay attention that duplicates are removed, so the last element will be chosen



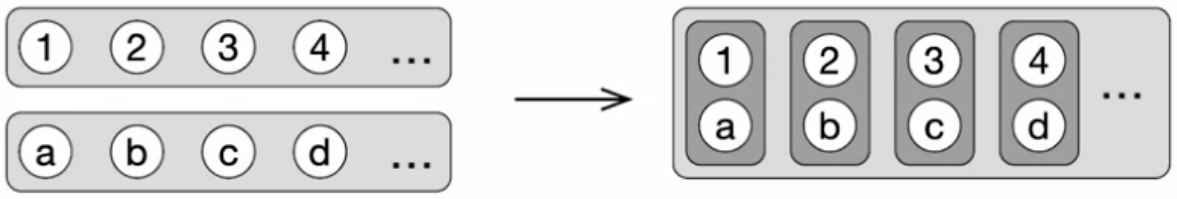
Associate

You can use as associate to build a map based on a list. As an argument, you pass allowed to creating the key value pair based on each list element, then associate builds a map by filling in specified keys and values. The first value in a pair becomes key in the map, the second becomes the value



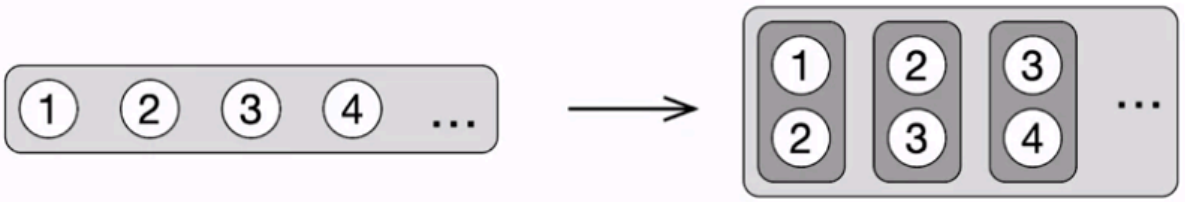
Zip

Zip provides you with a way to organize a couple of lists. It zips like a zipper the elements from two lists. It returns you a list of pairs where each pair contains one element from the first list and another element from the second list. If their initial list have different sizes, then the resulting list of pairs will have the length of the shortest list, the remaining elements from the longest list will be ignored.



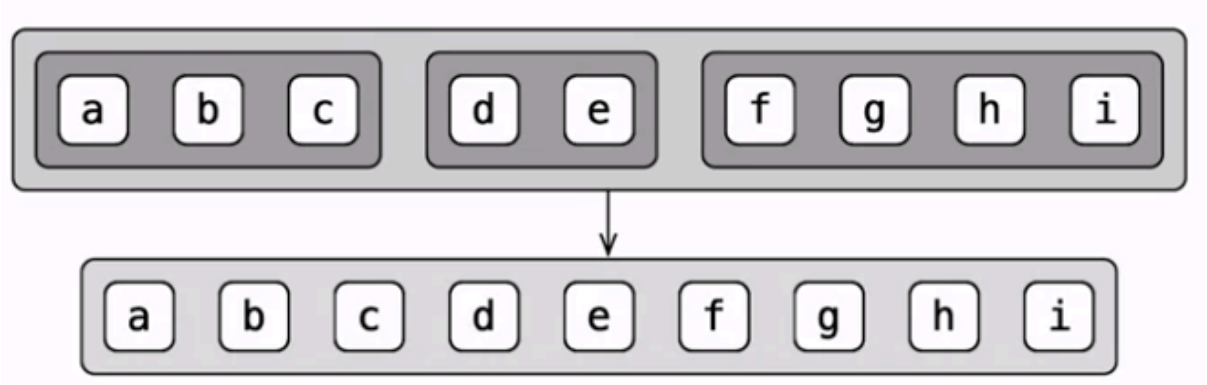
ZipWithNext

It returns you a list of pairs where each pair consists of neighboring elements is from the initial list. Note that, each element except the first and the last one will belong two pairs.



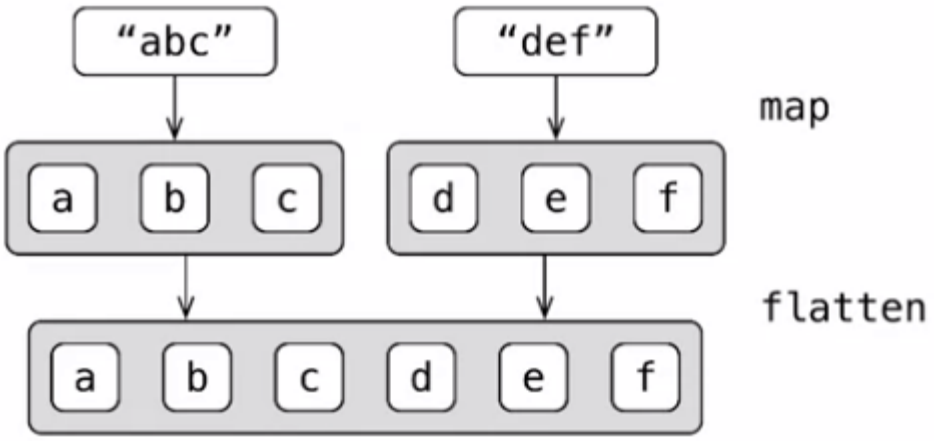
Flatten

Flatten is an extension function that must be called on a list of lists. It combines all the elements from the nested list and returns you a list of these elements as the result.



FlatMap

It combines two operations, map and flat. The argument to flatMap must be a lambda that converts each initial element to a list. Here, we first map each string into a list of characters. In the middle layer after applying the first map operations, we have at list of lists. Often, you'd prefer list of elements as a result instead and flatten does that. Here, flatMap returns a list of characters obtained from initial strings



Function Types

In Kotlin, you can store Lambda in a variable, which type does the variable have in this case? If we specify this type explicitly, we'll see a so-called function type. First, parameter types are written inside the parentheses and then an arrow, then the return type. In this case, is the type that takes two integer parameters and returns an integer as a result.

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

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

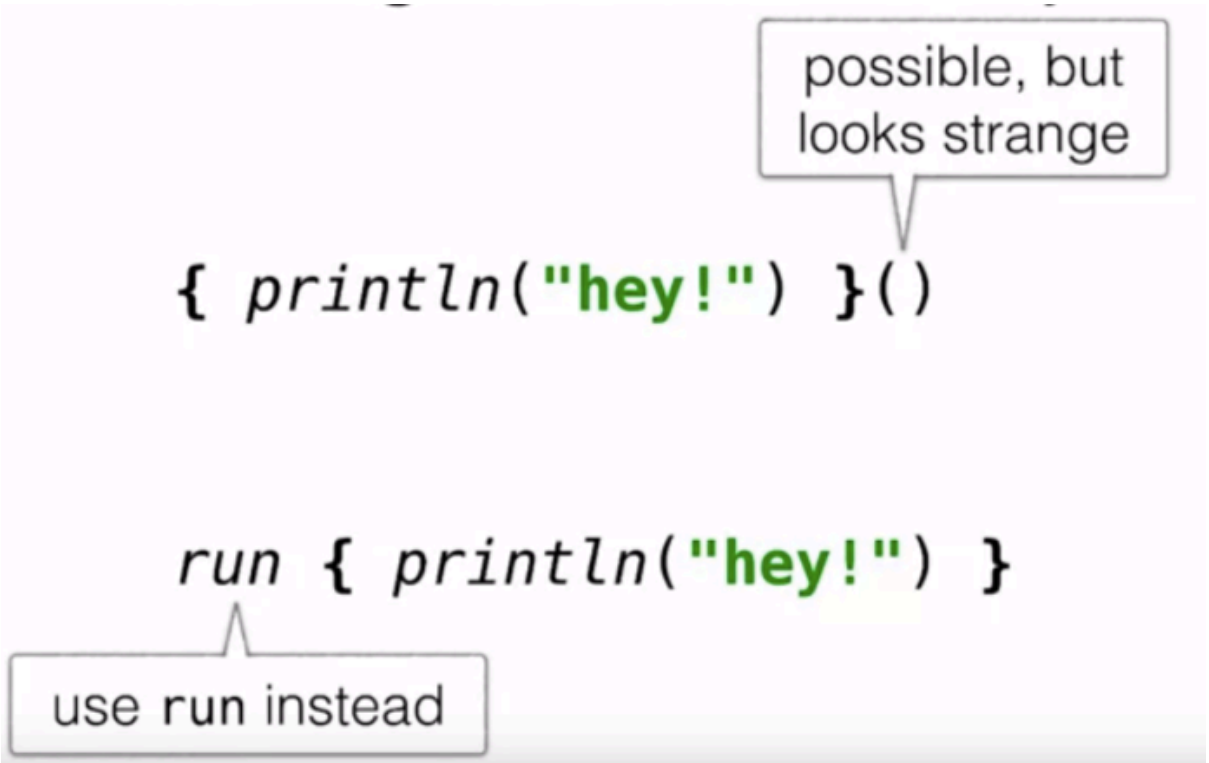
Calling stored function

```
val isEven: (Int) -> Boolean =
        { i: Int -> i % 2 == 0 }

val result: Boolean = isEven(42)   // true
```

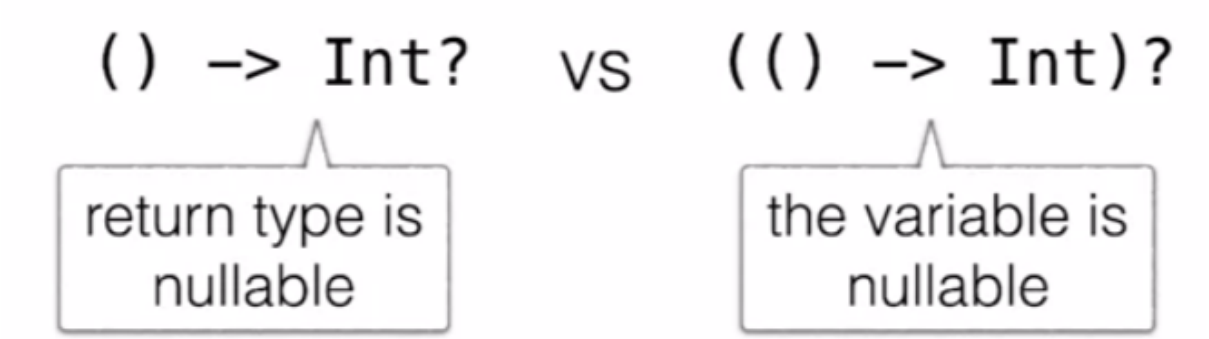
Function type as argument

```
val isEven = { i: Int -> i % 2 == 0 }

val list = listOf(1, 2, 3, 4)
list.any(isEven)                    true
list.filter(isEven)                 [2, 4]
```

Calling lambda directly



Function types and nullability



Working with a nullable function type

```
val f: (() -> Int)? = null

f()

if (f != null) {
    f()
}

f?.invoke()
```

Member References

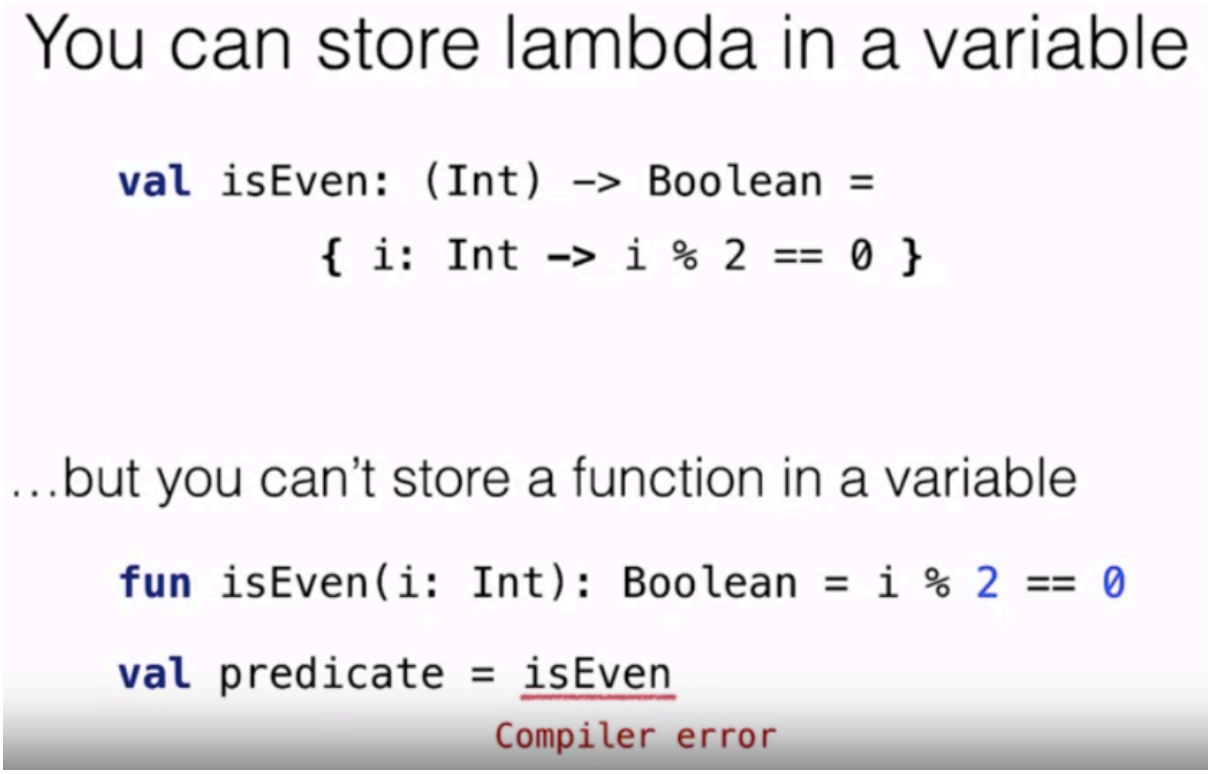
Like Java, Kotlin has member references, which **can replace simple Lambdas that only call a member function or return a member property**, it can convert Lambda to member reference automatically when it's possible. The syntax for member reference is the same as in Java. **First goes the class name, then the double colon, then the member which we refer**.

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

people.maxBy { it.age }

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

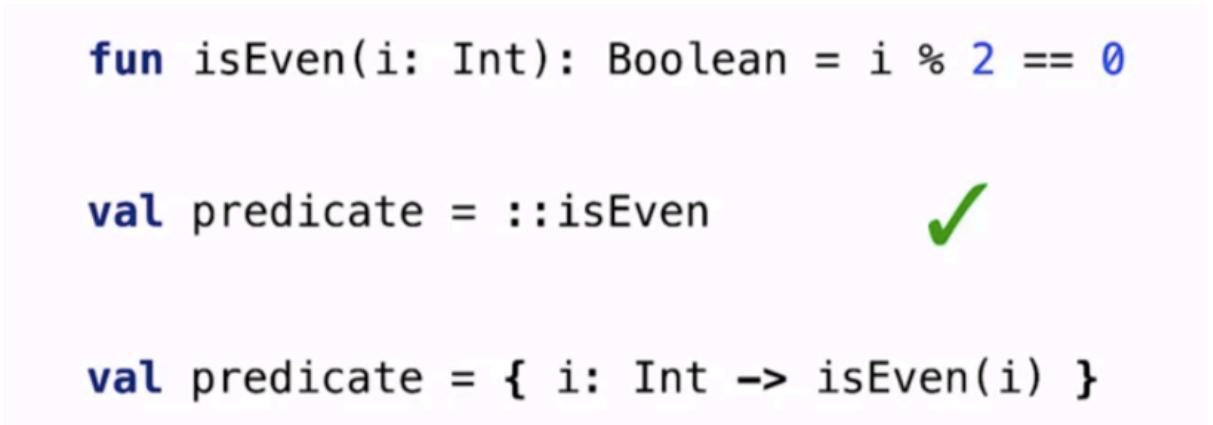
class

member

As we've previously discussed in Kotlin, **you can store Lambda in a variable**, **however, you can't store a function in a variable**. It's not like in a truly functional language where each function is a variable. No, in Kotlin **there is a clear distinction between functions and variables**, that are connected with how JVM works under the hood. If you try to assign a function to a variable, you'll get a compiler error



To fix this issue, use the **function reference syntax**. **Function references allow you to store a reference to any defined function in a variable** to be able to store it and qualitative it. Keep in mind that **this syntax is just another way to call a function inside the Lambda**, underlying implementation are the same.



If a reproach member is a property, or it's a **function that takes zero or one argument**, **then member reference syntax isn't that concise in comparison with the explicit Lambda** syntax. However, **if the reproached function takes several arguments, you have to repeat all the parameter names as Lambda parameters, and then explicitly pass them through**, that makes this syntax robust. Member references allow you to hide all the parameters, because the compiler infers the types for you.

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


val action = ::sendEmail
```

Function reference as argument

You can pass a function reference as an argument, whenever your Lambda tends to grow too large and to become too complicated, it makes sense to extract Lambda code into a separate function, then you use a reference to this function instead of a huge Lambda

```
fun isEven(i: Int): Boolean = i % 2 == 0


val list = listOf(1, 2, 3, 4)
list.any(::isEven)                  true
list.filter(::isEven)               [2, 4]
```

Non-bound & bound references:

In Kotlin, you can create a boundary reference, let's us discuss what it is. In this example, we use a regular non-bound reference, which refers to a member of the person class. If we check what type this member reference has, we see that the first argument of the function type is person. Whenever we want to call this variable or function tab, we need to pass the person instance explicitly. If we look under the hood and check what Lambda does correspond to this member reference, we'll find that this Lambda takes two arguments, person and age limit, it simply calls the member function is older inside on the past personnel element. **This reference is called non bound, since it's not bound to any specific instance**, you can call it on any object of the person class

Non-bound reference

```
class Person(val name: String, val age: Int) {
    fun isOlder(ageLimit: Int) = age > ageLimit
}

val agePredicate: (Person, Int) -> Boolean =
                            Person::isOlder
val alice = Person("Alice", 29)
agePredicate(alice, 21)        // true
```

Corresponding Lambda

```
class Person(val name: String, val age: Int) {
    fun isOlder(ageLimit: Int) = age > ageLimit
}

val agePredicate: (Person, Int) -> Boolean =
                            { person, ageLimit ->
                              person.isOlder(ageLimit) }
val alice = Person("Alice", 29)
agePredicate(alice, 21)        // true
```

Bound reference

Bound reference is a member reference that is attached to a specific instance of the class. Here, the alice variable is an instance of the class person, and you **can bound member efforts to these specific instance**. If we look at the type of the bound member reference, we see that now there is no person parameter because the person instance is already set, the older function takes int as a parameter, and the bound reference also expect only int as an argument. If we check which Lambda corresponds to this member reference to the hood, we'll find the Lambda that calls the member is older on the bound instance. In this case, this bound instance is Alice variable.

```
class Person(val name: String, val age: Int) {
    fun isOlder(ageLimit: Int) = age > ageLimit
}


val alice = Person("Alice", 29)
val agePredicate = alice::isOlder
agePredicate(21)              // true
```

Corresponding Lambda

```
class Person(val name: String, val age: Int) {
    fun isOlder(ageLimit: Int) = age > ageLimit
}

val alice = Person("Alice", 29)
val agePredicate: (Int) -> Boolean =
                  { ageLimit -> alice.isOlder(ageLimit) }
agePredicate(21)              // true
```

Bound to this reference

Member reference can be bound to **this** reference. Here, we'll return a predicate directly from the class person. **This predicate is a member reference to his older function**, that **this is the object of which this member reference is bound to**, and is usual for these we can omit it.

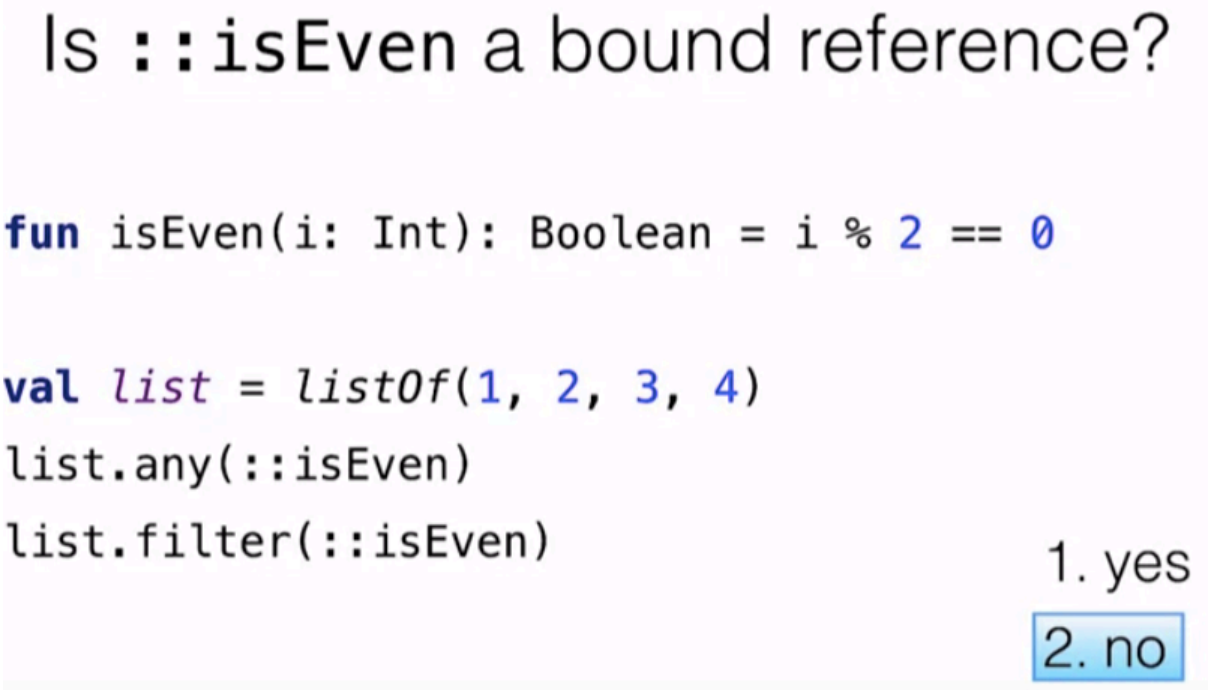
```
class Person(val name: String, val age: Int) {
    fun isOlder(ageLimit: Int) = age > ageLimit

    fun getAgePredicate() = this::isOlder
}
```

this can be omitted

Now, we have this nice short syntax without left-hand side, just reference to the member.

```kotlin
class Person(val name: String, val age: Int) {
    fun isOlder(ageLimit: Int) = age > ageLimit

    fun getAgePredicate() = ::isOlder
}
```

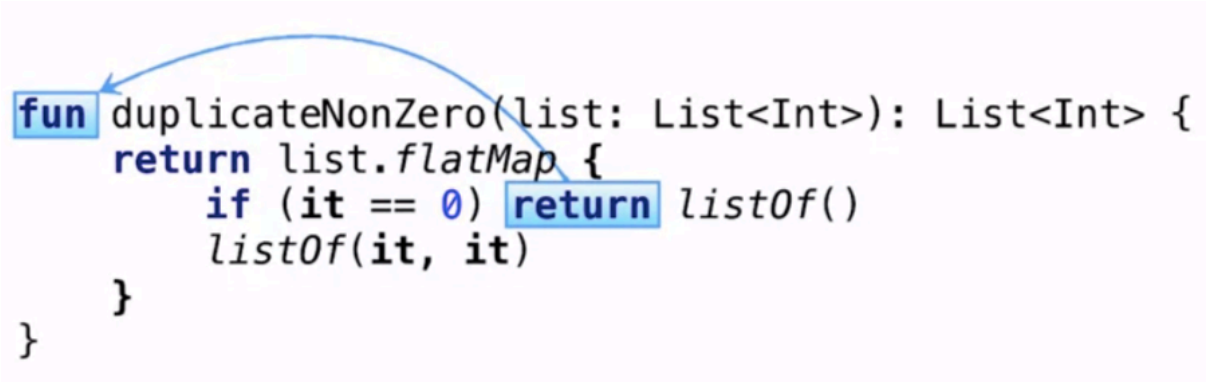
Check Question



The answer is no, because **here it's just a reference to a top-level function**. **There is no stored object in which this function is bound to**. **Whenever you see the banks and reference without the left-hand side, it's either a reference to a top-level function or a bound reference**. In ID you **can always navigate and see which faction that this function reference refers to**, it's a very convenient

return from Lambda

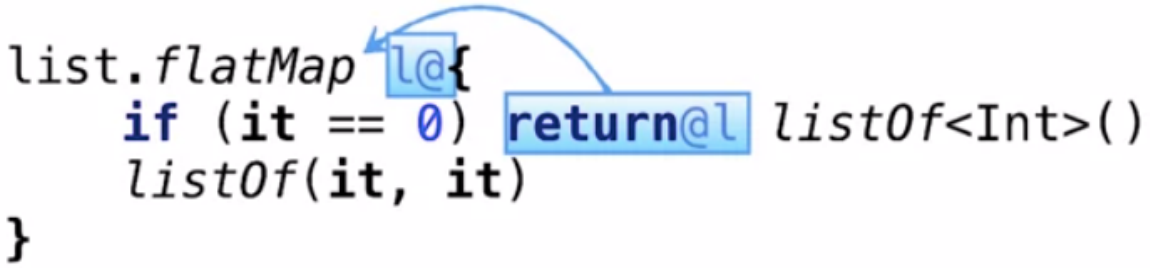
Return in Kotlin **always returns from a function** marked with **fun.**



Labeled return

What can you do if you need to return from and lambda? You can use the **labels returns syntax**. Here, the label return will return from the corresponding lambda. By default, you can use the name of the function that calls this lambda as a label.

But if you want, you can specify any **label name put it right before the lambda followed by add sine**. Here we provided the customer name just L to label the Lambda. You then use this name in a labeled Return



Using a local function

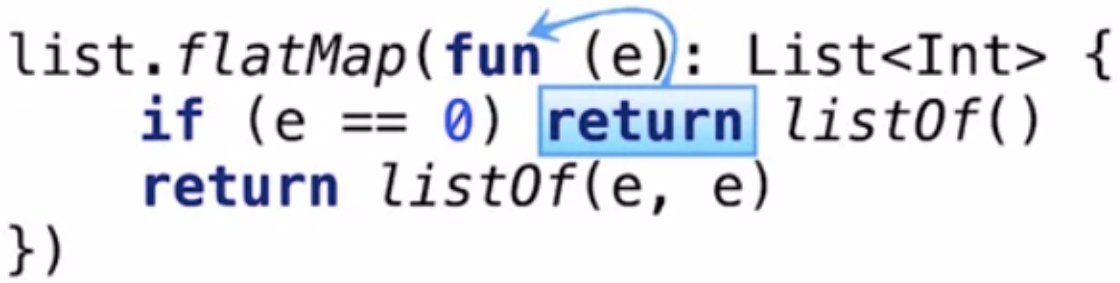
There is another solution to the same problem. Instead of lambda, you can **use a local function and then pass the reference to this local function**. In this case, return behaves as expected. It returns from the local function marked with the pond keyboard

```
fun duplicateNonZeroLocalFunction(list: List<Int>): List<Int> {
    fun duplicateNonZeroElement(e: Int): List<Int> {
        if (e == 0) return listOf()
        return listOf(e, e)
    }
    return list.flatMap(::duplicateNonZeroElement)
}

    println(duplicateNonZero(listOf(3, 0, 5)))
```

```
[3, 3, 5, 5]
```

Anonymous function

Now, the alternative solution is to use an anonymous function. If you don't want to invent a name for a local function, you can use an anonymous function instead. You can define an anonymous function in place, similar to the lambda when you pass it as an argument.



No return

Sometimes, it's possible to simply avoid return. In this case, we use the result of if instead of return and we get what we expect as a result.

```
fun duplicateNonZero(list: List<Int>): List<Int> {
    return list.flatMap {
        if (it == 0)
            listOf()
        else
            listOf(it, it)
    }
}

println(duplicateNonZero(listOf(3, 0, 5)))
```

```
[3, 3, 5, 5]
```

Properties

When you use a property in Kotlin, you don't call getters or setters. You use properties directly and access it like a variable. When you access the property by its name, the getter is called under the hood. If you want to change its value to call a setter, you do it like it was a variable. Under the hood, the setter is called.

Read-only & mutable properties

```
property = field + accessor(s)

val = field + getter

var = field + getter + setter
```

A property without a field

Backing field might be absent

```
class Rectangle(val height: Int, val width: Int) {

    val isSquare: Boolean
        get() {
            return height == width
        }
}


val rectangle = Rectangle(2, 3)
println(rectangle.isSquare)   // false
```

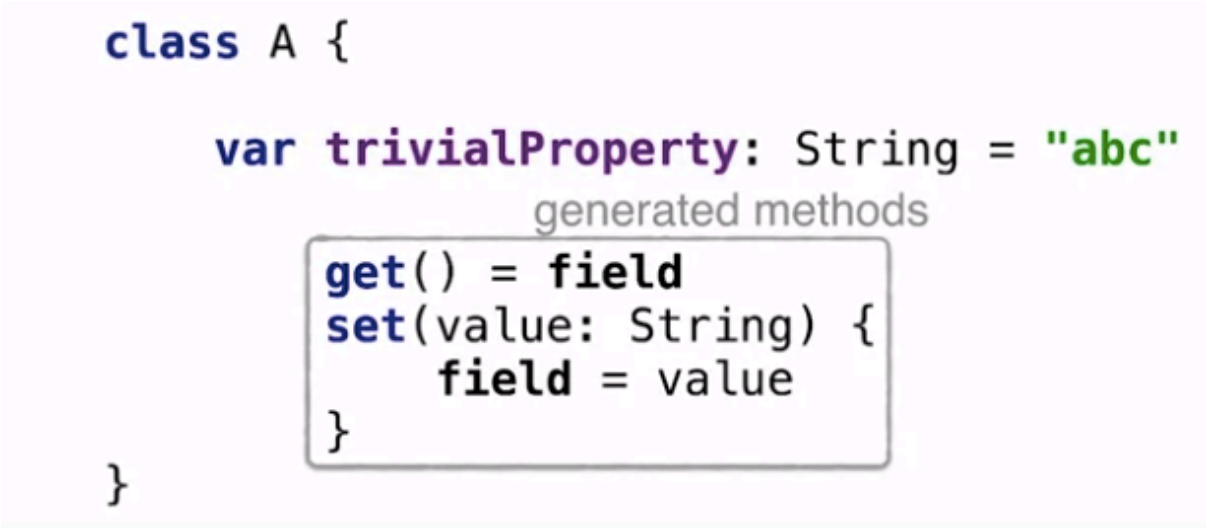
Fields

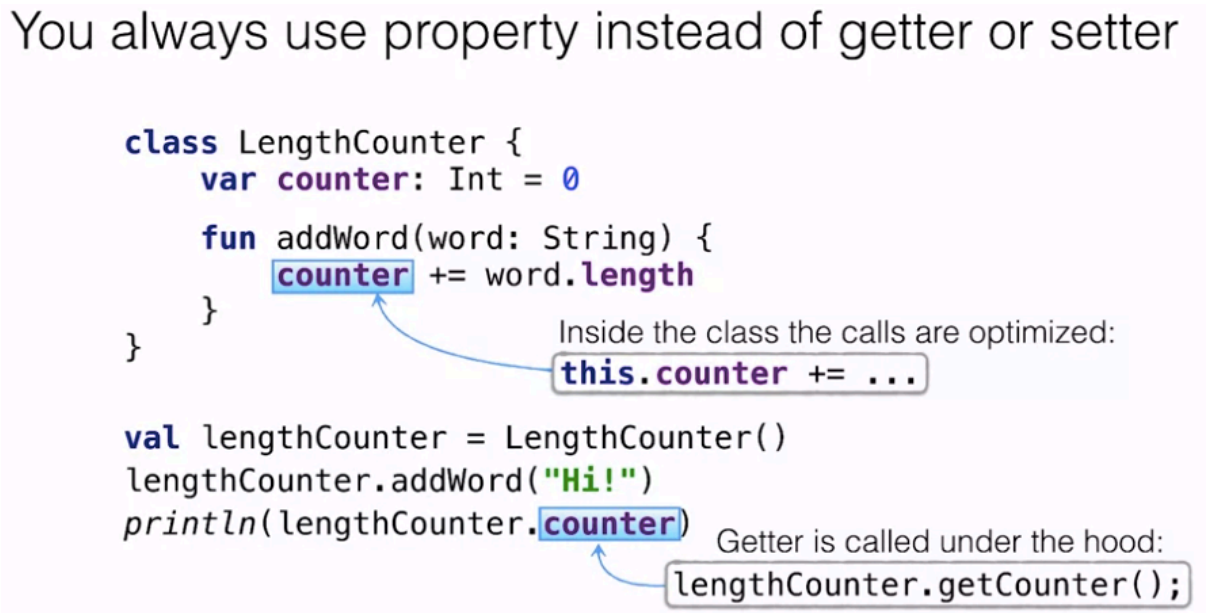
In Kotlin, you don't work with fields directly, you work with properties. However, if you need, **you can access a field inside its property accessors**.

```
class StateLogger {
    var state = false
        set(value) {
            println("state has changed: " +
                    "$field -> $value")
            field = value
        }
}


StateLogger().state = true
                state has changed: false -> true
```

Default accessors for mutable properties

If you **don't define accessors for a property**, the compiler **generates a trivial getter that returns the field value, and a trivial setter updating the value, if the property is mutable**. You don't access field's getters or setters directly, you only use properties both inside and outside of the class.





Changing Visibility of a setter

Sometimes you want a var **mutable property to be accessible only as a read-only property outside of the class**. For that, you can make a **setter private**. Then the getter is accessible everywhere. And therefore the property is accessible everywhere. But it's allowed to modify it only inside the same class

```
class LengthCounter {
    var counter: Int = 0
        private set

    fun addWord(word: String) {
        counter += word.length
    }
}
```

More about Properties

Property in interface

You can define a property in an interface. Why not? Under the hood, it's just a getter. Then you can redefine this getter in subclasses in the way you want.

```
interface User {
    val nickname: String
}

class FacebookUser(val accountId: Int) : User {
    override val nickname = getFacebookName(accountId)
}

class SubscribingUser(val email: String) : User {
    override val nickname: String
        get() = email.substringBefore('@')
}
```

Open property can’t be used in smart cast

Note that **all the properties in interfaces are open** and **can't be used in smart casts**.
Open means that they **can be written in subclasses, they are not final**.
The compiler shows you an arrow. It's impossible because this property has an open or custom getter

If a property has a custom getter, it's not safe to use it in a smart cast because the custom getter can return your new value on each access.

```
interface Session {
    val user: User
}


fun analyzeUserSession(session: Session) {
    if (session.user is FacebookUser) {
        println(session.user.accountId)
    }
}
```

Compiler error: Smart cast to 'FacebookUser' is impossible, because 'session.user' is a property that has open or custom getter

What you can do here, you can **introduce a local variable**, then this much cost applies. Alternatively, you can use other language mechanisms which we'll discuss later. Note also that smart casts don't also work for mutable properties, because the mutable property might be changed in a different thread after you have checked it for being of a specific type. To fix this, you again introduce a local variable.

```
interface Session {
    val user: User
}

fun analyzeUserSession(session: Session) {
    val user = session.user
    if (user is FacebookUser) {
        println(user.accountId)
    }
}
```

✓

Extension properties

In Kotlin, you can define extension properties. The syntax is very similar to the one of defining extension functions. You simply define a property, but specify their stereotypes cursed. You can access the receiver as this reference inside accessories. Or you can emit this reference and call members as in the second example

```kotlin
val String.lastIndex: Int
    get() = this.length - 1

val String.indices: IntRange
    get() = 0..lastIndex


"abc".lastIndex  // 2
"abc".indices    // 0..2
```

Mutable Extension properties

Here, we define the mutable extension property lastChar on StringBuilder. It allow us to get or update the last character. You then can use it as a regular property with the concise syntax to access setChar that just assigns a value to a property

```kotlin
var StringBuilder.lastChar: Char
    get() = get(length - 1)
    set(value: Char) {
        this.setCharAt(length - 1, value)
    }


val sb = StringBuilder("Kotlin?")
sb.lastChar = '!'
println(sb)            // Kotlin!
```

Lazy or late initialization of properties

In this video, you'll learn how to define a Lazy Property. Also how to define the property that should be initialized not in the constructor, but sometime later.

Lazy

Lazy Property is a property which values are computed only on the first success. It's lazy in a sense that it won't do anything unless the result is really needed.
Lazy is **a function that takes lambda as an argument**. Inside this lambda, you provide a way to compute a value that should be stored in this property.
When you access this property, you see that its value is computed only ones, when we access it for the first time. After that, the stored value is returned.

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

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

```
computed!
Hello
Hello
```

The lazy property is lazy in a sense that it won't do anything unless it's needed. In this case, the result is not needed and that's why it's not computed at all

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main(args: Array<String>) {
    // no lazyValue usage
}
```

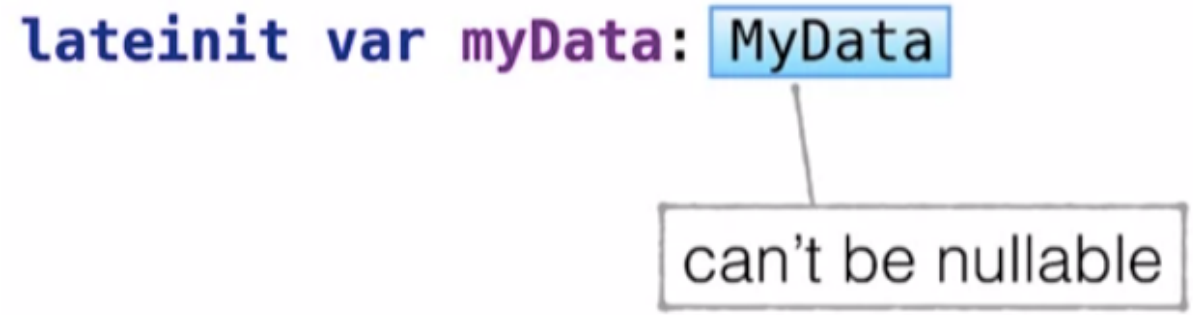
Lateinit

Sometimes, we want to initialize the property **not in the constructor, but in a specially designated for that purpose method**. Here, we initialized the myData property in onCreate method, but not in the constructor

Lateinit constraints

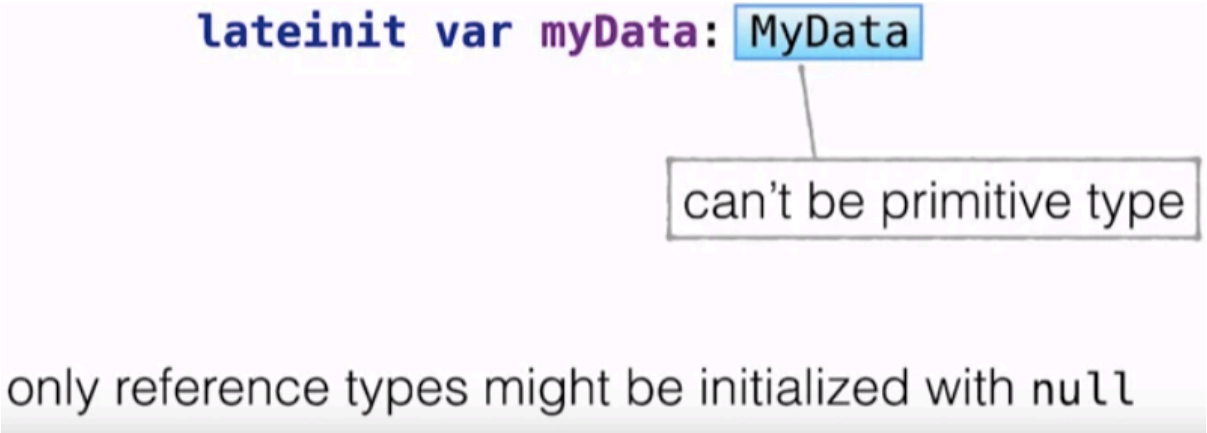
There are some constraints that apply to lateinit properties. For instance, it **can't be val**.

Another constraint obvious is that, **the type of this property is non-nullable**. Thus a purpose of lateinit to make the type of the variable non-nullable, so that we won't have to cope with nullability issues.



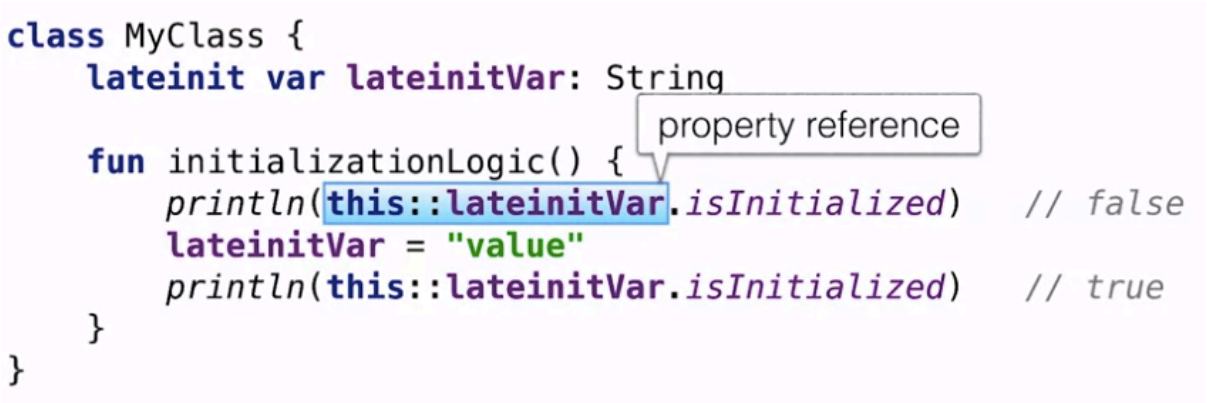
Another constraint, is that lateinit property **can't have a primitive type**. That's also connected with the underlying implementation. **Only reference types might be initialized**

with now under the hood. If you know which our value, the primitive can be initialized with, you can do it by hand. Use zero for an integer for instance. Thus the lateinit modifier can be **only applied to non-nullable reference types**.



Checking lateinit initialization

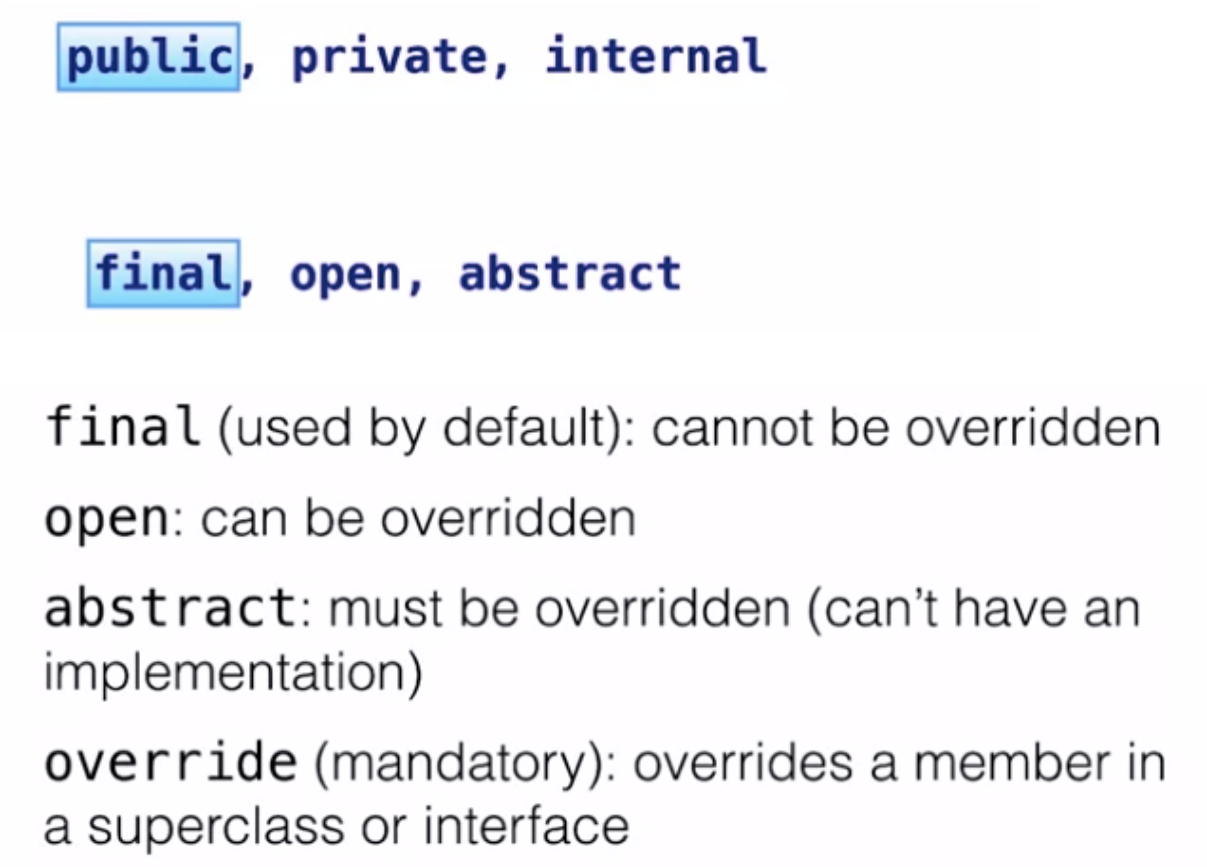
It's possible to check whether lateinit property was or wasn't initialized. To do that, you call **is initialized** on the property reference



OOP in Kotlin

Defaults are diferent

The defaults in Kotlin are different. Any declaration i**s public and final by default**. If you want to **make it non-final, you explicitly need to mark it as open**

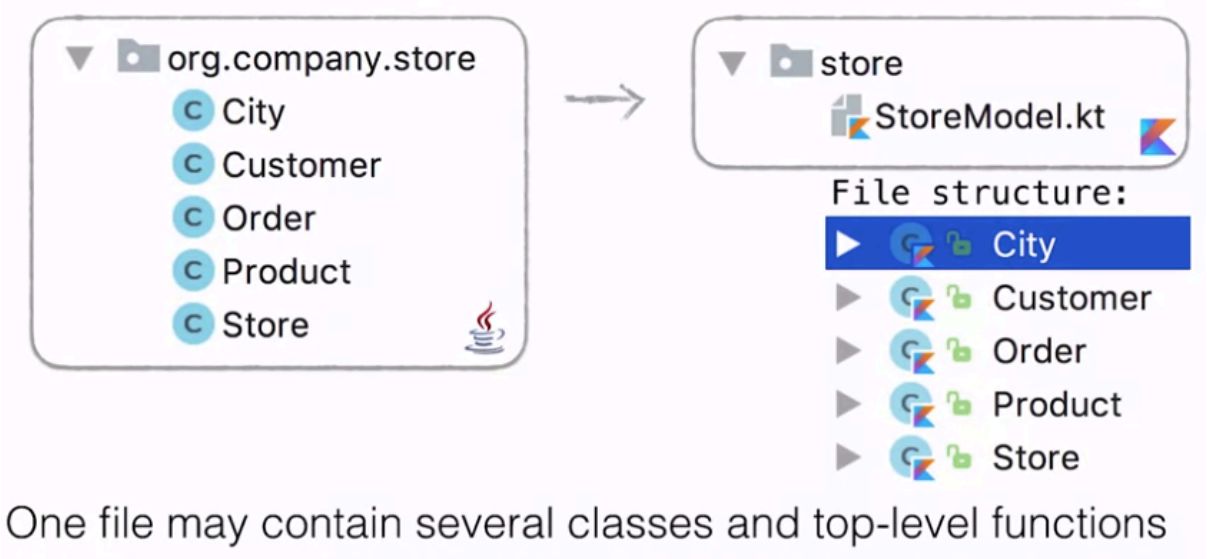


Modifier	Class member	Top-level declaration
public	visible everywhere	
internal	visible in the module	
protected	visible in subclasses	———
private	visible in the class	visible in the file

Package Structure

There is also difference with package structure. In Java, every class should be located in its own separate file even if the class is very small. In Kotlin, it's no longer the case, **you can put several classes into one file**, is especially useful if your classes are data classes that is small and simple. You can combine several classes connected to each other in one file.

You can also **put top-level declarations functions and properties and the class into one file**



Constructors, Inheritance syntax

A constructor is a special function used to initialize a newly created object. We **don't use new keyword in Kotlin**. You **call a constructor as a regular function.**
If you define no constructors in the class, the Kotlin compiler generates one without parameters by default

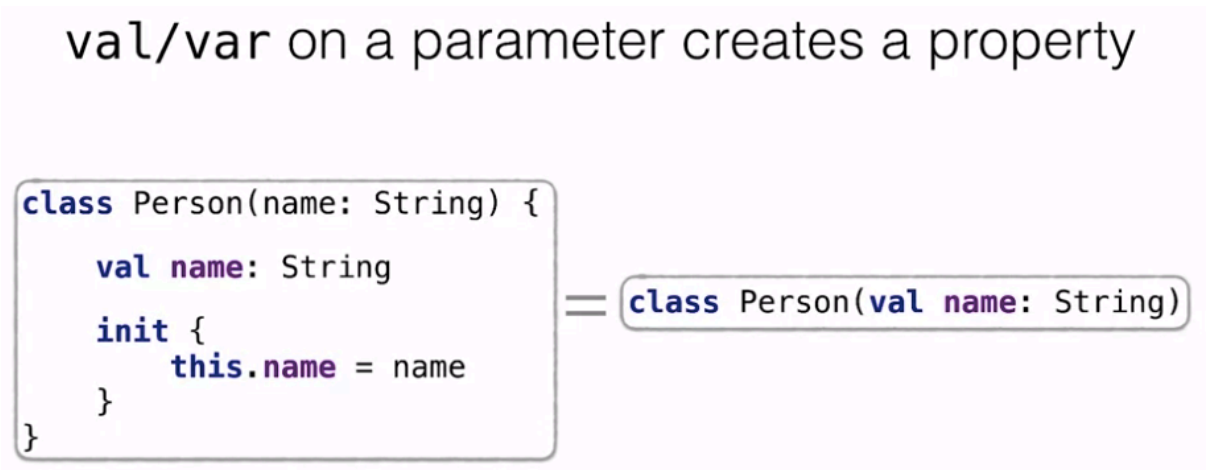
Primary Constructor

The main constructor is called the **primary constructor**. It has a very concise syntax when it's trivial. It **only initializes the properties with his past values**. Here we have two properties which I initialized with the values from the construction.

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

You can put the complicated initialization logic inside the **init** section, which represents the **constructor body**. By default, you define properties inside of the class body between the curly braces. Here you can see the full syntax when you initialize the property with the corresponding constructor parameter in the constructor body.

When you **put val or var before the parameter, that automatically creates a property**. **Without val or var, it's on the constructor parameter**.



Constructor Visibility

You can change the visibility of the constructor if needed. In this case, you need to explicitly use the **constructo**r keyword and put it between the class name and the parenthesis. You can add **private** or **internal** visibility whatever you need.

```
class InternalComponent
internal constructor(name: String) {
    …
}
```

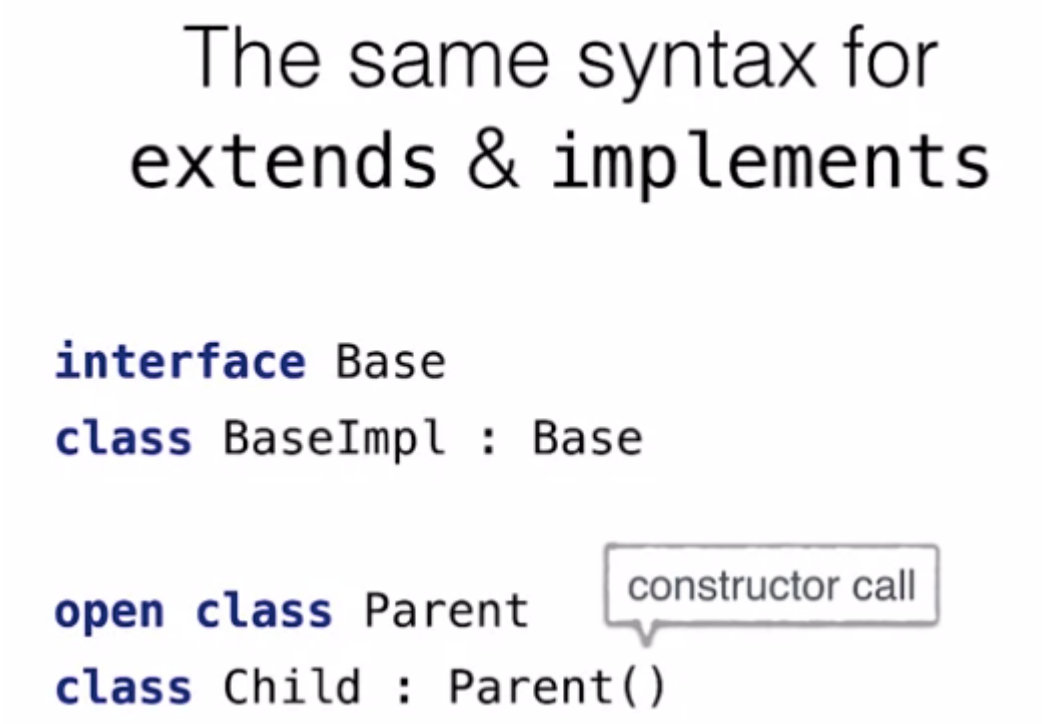
Secondary Constructor

You can define secondary constructors. If default arguments don't do a job for you and you need a different argument list, you can do that. Each secondary constructor **must call another secondary or primary constructor**.

```
class Rectangle(val height: Int, val width: Int) {

    constructor(side: Int) : this(side, side) { … }
}
```

`this(...)` calls another constructor of the same class

Inheritance (different sintax)

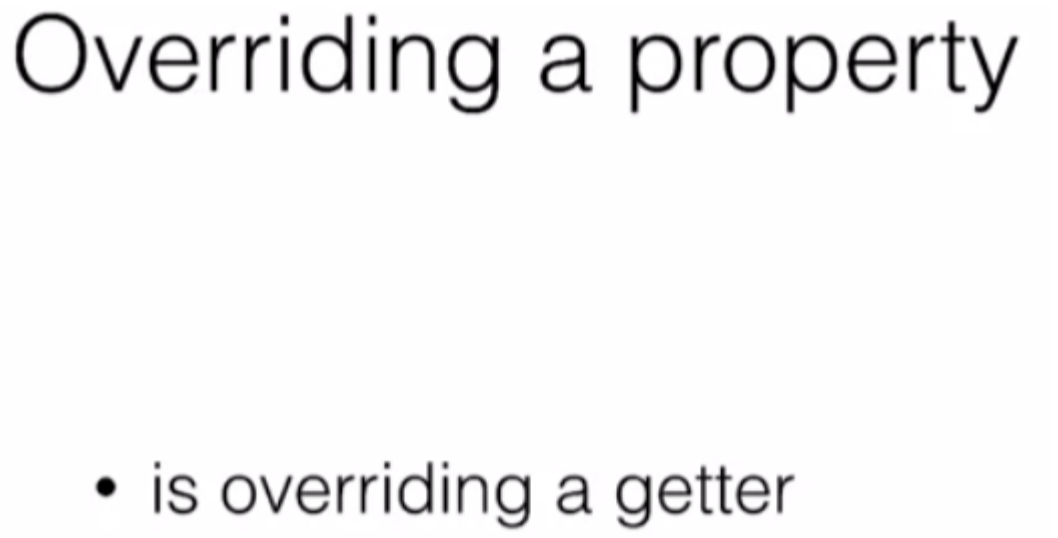


Calling the constructor of the parent class

```
open class Parent(val name: String)
class Child(name: String) : Parent(name)


open class Parent(val name: String)
class Child : Parent {
    constructor(name: String, param: Int) : super(name)
}
```

Overriding a property

When you override a property, in fact, you override a getter not a field. For instance, you can override a property that doesn't have a field with a property that has a field.



Class modifiers - I

Enums

If you need a class with a fixed number of values, you can define the these values as **enum** constants. The difference with Java is that now enum is not a separate instance, but a modifier before the class keyword.

Importing enum constants

```
package mypackage

import mypackage.Color.*

enum class Color {
    BLUE, ORANGE, RED
}

fun getDescription(color: Color) =
    when (color) {
        BLUE -> "cold"
        ORANGE -> "mild"
        RED -> "hot"
    }
```

Enum class with properties

```
enum class Color(
        val r: Int, val g: Int, val b: Int
) {
    BLUE(0, 0, 255), ORANGE(255, 165, 0), RED(255, 0, 0);

    fun rgb() = (r * 256 + g) * 256 + b
}


println(BLUE.r)         // 0
println(BLUE.rgb())     // 255
```

Data Class

data modifier

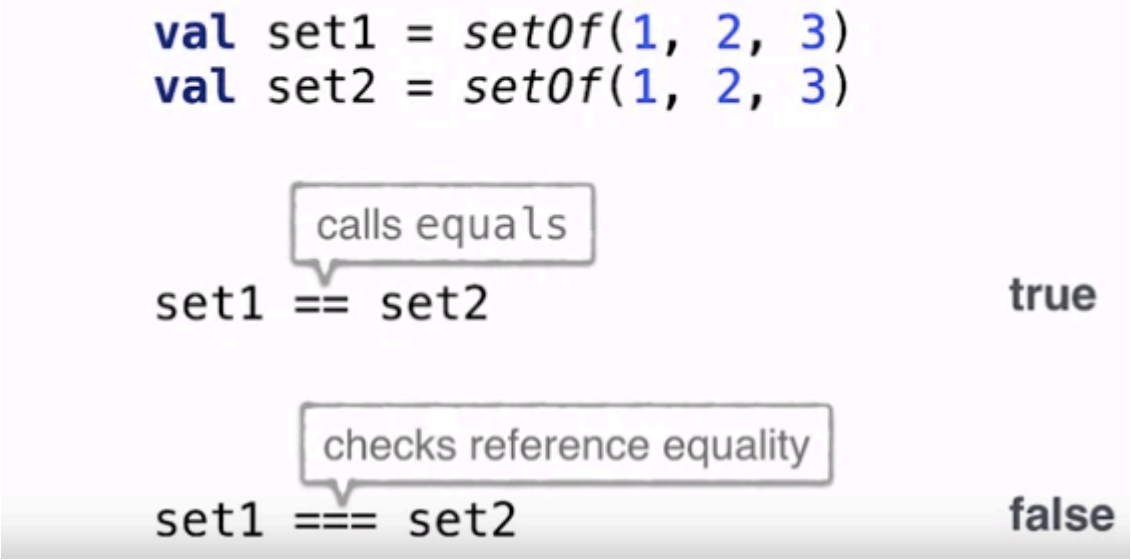
Generates useful methods:
`equals`, `hashCode`, `copy`, `toString`,
and some others

Copying the instance

```
data class Contact(val name: String, val address: String)

contact.copy(address = "new address")
```

Equals & reference equality

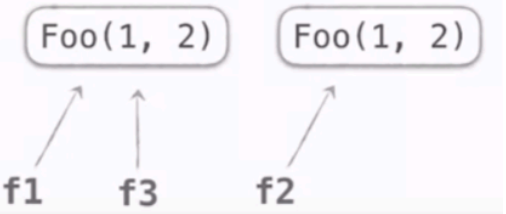
By default, when you use the **double equals sign**, it **goes equals under the hood**. If your class redefines equals, it will compare the elements in a meaningful way. If you still need **reference equality** for some reason, you can **use the triple equal sign**



By default, like in Java, in Kotlin, every class **inherits default equals implementation,** which just **checks reference equality**. When we compare reference of the type foo, we compare them by reference equality. Despite equals is called, by default, this equals is the

trivial one. That is why these lines brings false. If we compare the reference that refer to the same object in memory, we'll get true obviously.

```
class Foo(val first: Int, val second: Int)

val f1 = Foo(1, 2)
val f2 = Foo(1, 2)
println(f1 == f2) // false

val f3 = f1
println(f1 == f3) // true
```



For **data classes**, the right **equals** and **hashCode** methods **are generated**. We now **compare the elements by their content**

```
data class Bar(val first: Int, val second: Int) {

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Bar) return false
        return (first == other.first
                && second == other.second)
    }

    override fun hashCode(): Int =
            first * 31 + second
}
```

generated methods

Properties in primary constructor

Note that the compiler **only uses the properties defined inside the primary constructor** for the **automatically generated functions** like **to-string**, **equals** and **hashCode**. To exclude a property from the generated implementations, declare it inside the class body like the nickname property in this example.
Here, the **first user equals this second one** despite they have different values for the nickname property because on the **email values are compared by default**

```
data class User(val email: String) {
    var nickname: String? = null
}

val user1 = User("voldemort@gmail.com")
user1.nickname = "Voldemort"
println(user1)  // User(email=voldemort@gmail.com)


val user2 = User("voldemort@gmail.com")
user2.nickname = "YouKnowWho"
println(user1 == user2)  // true
```

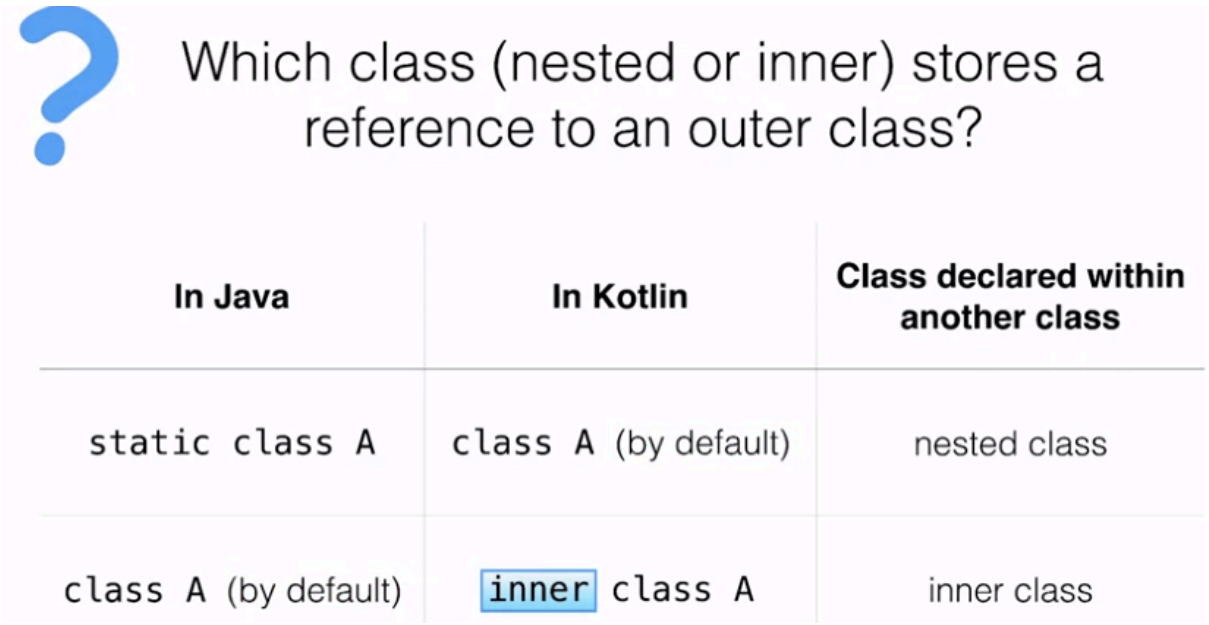
Class modifiers - II

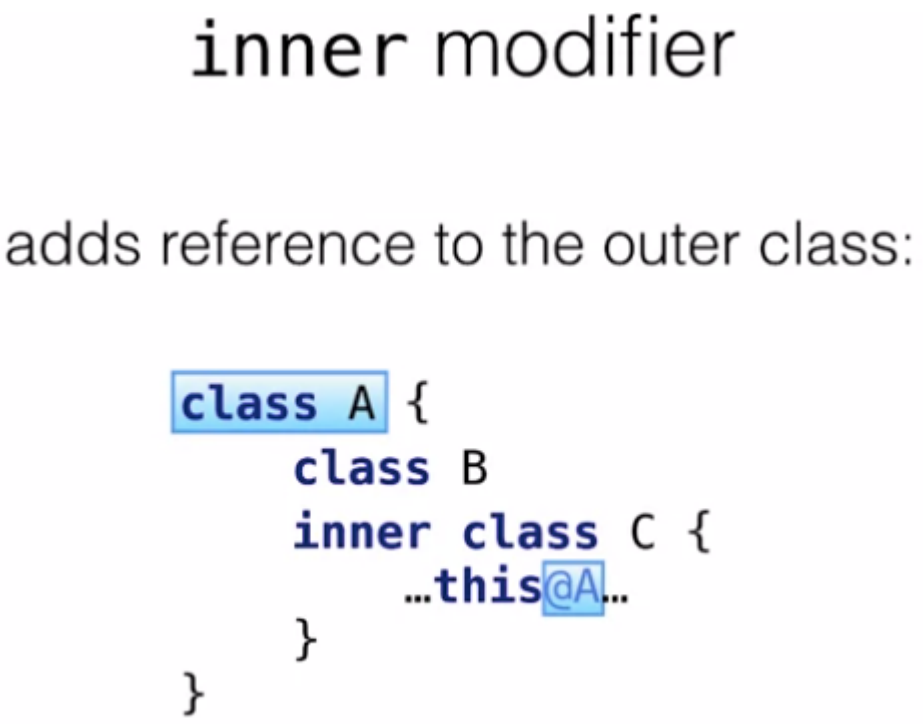
Sealed modifier

Restricts class hierarchy:
all subclasses must be located in the same file

```
sealed class Expr
class Num(val value: Int) : Expr()
class Sum(val left: Expr, val right: Expr) : Expr()

fun eval(e: Expr): Int = when (e) {
    is Num -> e.value
    is Sum -> eval(e.left) + eval(e.right)
}
```

(Nested or Inner) Class





Class Delegation

Se tiene una clase que implementa dos interfaces:

```kotlin
interface Repository {
    fun getById(id: Int): Customer
    fun getAll(): List<Customer>
}

interface Logger {
    fun logAll()
}

class Controller(
        val repository: Repository,
        val logger: Logger
) : Repository, Logger {
   ...
}
```

El siguiente código se puede genera manualmente ó generar los métodos delegados

```kotlin
class Controller(
        val repository: Repository,
        val logger: Logger
) : Repository, Logger {

    override fun getById(id: Int) = repository.getById(id)

    override fun getAll(): List<Customer> = repository.getAll()

    override fun logAll() = logger.logAll()
}
```

Se puede generar los metodos implementando las interfaces por delegacion a estas instancias. **By** significa por delegacion a la siguiente instancia.

```kotlin
class Controller(
        repository: Repository,
        logger: Logger
) : Repository by repository, Logger by logger
```

implements an interface by delegating to …

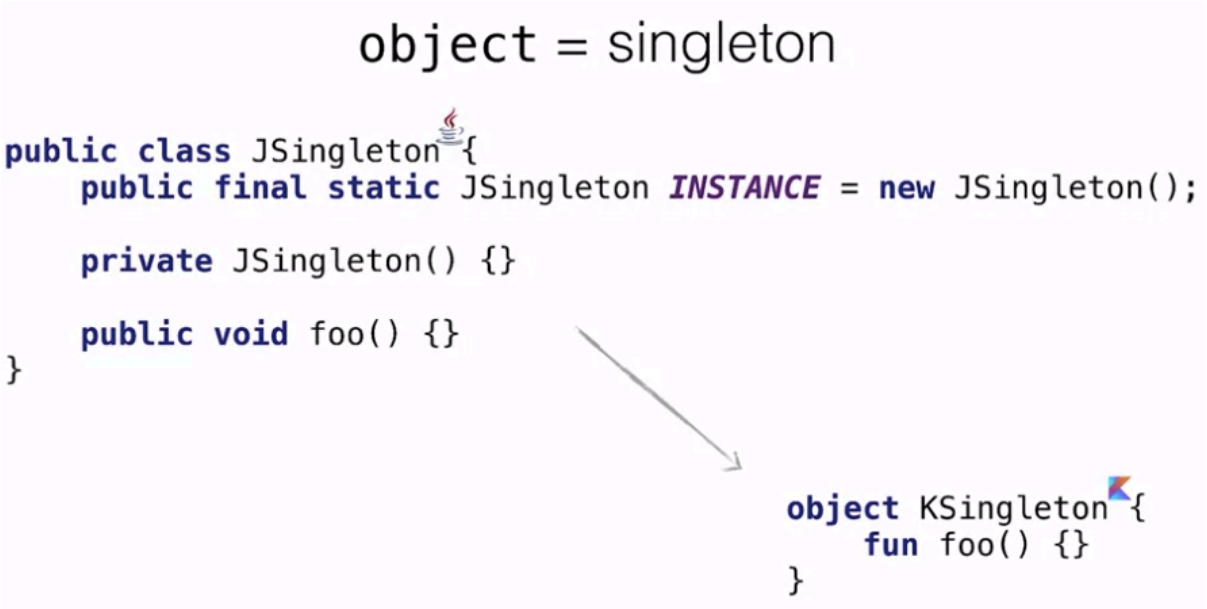
Class delegation

```
interface Repository {
    fun getById(id: Int): Customer
    fun getAll(): List<Customer>
}

interface Logger {
    fun logAll()
}

class Controller(
        repository: Repository,
        logger: Logger
) : Repository by repository, Logger by logger

fun use(controller: Controller) {
    controller.logAll()
}
```

Objects, object expressions & companion objects

Object Declaration = Singleton

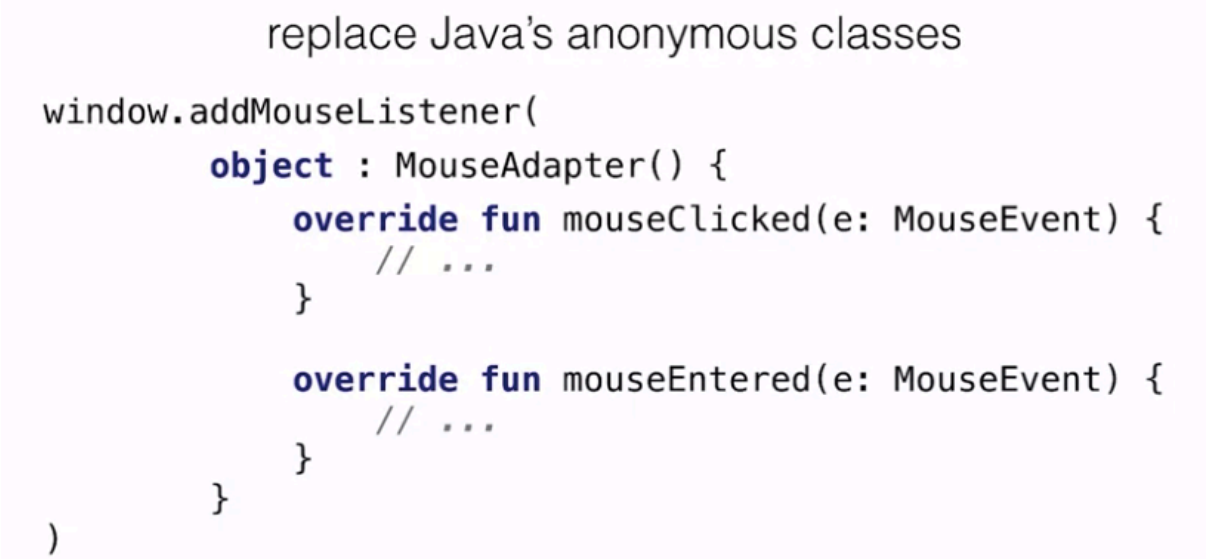
```
object KSingleton {
    fun foo() {}
}


KSingleton.foo()
```



Object Expression

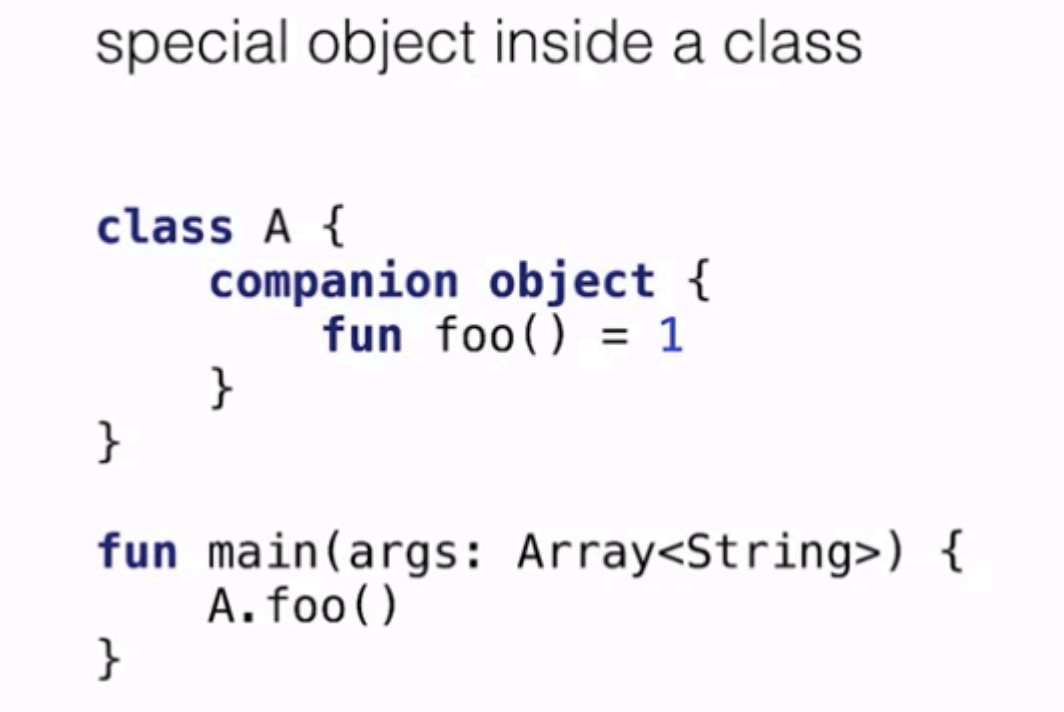
Imagine you need to implement an interface, override a couple of methods in a class. That's going to be the only usage of this new class, so you don't want to create a named class for that.In Java, you create an anonymous class in this situation. In Kotlin, you use object expressions.

Note that whenever you interface, it has **only one single abstract method**, then you can **use lambda**. There is no need for object expression. But if you need to implement **several methods**, then you will use **object expressions**.



Companion objects

It's a nested object inside a class but a special one. The one which **members can be accessed by the class name**.
In Kotlin, there are **no static methods** like in Java, and **companion objects might be a replacement for tha**t.



At first, companion object **can implement an interface**. Sometimes, we lack this feature in Java. It would be nice if a static method could **override a member of an interface**. But for static, that's not possible. Now, it's **possible for companion object.**
Here, a companion object implements factory. You can use the companion object instance accessed simply by the class name as an instance of a factory

can implement an interface

```
class A {
    private constructor()

    companion object : Factory<A> {
        override fun create() = A()
    }
}

fun <T> createNewInstance(factory: Factory<T>)

createNewInstance(A)

A.create()
```

Another thing which you can do with companion object, is you can define extension straight. To distinguish an extension to a class from an extension to a companion object, you use the companion suffix.After that, you can call such extension function like a companion object member simply by the class name.

companion object can be a receiver of extension function

```
// business logic module
class Person(val firstName: String, val lastName: String) {
    companion object { ... }
}

// client/server communication module
fun Person.Companion.fromJSON(json: String): Person {
    ...
}

val p = Person.fromJSON(json)
```

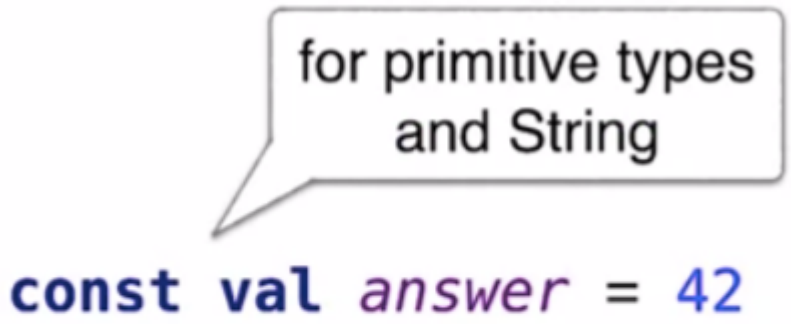
Constants

If you **define a constant of a primitive type of string** in Kotlin, you can declare it using a **const** modifier, that will make it a compile-time constant. For **reference types**, if for some reason you **don't want to generate getters** under the hood, you apply the **@JvmField** annotation, that will instruct the compiler to **generate only the field**.

- `const` (for primitive types and String)
- `@JvmField` (eliminates accessors)

Compile time constant

The Kotlin compiler also **inlines** the value of such constant. It replaces its name with its value everywhere in the code. Note that it works **only for primitive types and strings**.

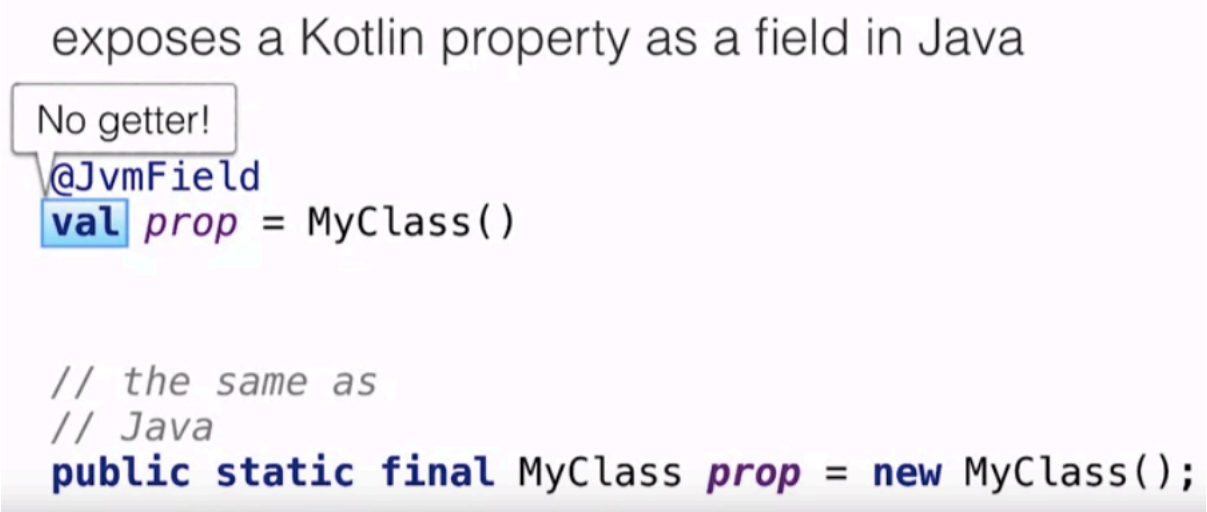


the value is inlined

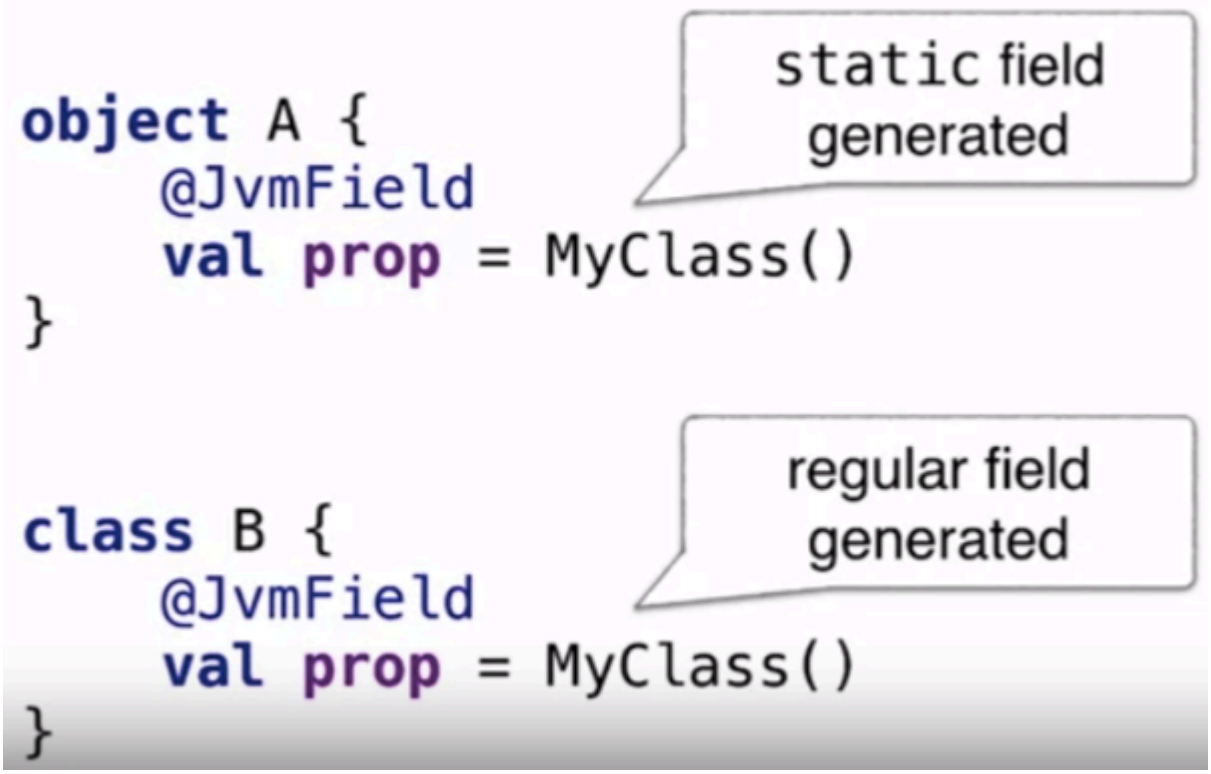
@JvmField

@JvmField **exposes a property as a field**, it makes it public if a property is public. Then this field can be accessed from Java. It's useful for some frameworks that need public fields

available. After you apply @JvmField annotation, **no getter is generated for read-only property and no setter if the property is mutable**. Here, prop is a top level property. So applying @JvmField is the same as defining static field directly.



@JvmField makes a **property static** if used at the **top level or inside object**. If used inside the regular class, the regular field will be generated.



@JvmStatic

When you simply define a **property inside an object**, then it becomes **available from Java only via a getter**. Because the field, **by default, is private**. As the getter is only available as

a member of instance field. To **make it available as a static member**, you can **apply @JvmStatic annotation**. In this case, however, you can **only access the getter as a static member not the field**. That means it doesn't make sense to apply @JvmStatic properties.

```
object SuperComputer {
    @JvmStatic
    val answer = 42
}
```

```
// Java
SuperComputer.getAnswer()
```

To expose the field, apply the **@JvmField** annotation.

```
object SuperComputer {
    @JvmField
    val answer = 42
}
```

```
// Java
SuperComputer.answer
```

In the **case of primitive types and strings**, you can use **const** modifier.

```
object SuperComputer {
    const val answer = 42
}
```

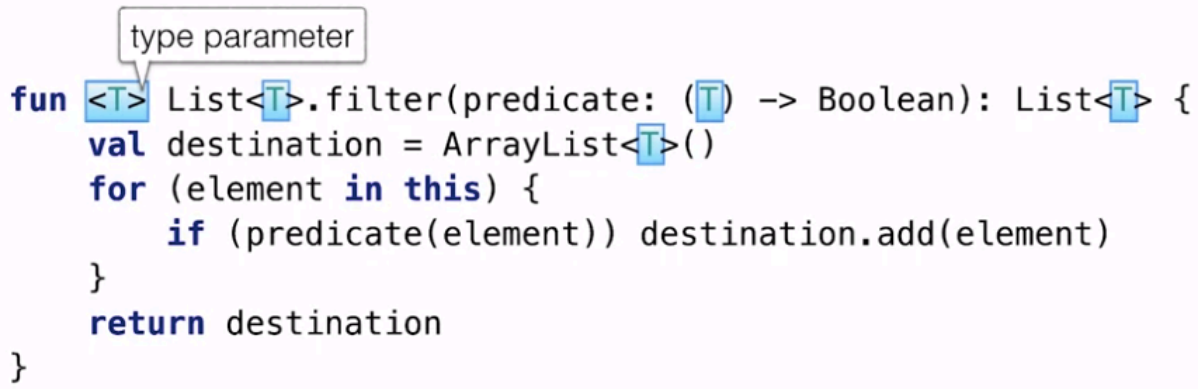
```
// Java
SuperComputer.answer
```

Generics

Generic interfaces and classes

```
interface List<E> {
    fun get(index: Int): E
}
```

Generic Functions



Nullable Generic Argument

```
fun <T> List<T>.firstOrNull(): T?
```

Non-nullable upper bound

If you want to restrict the generic argument so that it was not nullable, you can specify a **non-null upper bound**. You add an upper bound right after the type parameter declaration using the same column which replaces the extents cube root in Kotlin

```
fun <T : Any> foo(list: List<T>) {
    for (element in list) {

    }
}
```

```
foo(listOf(1, null))
```

Error: Type parameter bound for T is not satisfied: inferred type Int? is not a subtype of Any

Type parameter constraints

You can specify a different upper bound. In this case, you can pass both integer and double numbers, since they both extent number.

```
fun <T : Number> oneHalf(value: T): Double {
    return value.toDouble() / 2.0
}
```

Comparable upper bound

```
fun <T : Comparable<T>> max(first: T, second: T): T {
    return if (first > second) first else second
}


max(1, 3)  // 3
```

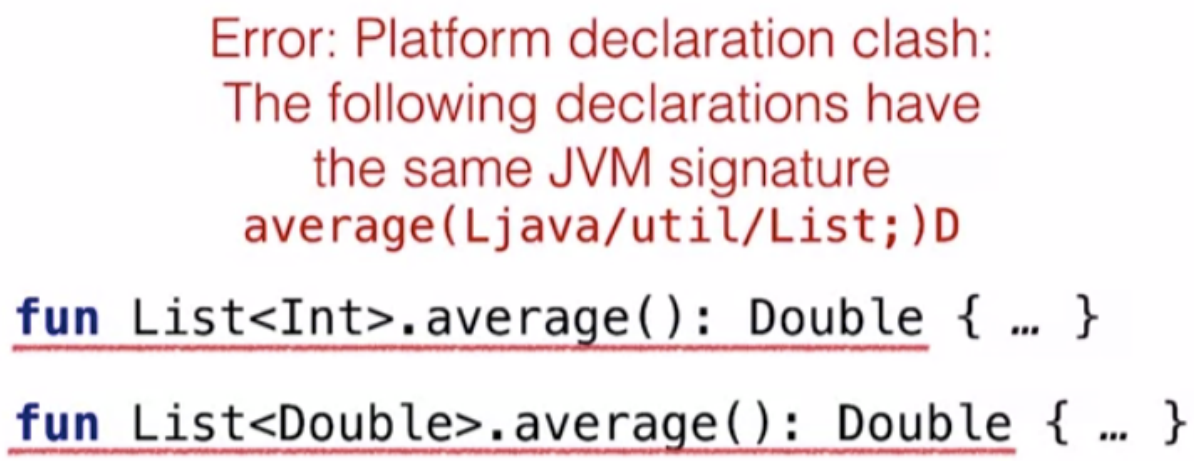
Multiple Constraints

To define multiple upper bound constraints prototype parameter, use the **where** syntax edge of the function declaration. Here, you can pass any type that extends to different interfaces, char sequence and appendable. Stringbuilder implements both terrace sequence and dependable, so it's a valid argument for these function.

```
fun <T> ensureTrailingPeriod(seq: T)
        where T : CharSequence, T : Appendable {
    if (!seq.endsWith('.')) {
        seq.append('.')
    }
}


val helloWorld = StringBuilder("Hello, World")
ensureTrailingPeriod(helloWorld)
println(helloWorld)  // Hello, World.
```

Same JVM Signature

Because of JVM platform constraints, you can't define two functions with the same JVM signature, the signature with erased generic type parameters. Here, we tried to define two extension functions on list int entries double. The functions differ only in generic type argument value, int versus double, and that's not enough, the Kotlin compiler doesn't allow that.



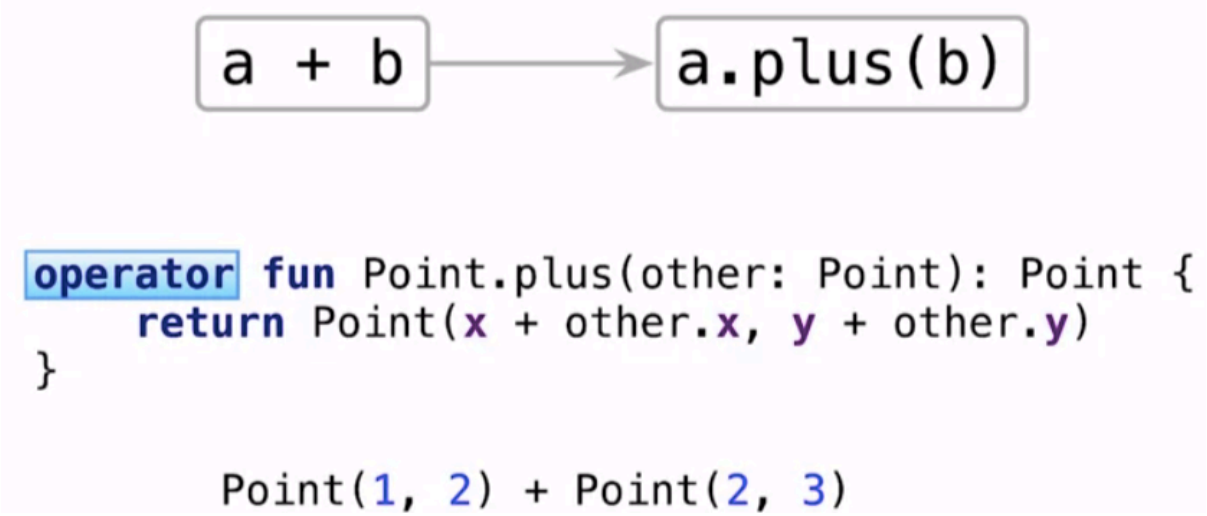
However, applying one annotation can take this example and make such declaration legal. We've already discussed these annotation but apply it only to the whole file before. Check your guess. There's **JvmName**.
When applied before the package declaration, it modifies the filename under the hood, but also **can modify the name of the function** at the bytecode. In this case, two functions now have different names at the bytecode. The first one is average, the second one is average of double
However, from Kotlin, you use both functions simply by the average name. The. Kotlin compiler infers which function you want to call in each case.

```
fun List<Int>.average(): Double { … }

@JvmName("averageOfDouble")
fun List<Double>.average(): Double { … }
```

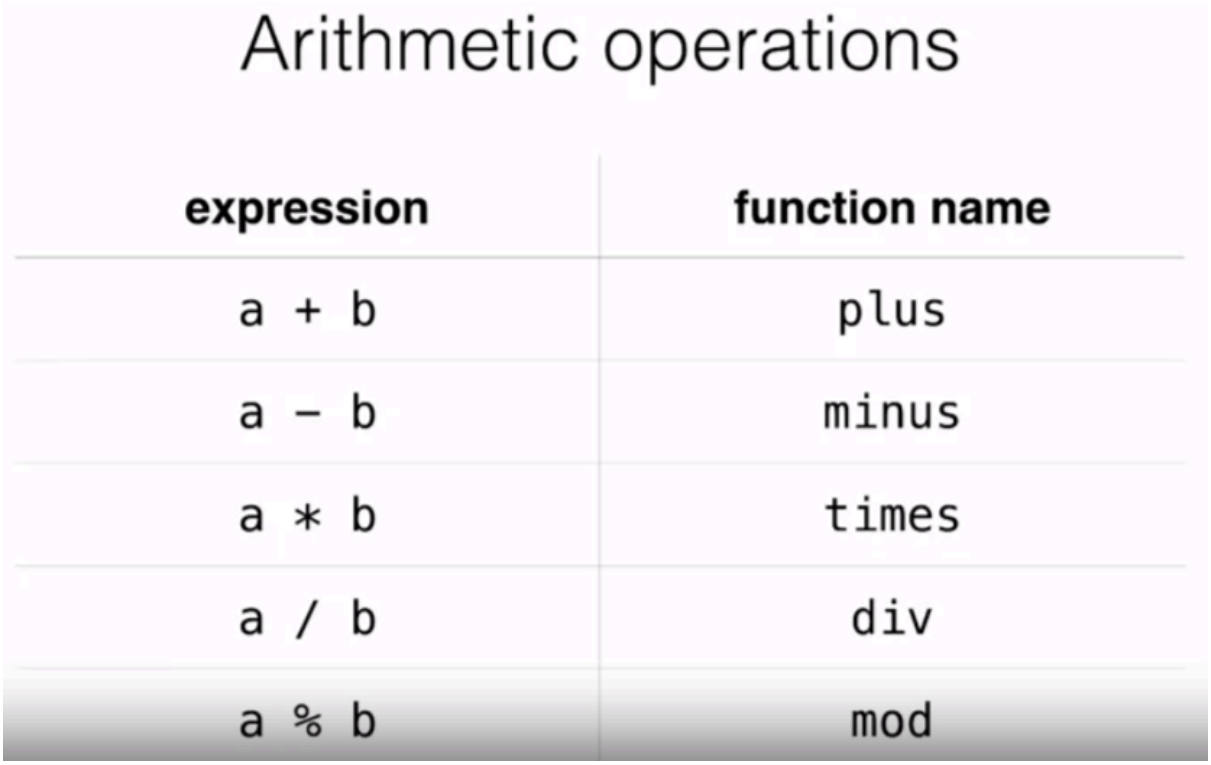
Conventions

Operator Overloading

Arithmetic operations



There is the fixed list of arithmetic operations. It's not that you can use any name



There are no restrictions on parameter type. It doesn't have to be the same as the receiver type. For instance, you can define a function that multiplies the point and an integer number.

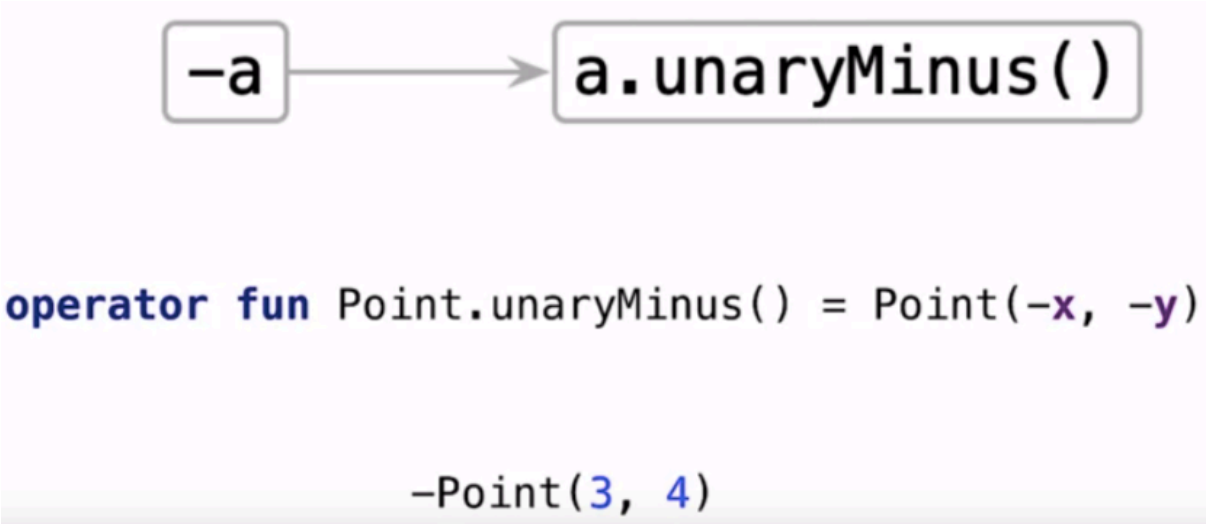
No restrictions on parameter type

```
operator fun Point.times(scale: Int): Point {
    return Point(x * scale, y * scale)
}
```

```
Point(1, 2) * 3
```

Unary Operations

There is a similar list of unary operations. Unary operator is a **function without arguments** which we can call as an operator on this specified receiver




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

```
-Point(3, 4)
```

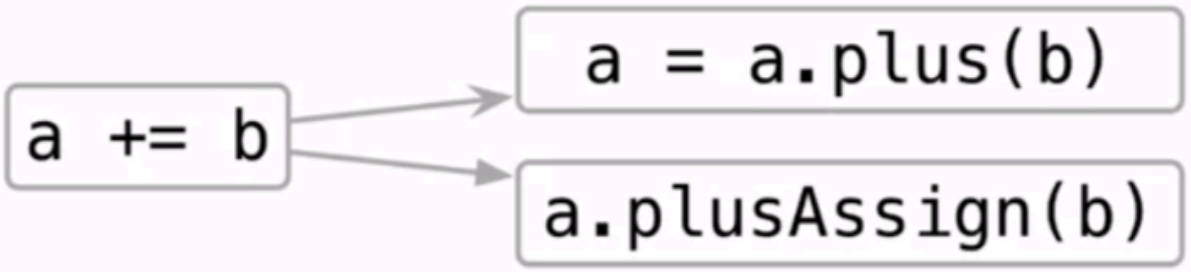
Unary Operators List

Unary operations

expression	function name
+a	unaryPlus
-a	unaryMinus
!a	not
++a, a++	inc
--a, a--	dec

Assignment Operations

There are two possible meanings for their plusAssign operations. At first, it might be resolved to a plus function if the mutable variable is changed. That means you simply modify its value. Another option is the plusAssign function. If it is defined, then this operation might be resolved to plusAssign call under the hood.



Conventions for lists

This convention works for list. If you **use plus on the read-only list**, pay attention that then it will **create a new list and return it as a result**. From mutableList, you can use plusAssign and it calls the corresponding plusAssign function. It's a bit confusing.

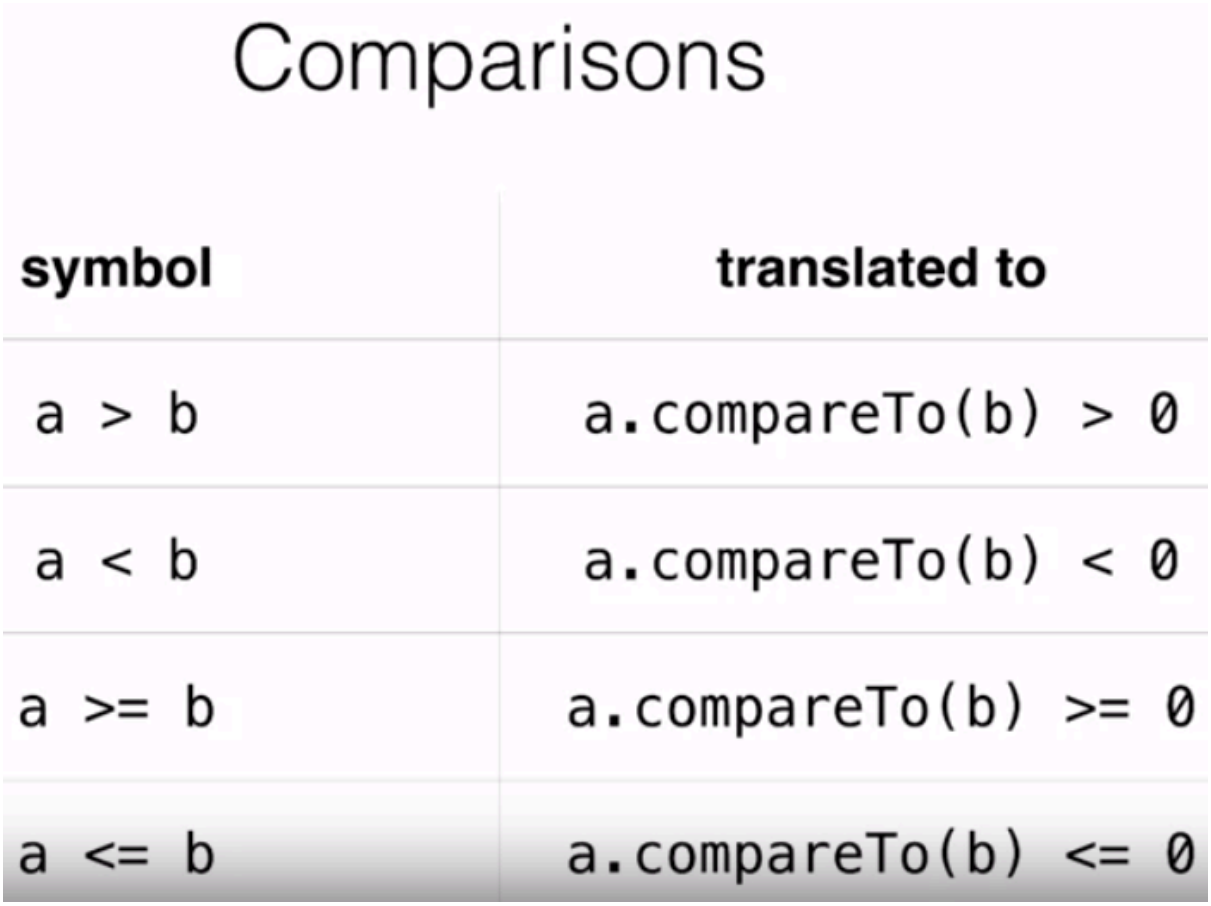
```
val list = listOf(1, 2, 3)
val newList = list + 2

val mutableList = mutableListOf(1, 2, 3)
mutableList += 4
```

Conventions

Comparisons

```
"abc" < "def"

myComparable1 <= myComparable2
```

In Kotlin, you can compare strings using this nice syntax. Not on these chains, but any classes that define comparative method. Under the hood, the comparison operators are all compiled using the comparative method comparisons. Note that if your class **implements comparable** interface, then you can automatically compare its instances using this nice syntax.



Equality check

Equality check, as we've already discussed, is compiled to calling equals under the hood. It not only invokes equals, but also correctly handles nulls. You no longer need to remember to check a variable for being not null before equality check

```
s1 == s2
```

Calls `equals` under the hood:

```
s1.equals(s2)
```

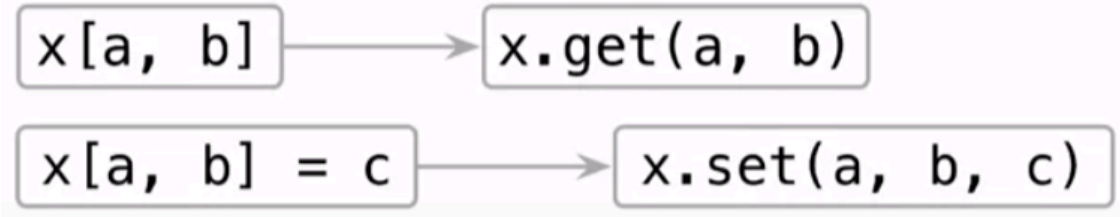
Correctly handles nullable values:

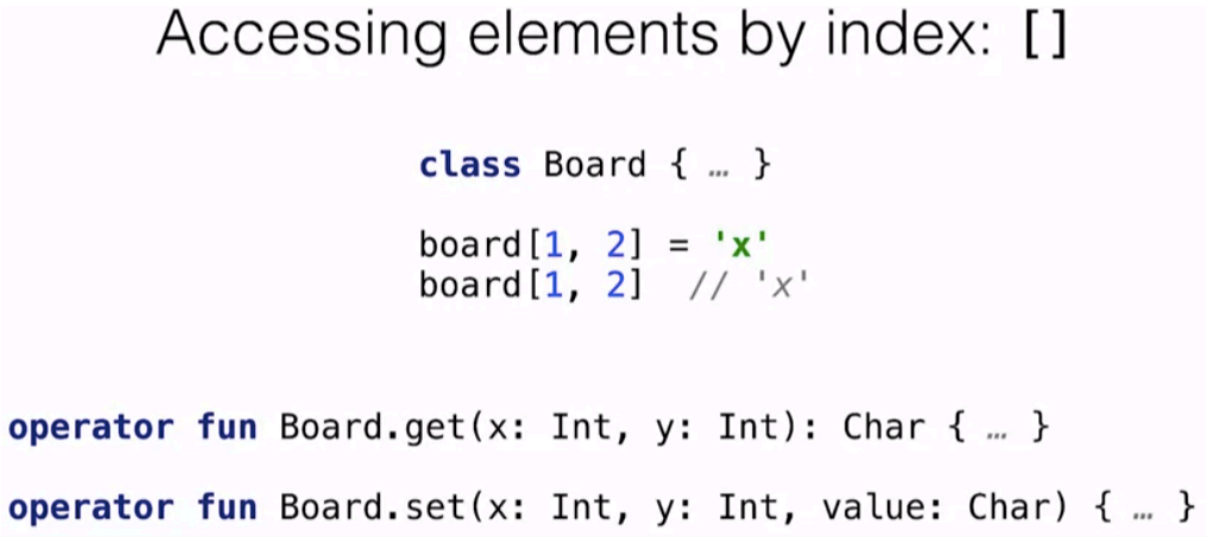
```
s1 == s2          // s1.equals(s2)
null == "abc"     // false
null == null      // true
```

Accessing Element By Index: []

When you access the elements by index, that also works through a convention. Under the hood, the **get** and **set** methods are called. You can define get and set operator functions as members or extensions for your own custom classes. Note that **you can put several arguments inside square brackets**

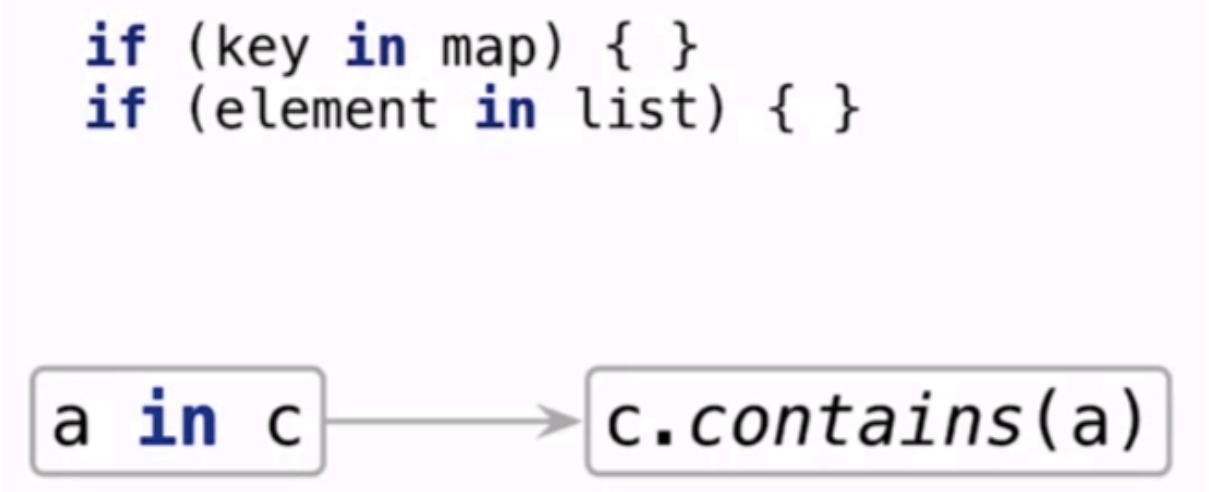
```
map[key]
mutableMap[key] = newValue
```





The In Convention

In also works through a convention. When you check whether an element belongs to a map a list, or range, under the hood **contains** is called.



The rangeTo Convention

Creating a range is another convention. Whenever **you use the double-dot syntax**, you actually call the **range to** operate your function. That means you can support the same syntax for your custom classes

```
if (s in "abc".."def") { }
for (i in 1..2) { }

val oneTo100: IntRange = 1..100
for (i in oneTo100) { }
```

We have to implement a “Class-Range” that extends from “CloseRange<T>, overriding the members “start”, “endInclusive”, and its functions like “contains”, “toString”, etc.. Then override the operator “rangeTo” that must return an instance of Class-Range.

```
class RationalRange(
    override val start: Rational,
    override val endInclusive: Rational
) : ClosedRange<Rational> {

    override fun contains(value: Rational): Boolean {
        val startFraction : Float = start.fraction
        val endFraction : Float = endInclusive.fraction
        val valueFraction : Float = value.fraction

        return valueFraction in startFraction..endFraction
    }

    override fun toString(): String = "$start..$endInclusive"
}
```

```
//Overload Operator ".."
operator fun rangeTo(that: Rational) = RationalRange(this, that)
```

The iterator Convention

For loop iteration also goes through a convention. In Kotlin, you can iterate over a string. I've told you that Kotlin string under the hood is a regular Java length string. But Java length string doesn't implement iterable interface. So in Java, you can't iterate over it. In Kotlin, that's possible because we can define this **iterator operator** as an extension function.

```
operator fun CharSequence.iterator(): CharIterator

for (c in "abc") { }
```

Destructuring declarations

Destructuring declarations allow you to define several variables at once by assigning one expression and automatically destructuring it.Here in both cases, **we initialize two variables using one pair**. The pair components are automatically assigned to the first and to the second variables

```
val (description, colour) = when {
    degrees < 10 -> "cold" to BLUE
    degrees < 25 -> "mild" to ORANGE
    else -> "hot" to RED
}


val (oldest, youngest) = allPossiblePairs
        .maxBy { it.first.age - it.second.age }!!
```

Syntax

The **similar this destructuring works in a for loop, or for lambda** argument. Note that you always put the several variables that you want to initialize after the **destructuring inside parentheses**.

```
val (first, second) = pair
for ((key, value) in map) { }
map.forEach { (key, value) -> }
```

```
val (a, b) = p
```

→

```
val a = p.component1()
val b = p.component2()
```

Under the hood

Under the hood, the assignment syntax is compiled to calling component one, component two, etc functions.

```
val (first, second) = pair
```

```
val first = pair.component1()
val second = pair.component2()
```

```
for ((key, value) in map) {
}
```

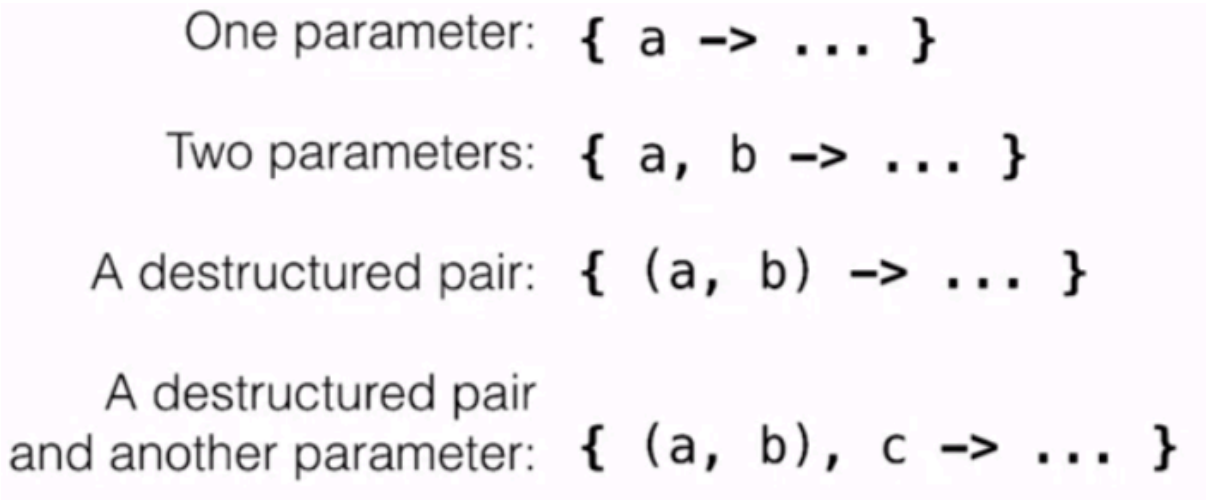
```
for (entry in map) {
    val key = entry.component1()
    val value = entry.component2()
}
```

```
map.forEach { (key, value) ->
}
```

```
map.forEach { argument ->
    val key = argument.component1()
    val value = argument.component2()
```

Destructuring in Lambdas

Note the difference between declaring two parameters and declaring a destruction pair inside of a parameter. You're **always surrounded with a parenthesis** of the variables which are the result of the destructuring.



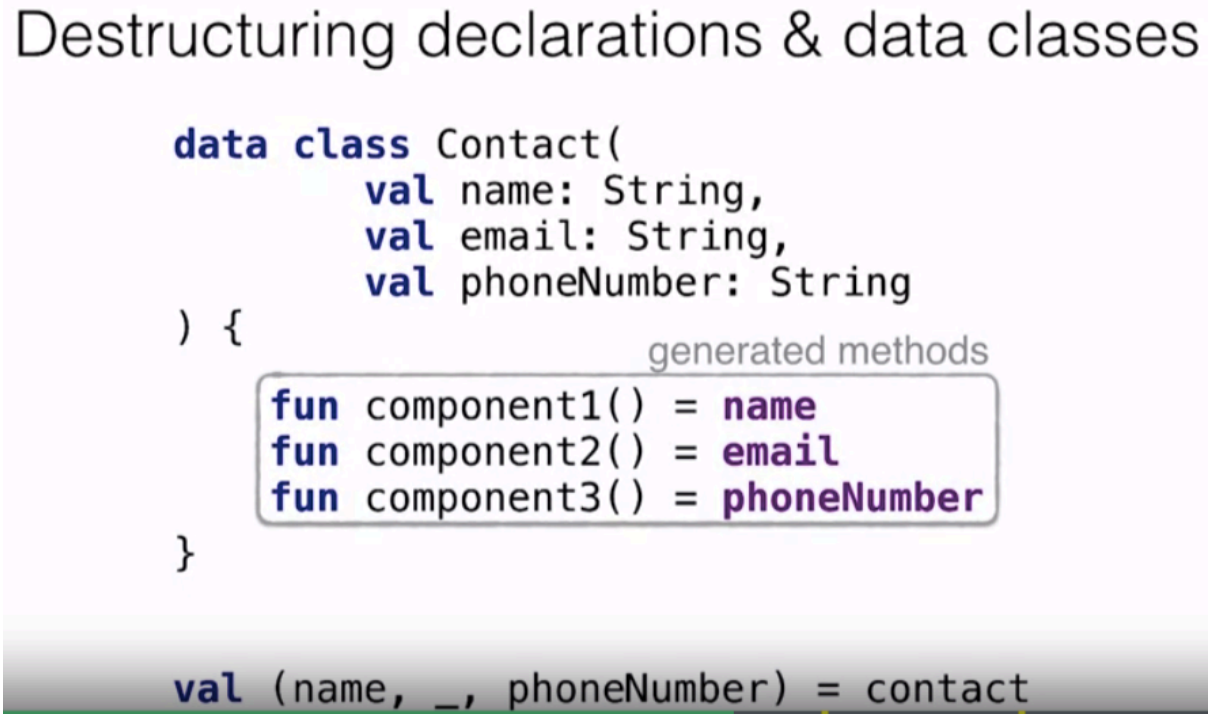
Iterating over list with an index

Note that iterating over list with this index also works using destructuring declarations. With index, extension function returns a list of index to value elements. Index value has component one and component two defined, and therefore can be used with destructuring syntax.

```
for ((index, element) in list.withIndex()) {
    println("$index $element")
}

for (indexedValue in list.withIndex()) {
    val index = indexedValue.component1()
    val element = indexedValue.component2()
    println("$index $element")
}

fun <T> Iterable<T>.withIndex(): List<IndexedValue<T>> { … }
```

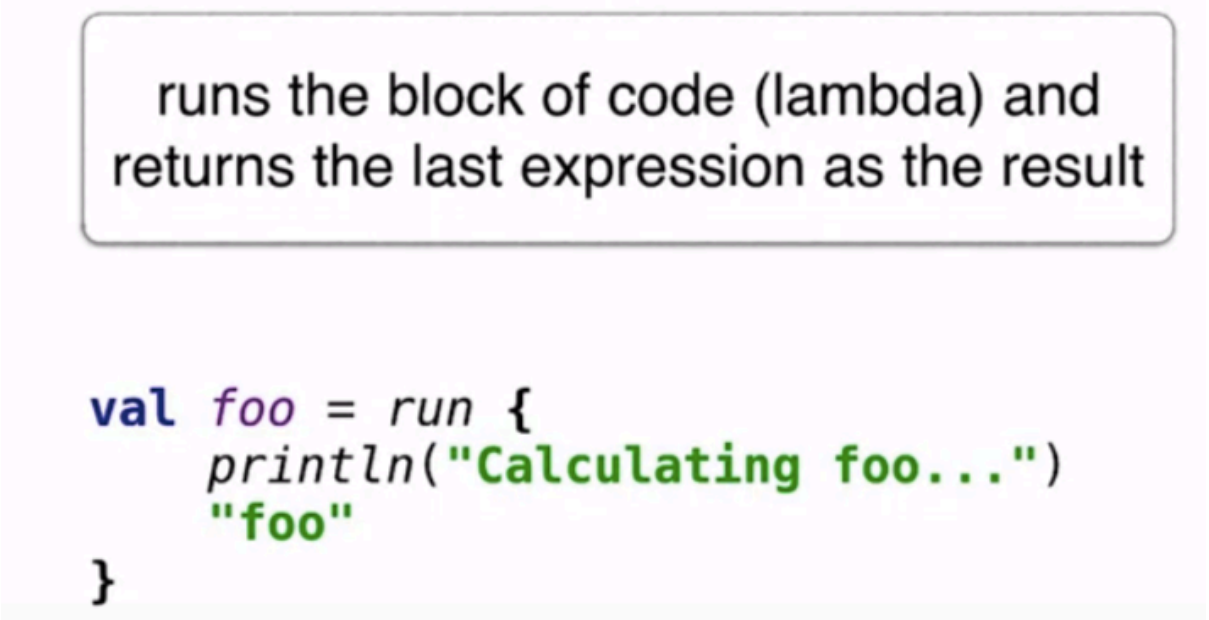
Data Classes

The order of the destructured variables is set. For data class, the compiler simply follows the **order of the constructor val arguments**.
Note that you can omit one of the new variables if you are not going to use it. Put **underscore (_)** instead of its name. Then, it takes part in destructuring, but the variable is not created

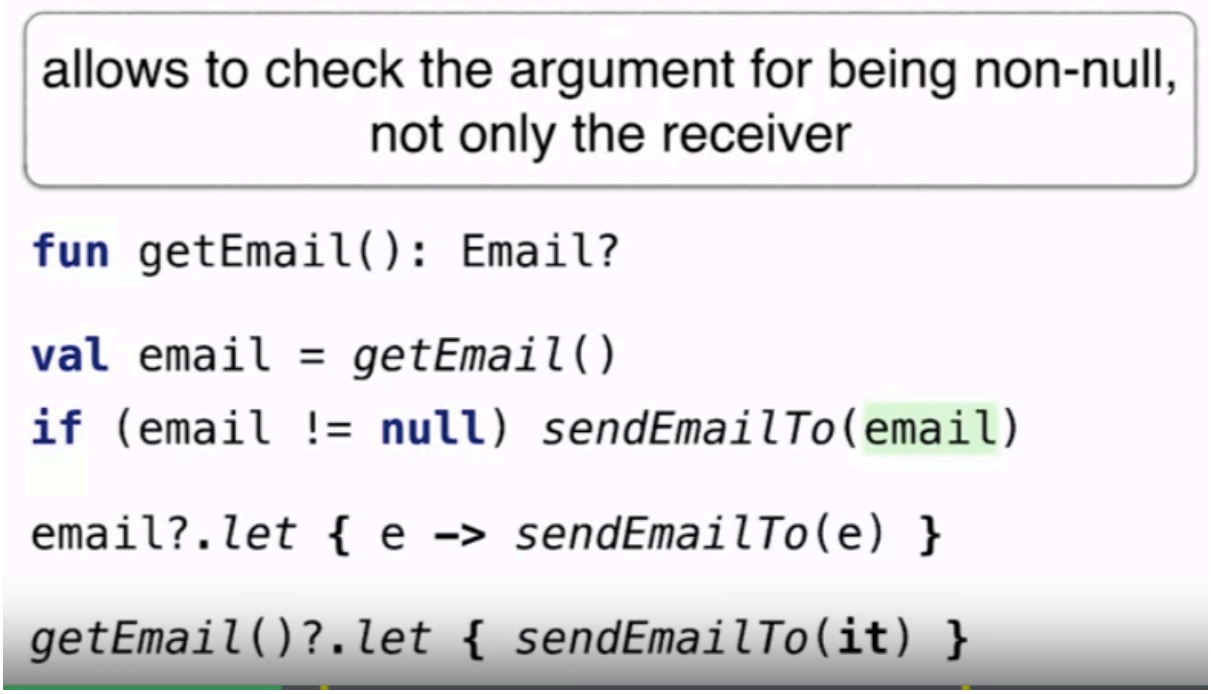


Inline functions

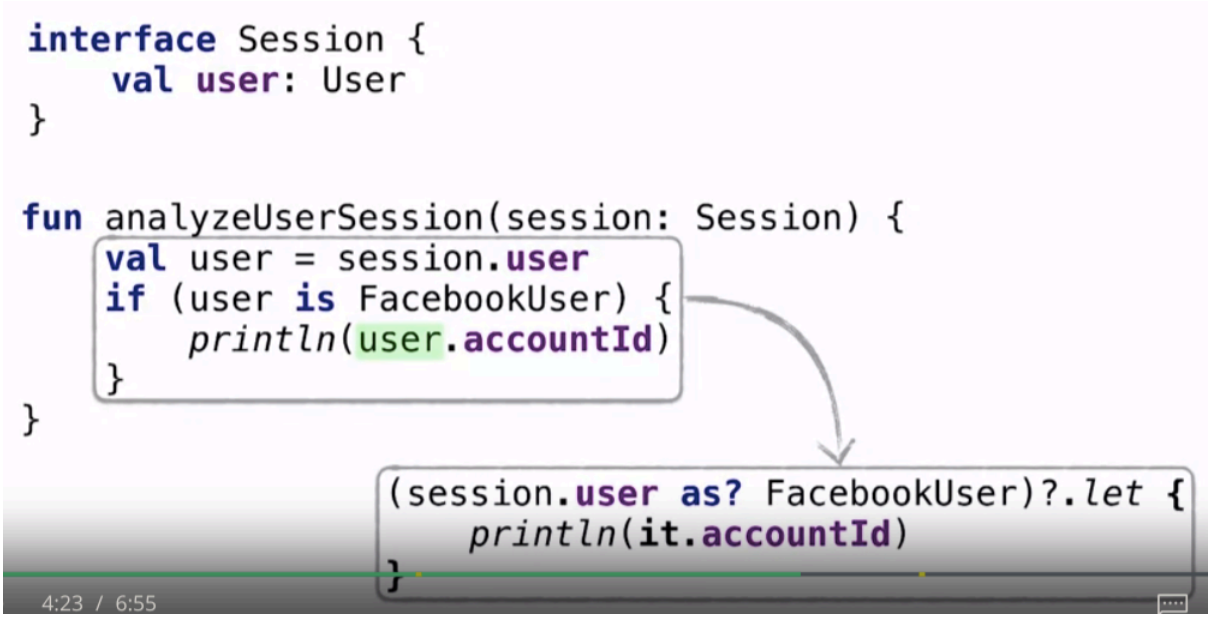
run



let



Remember, the example when we have a **property defined in an interface**, it's an **open property**, that's why you **can't smart cast it**. The compiler doesn't know how it can be written in this App classes. **To smart cast, you need to copy the property to a new local variable**. **Let gives you an alternative**. If the property session user can be cast to Facebook user, we perform an action on it. Otherwise, nothing happens



takeIf

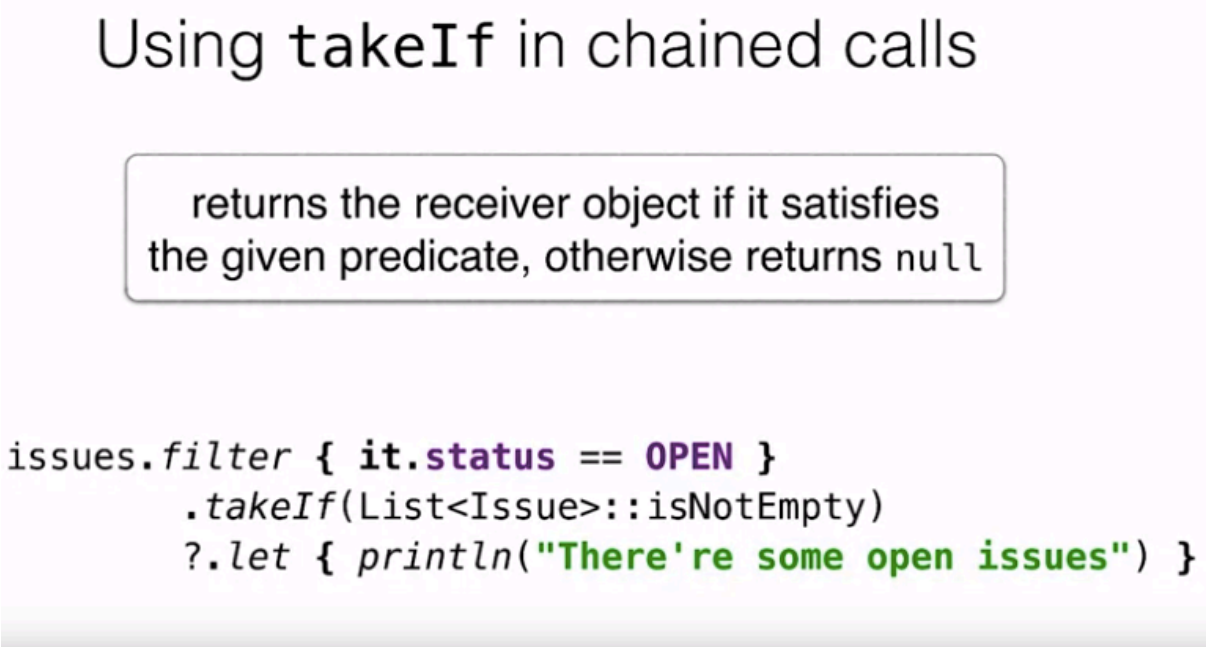
returns the receiver object if it satisfies the given predicate, otherwise returns `null`

```
issue.takeIf { it.status == FIXED }

person.patronymicName.takeIf(String::isNotEmpty)
```

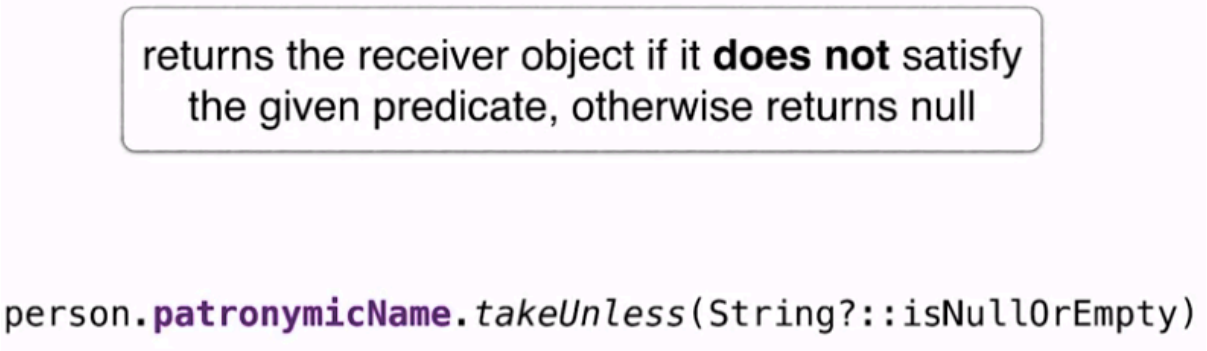
What is the result of `takeIf` call below?

```
val number = 42
number.takeIf { it > 10 }      42

val other = 2
other.takeIf { it > 10 }     // null
```

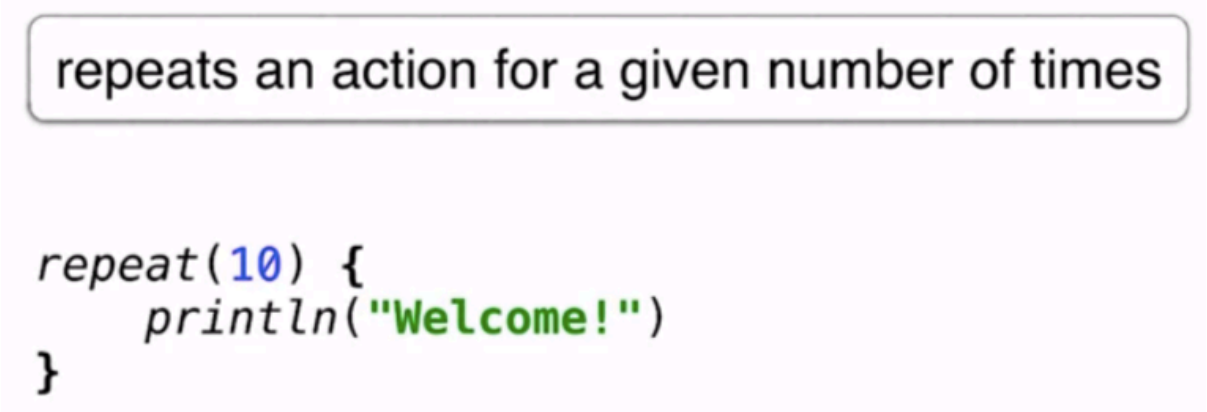
Using takeIf in chained calls



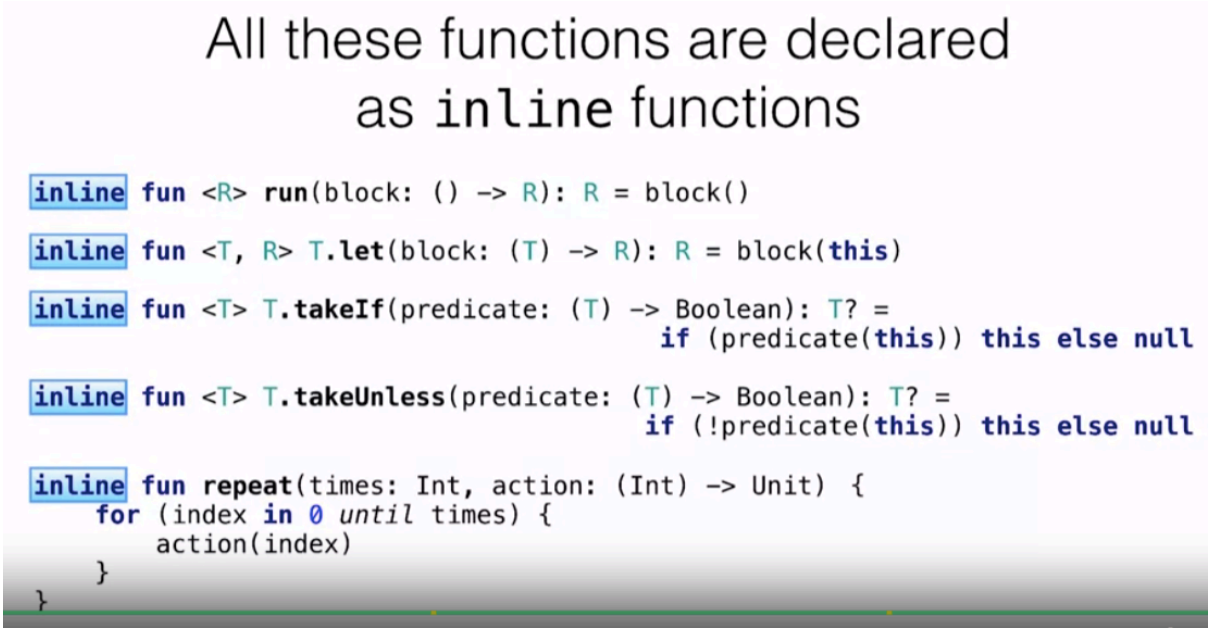
takeUnless



repeat



Inline Functions Declarations



The power of inline functions

Inline functions does not generate performance overhead because the compiler substitute its code where it is called.

Inlining of run

Run is the simplest example. It only calls the lambda and does nothing else. At the bytecode level while generating the code for their own code, the compiler instead of calling run and creating an object called the lambda, will **generate the lambda body directly**. As a result, we have no performance over head or creating an anonymous class and an object for the lambda

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

```
val name = "Kotlin"
run { println("Hi, $name!") }
```

Generated code (in the bytecode):

```
val name = "Kotlin"
println("Hi, $name!")
```

inlined code of lambda body

Inlining of takeUnless

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

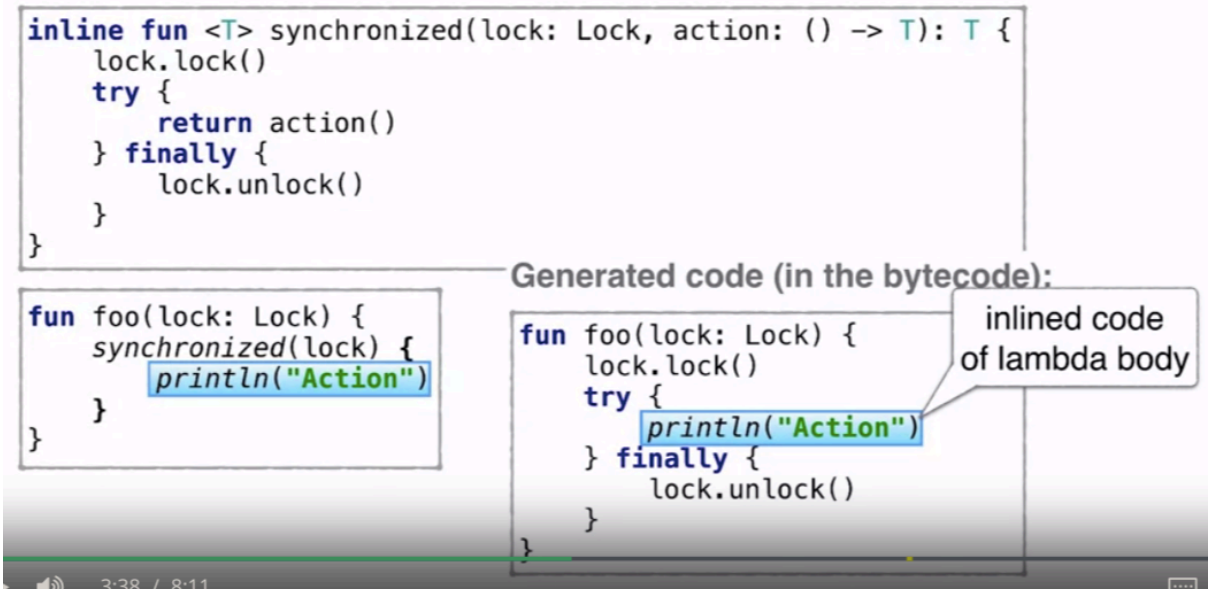
```
fun foo(number: Int) {
    val result = number.takeUnless { it > 10 }
    ...
}
```

Generated code (in the bytecode):

inlined code of lambda body

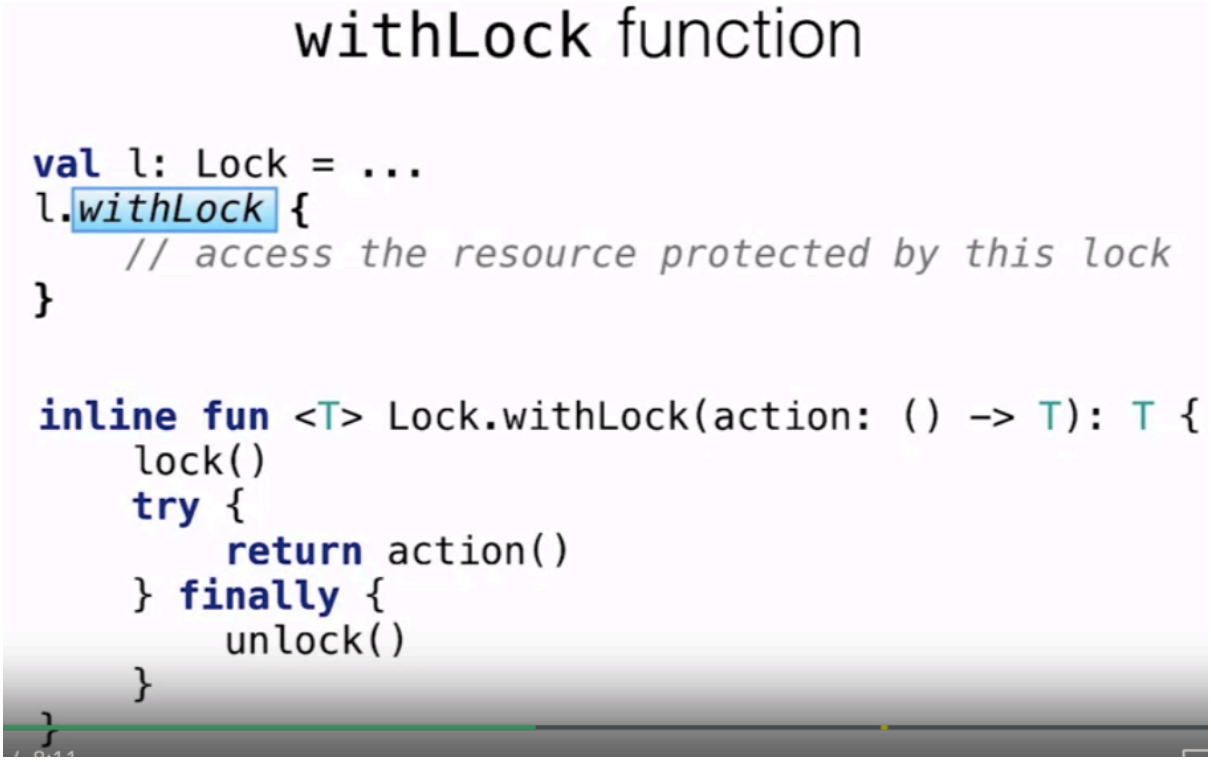
```
fun foo(number: Int) {
    val result = if (!(number > 10)) number else null
    ...
}
```

Inlining of synchronized



withLock Function

Note that whenever you need to do something with lock instead of synchronized in kotlin, you can use **withLock**. It does a similar thing but it is defined as an extension.



No performance overhead
when you use

`run, let, takeIf, takeUnless, repeat,`
`withLock, use`

No anonymous class or extra objects
are created for lambda under the hood

@InlineOnly

That means that this function is always inlined

`@InlineOnly`

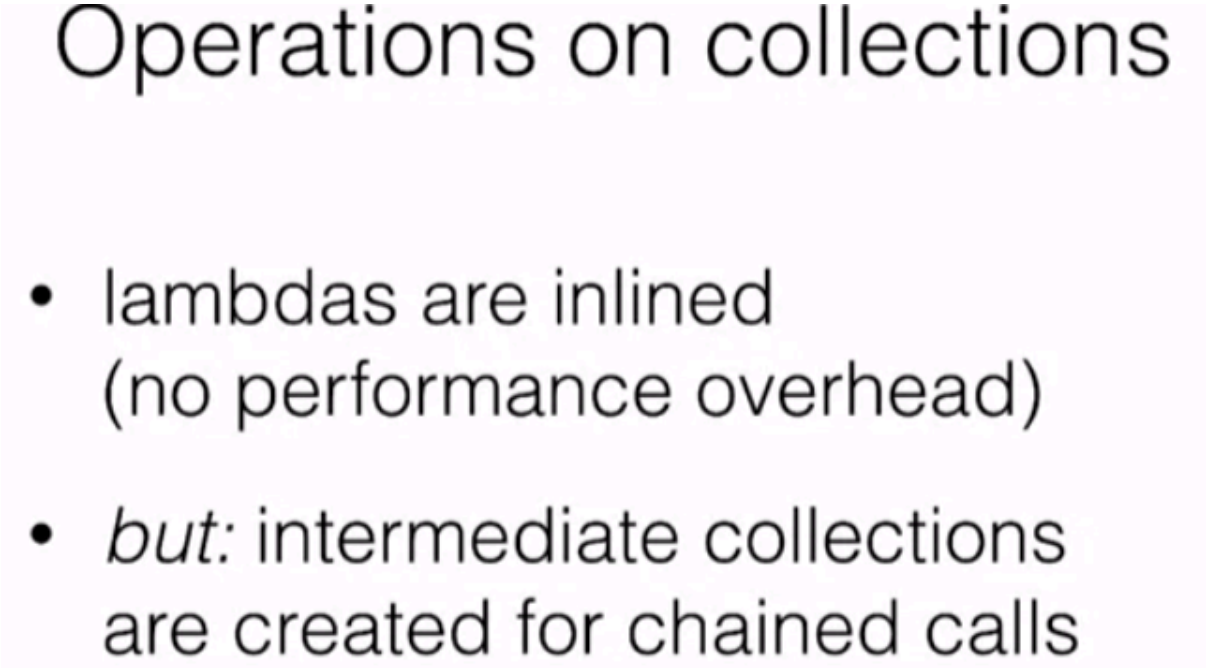
Specifies that this function should not
be called directly without inlining

```
@kotlin.internal.InlineOnly
public inline fun <R> run(block: () -> R): R = block()
```

Sequences

Collections vs Sequences

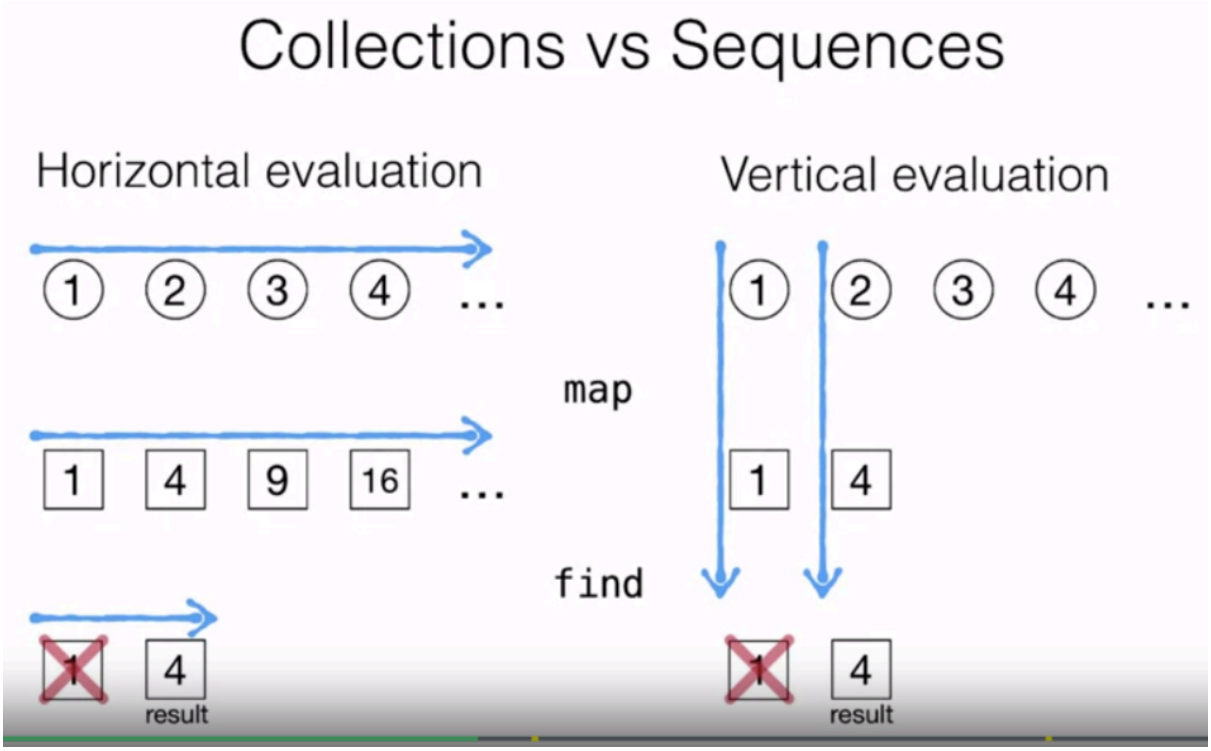
The benefit of in-lining the lambdas is as we already know, that there is no performance overhead. For instance, when you need on the filtering, you call filter and everything is fine. However, if you have a **chain of calls**, several calls one after another, then you'll have a different kind of problem, probably a bigger performance overhead because an intermediate collection will be created for each of the chained calls. There is an option how to avoid it. This option is called sequences.
Sequences can be compared in a sense with the Java eight streams because sequences and streams, both perform computations in a lazy manner.
If we use operations on collections, they eagerly return the result, while the operations on sequences postponed the actual computation, and therefore avoid creating intermediate collections.



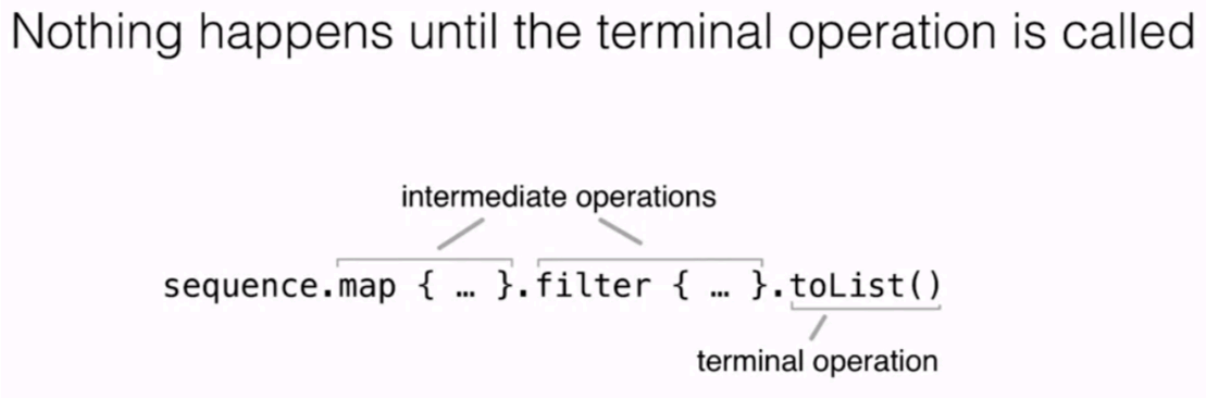
From List to Sequence

```
val list = listOf(1, 2, -3)
val maxOddSquare = list
        .asSequence()
        .map { it * it }
        .filter { it % 2 == 1 }
        .max()
```

Operations

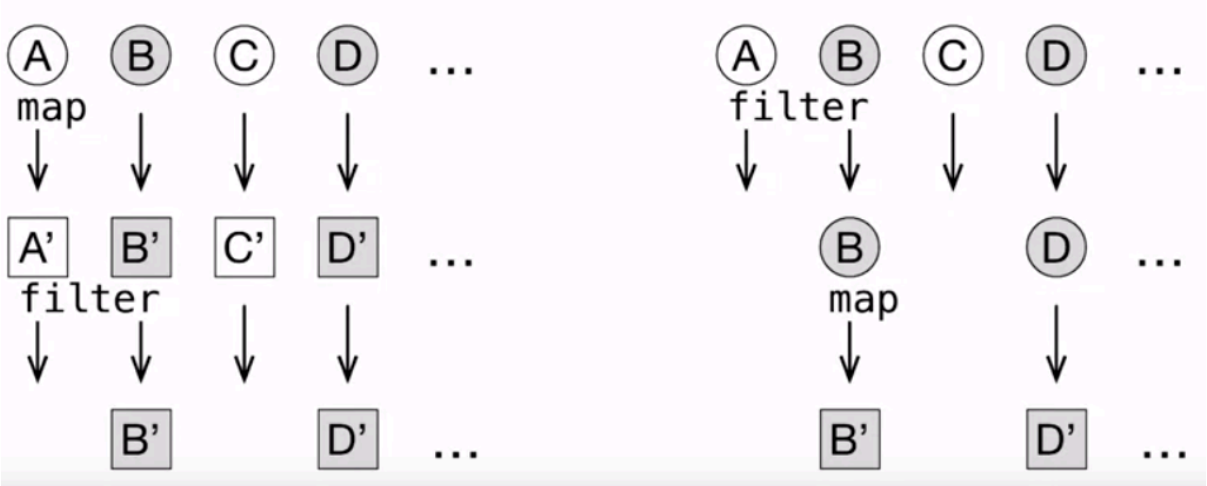


Intermidial and Terminal Operations



Intermediate operations are the ones that **return another sequence** are **not marked as inline**. The operations to be performed must be stored to make it possible to call them later. Postponing the operations until they are needed is the core idea of the lazy evaluation. Because the action in a lambda must be stored somewhere, that means that the corresponding lambda cannot be inline. **Terminal** operations which **return a primitive value**, **or a type, or a collection** can be declared as an **inline**

Order of operations is important



Generating a Sequence

If you need to build a sequence from scratch, for instance, you **define a way to receive each new element** from the network, you can use the generateSequence function. Here, it generates a sequence of random numbers. **When the lambda that computes each new element returns null, the sequence is finished**.
Note how we used the takeIf function to check the predicate, and return null if this predicate is not satisfied

```
generateSequence { Random.nextInt() }
```

```
val seq = generateSequence {
    Random.nextInt(5).takeIf { it > 0 }
}
println(seq.toList())
```

```
sample output: [4, 4, 3, 2, 3, 2]
```

Reading input

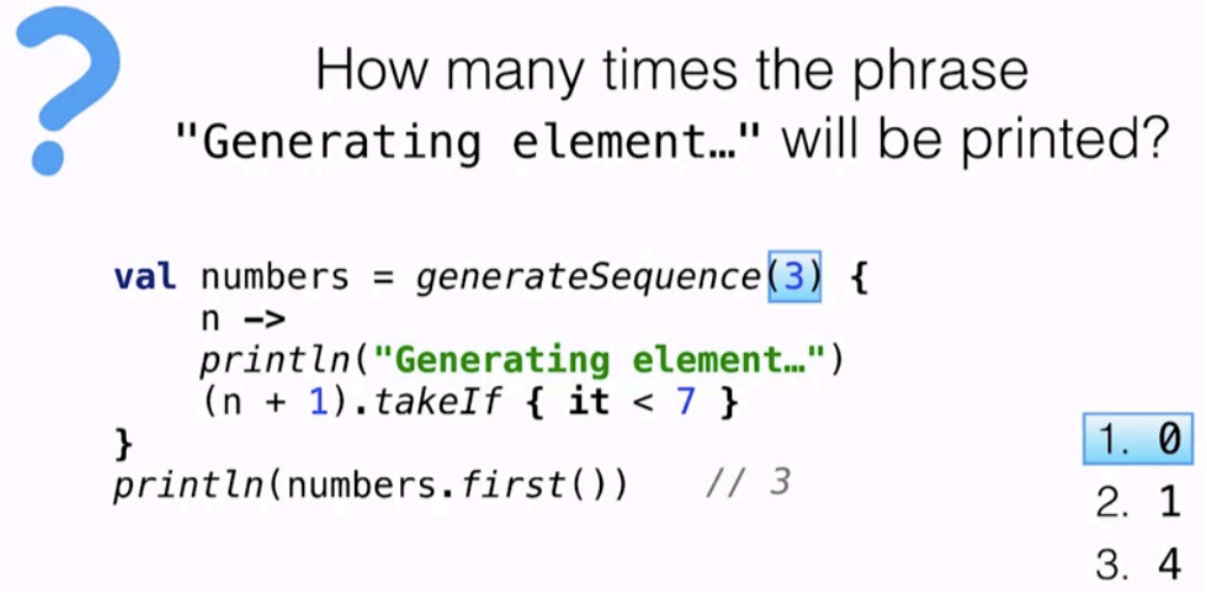
```
val input = generateSequence {
    readLine().takeIf { it != "exit" }
}
println(input.toList())
```

```
>>> a
>>> b
>>> exit
[a, b]
```

Generating an infinite sequence

```
val numbers = generateSequence(0) { it + 1 }
numbers.take(5).toList()                    [0, 1, 2, 3, 4]


// to prevent integer overflow:
val numbers = generateSequence(BigInteger.ZERO) {
    it + BigInteger.ONE
}
```

Excercise



The right answer is zero because we only asked for the first element, and this first element is already given. Because sequences are evaluated lazily, nothing happens until you explicitly ask for it. Here, we only ask for the first element, so this lambda is never called.

When you explicitly ask for other elements, then the necessary elements are generated.

```
println(numbers.toList())      Generating element...
                               Generating element...
                               Generating element...
                               Generating element...
                               [3, 4, 5, 6]
```

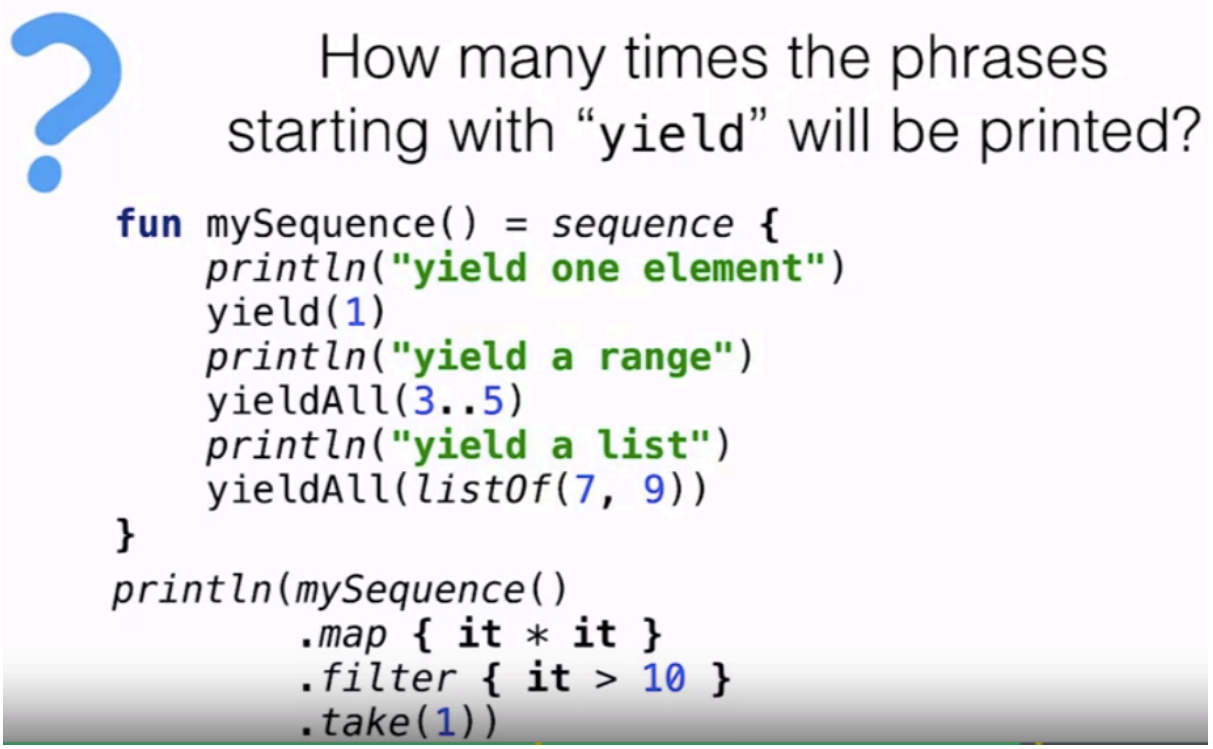
Yield

Another important functionality that you can use to generate a sequence is yield. Yield allows you to yield elements in a custom way. The lambda there generates a sequence element must be either based on external sources or based on the previous element, but you cannot really customize that. With yield, you can generate any elements in any convenient order. You need to use yield inside lambda argument of the sequence function.

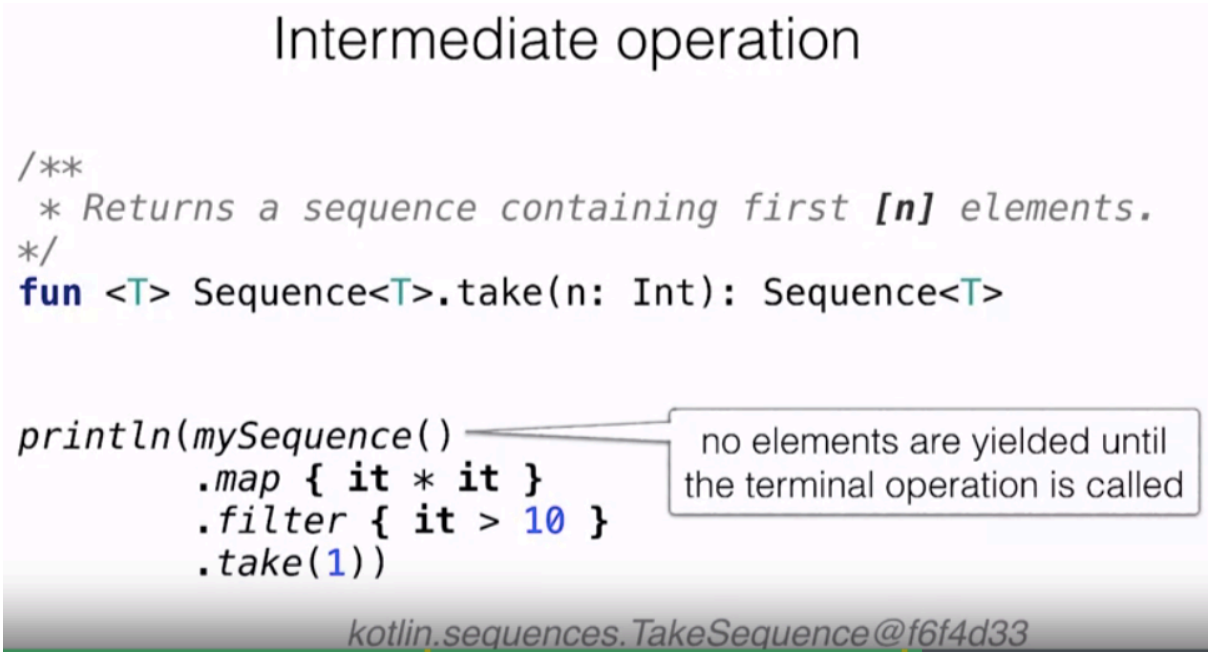
```
val numbers = sequence {
    var x = 0
    while (true) {
        yield(x++)
    }
}
numbers.take(5).toList()  // [0, 1, 2, 3, 4]
```

```
sequence {
    yield(value)
    doSometing()
    yieldAll(list)
    doSomethingElse()
    yieldAll(sequence)
}
```

Excercise



The right answer is zero. It was a tricky question because the take function is only an intermediate operation, it returns another sequence. But no elements are yielded until the terminal operation is called. In this case, no terminal operation is called, so nothing happens.



If we put a terminal operation

Building a sequence in a lazy manner

```
fun mySequence() = sequence {
    println("yield one element")
    yield(1)
    println("yield a range")
    yieldAll(3..5)
    println("yield a list")
    yieldAll(listOf(7, 9))
}

println(mySequence()
        .map { it * it }
        .filter { it > 10 }
        .first())
```

```
yield one element
yield a range
16
```

```
1  3  4  …
1  9  16
-  -  16
16
```

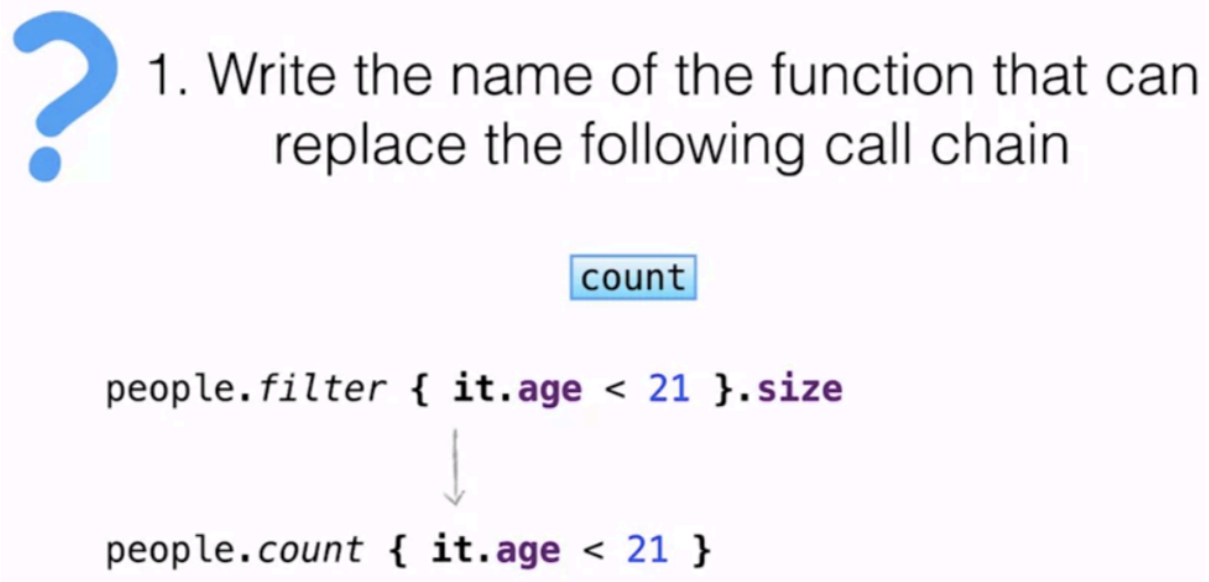
Building a sequence in a lazy manner

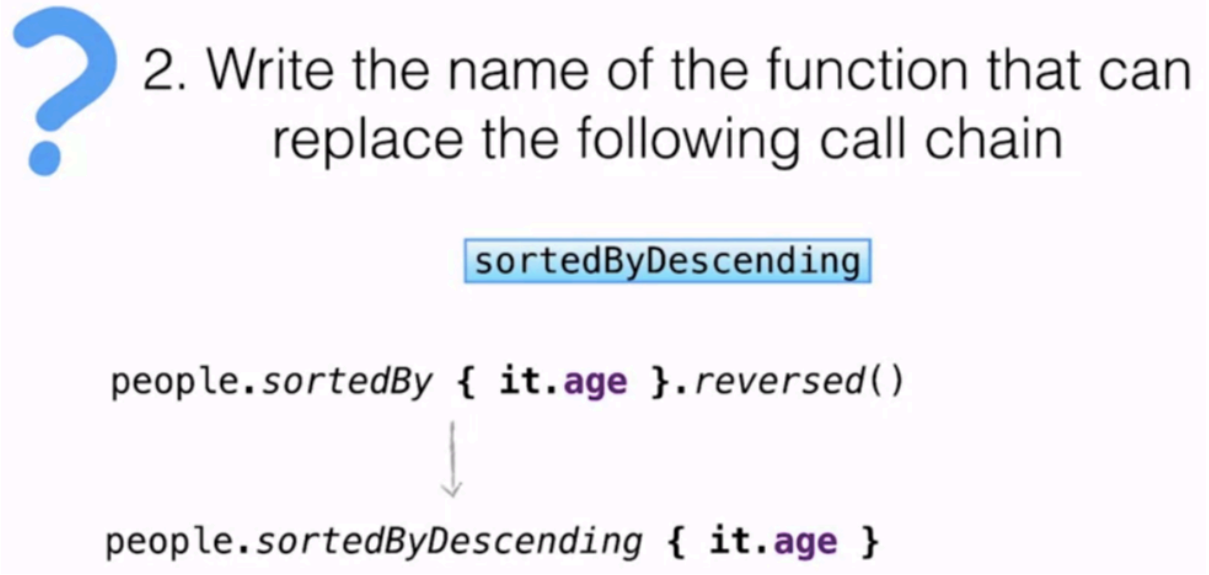
```
fun mySequence() = sequence {
    println("yield one element")
    yield(1)
    println("yield a range")
    yieldAll(3..5)
    println("yield a list")
    yieldAll(listOf(7, 9))
}

println(mySequence()
        .map { it * it }
        .filter { it > 10 }
        .first())
```

won't be called

```
1  3  4  …
1  9  16
-  -  16
16
```

Library Functions Exercises





3. Write the name of the function that can replace the following call chain

mapNotNull

```
people
    .map { person ->
        person.takeIf { it.isPublicProfile }?.name
    }
    .filterNotNull()
```

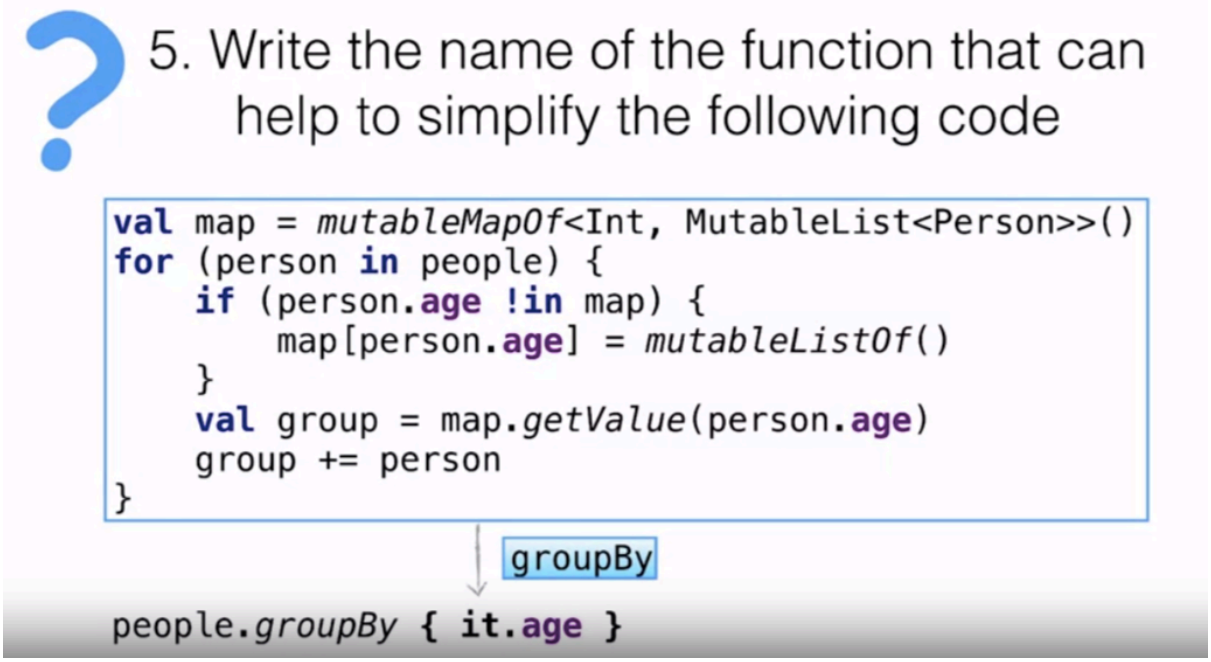
```
people.mapNotNull { person ->
    person.takeIf { it.isPublicProfile }?.name
}
```

4. Write the name of the function that can help to simplify the following code

```
val map = mutableMapOf<Int, MutableList<Person>>()
for (person in people) {
    if (person.age !in map) {
        map[person.age] = mutableListOf()
    }
    val group = map.getValue(person.age)
    group += person
}
```

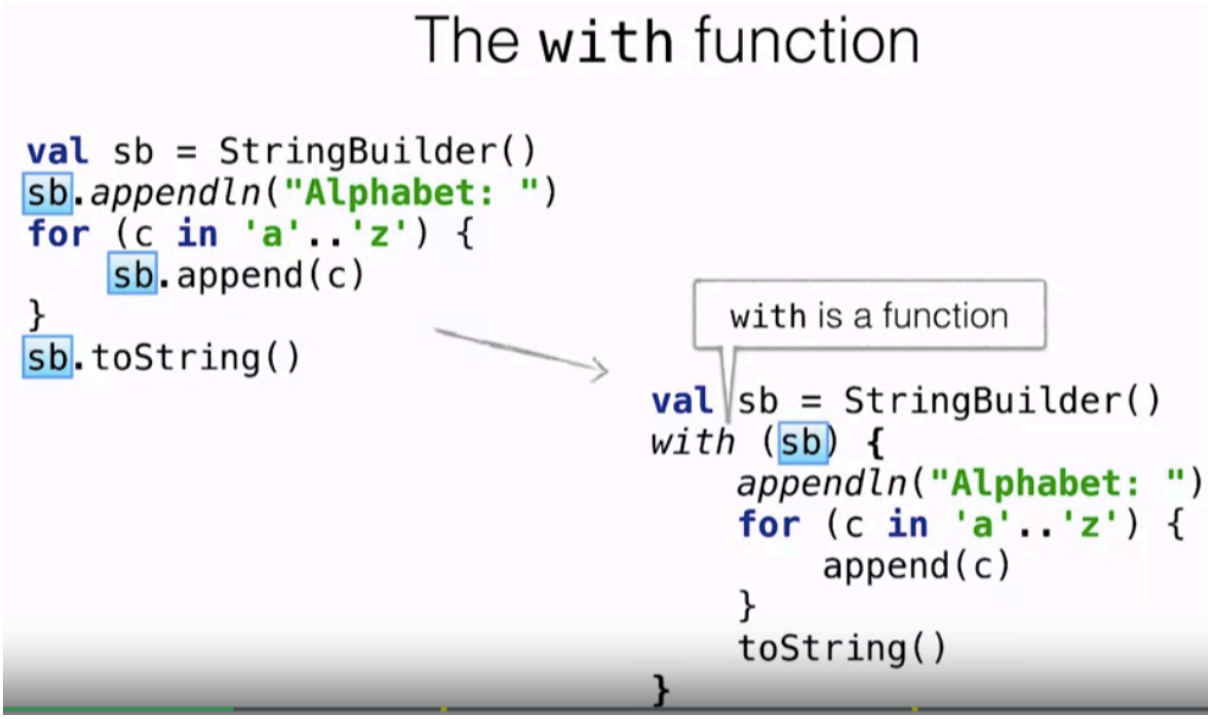
getOrPut

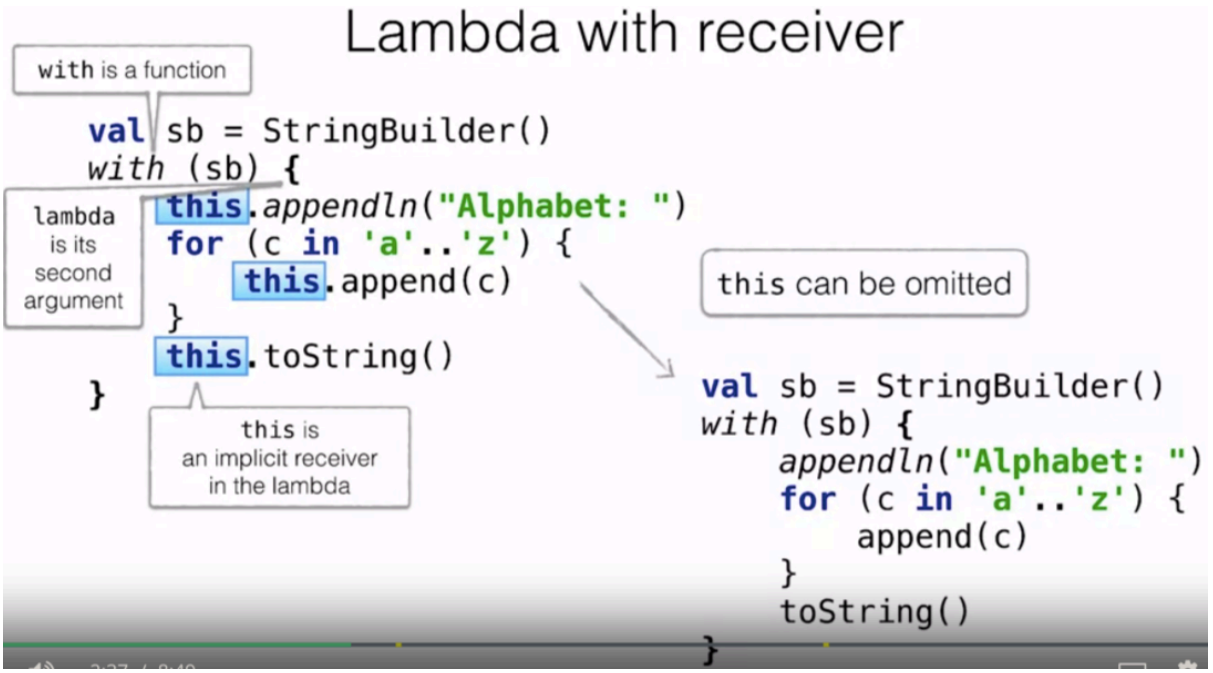
```
val group = map.getOrPut(person.age) { mutableListOf() }
group += person
```



Lambda with receiver

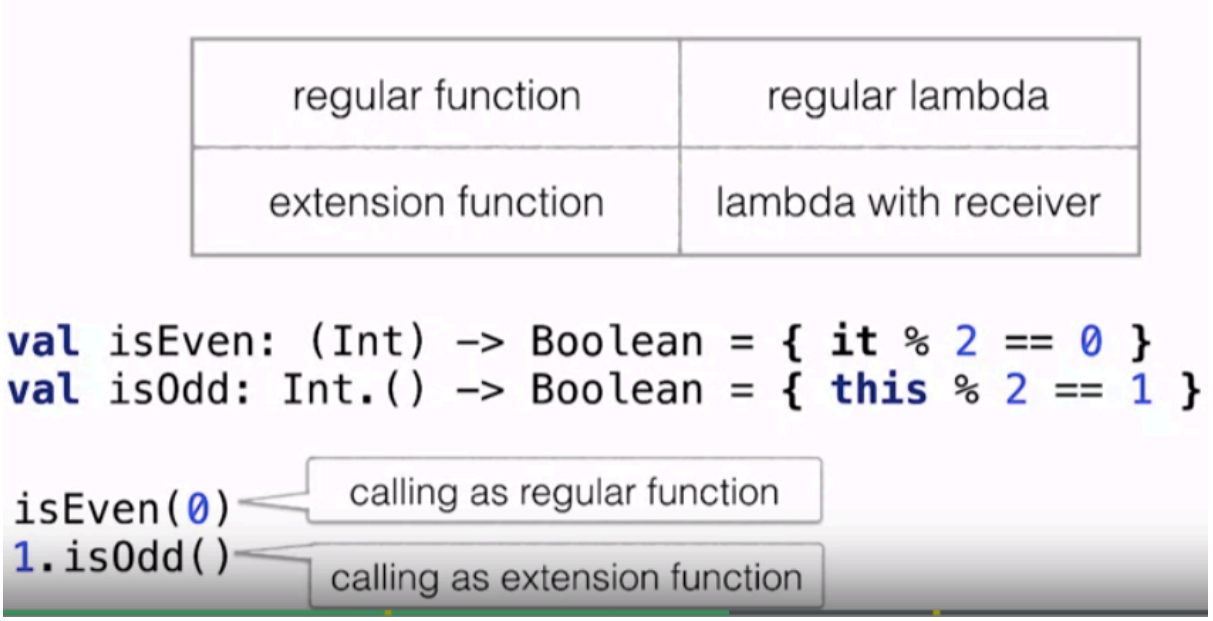
In essence, it might be considered as a union of two ideas of two other features: **extension functions and lambdas**.





Lambda vs Lambda with receiver

You can see how the type for lambda with receiver looks in Kotlin. You **put a receiver type before the parameter lis**t, then **follows the dot**, then the perimeters as usual in the parenthesis. The return type is specified after the arrow, like forever the lambda. When you call a regular lambda, you call it as a regular function. You can call lambda of the receiver, you call it as an extension function. You see this correspondence. Regular lambdas correspond to regular functions, lambda's with the receiver correspond to extension functions.



More useful library functions

```
inline fun <T, R> with(receiver: T, block: T.() -> R): R = receiver.block()
inline fun <T, R> T.run(block: T.() -> R): R = block()
inline fun <T, R> T.let(block: (T) -> R): R = block(this)
inline fun <T> T.apply(block: T.() -> Unit): T { block(); return this }
inline fun <T> T.also(block: (T) -> Unit): T { block(this); return this }
```

with

We've already seen **with**, it **takes an expression as an argument and uses it as the receiver inside of the following lambda**. Here we can **access members** of window without explicit specification. We're just directly window properties.

```
with (window) {
    width = 300
    height = 200
    isVisible = true
}
```

run:

Like **with**, but extension.
Run is very similar to with but it is defined as an extension, which makes it **possible to use it with a null-able receiver**. If the receiver, window in our case, can be null, then you can't use with because this will be null-able inside the lambda, then these can't be admitted, you will need to explicitly check whether it's not null. Run is helpful for such a case because you can use it with safe access. Run will be called only when the receiver is not null. But note that in many other cases run and with are interchangeable.

```
val windowOrNull = windowById["main"]
windowOrNull?.run {
    width = 300
    height = 200
    isVisible = true
}
```

You can use the whole expression as the receiver, here the run will be called only when windowById main exists and is not null.

```
windowById["main"]?.run {
    width = 300
    height = 200
    isVisible = true
}
```

apply:

return receiver as result.
Apply is different because it **returns the receiver as a result**. We haven't used it, but actually with and drawn functions return the result of the lambda. **The last expression inside the lambda will be the result of the whole invocation**. Sometimes, however, it's convenient to return. the receiver as the result. For instance, in a chain of calls, you can call other methods afterward or again like in this example, simply **assign the results to a variable**. When you create a window instance, you might want to modify some of its properties right away. Apply is really useful for that because the result is the modified main window. Note that we assign the variable here only the result is non-null and if it's null we stop the current execution and return from the outer function.

```
val mainWindow =
    windowById["main"]?.apply {
        width = 300
        height = 200
        isVisible = true
    } ?: return
```

also:

regular argument inside of this
Also is similar to apply, it **returns the receiver as well**. However, there is the difference that **it takes a regular lambda not lambda with a receiver** as an argument. Lambda with the receiver is really useful when you can admit this reference because you only call it's members. However, there are cases when you pass the receiver is an argument, like here when we pass a window as an argument to the function showWindow. In this case, the also function is better because it takes a regular lambda as an argument. Here, **it** inside also **refers to window**. You can introduce the window name for it If you want.

```
windowById["main"]?.apply {
    width = 300
    height = 200
    isVisible = true
}?.also {
    showWindow(it)
}
```

Differences

This table shows you the difference between all these functions. With goes in a cell together with run. **With is only one non-extension function here**. **All other functions including run are extensions**. The **difference between them is which functions return the result of the lambda and which functions return the receiver**. The functions i**n the first column take lambda with the receiver as an argument**, **with run** as **apply**. The r**est two take the regular lambda as an argument**.

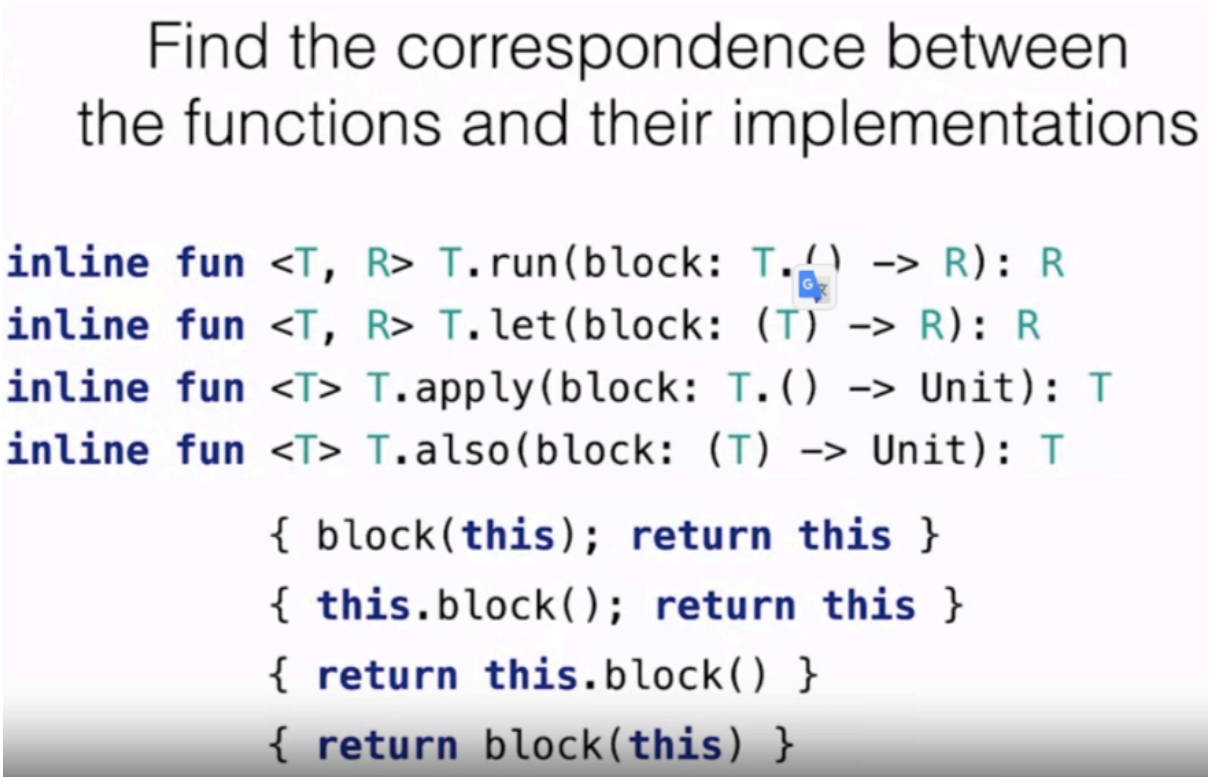
	{ .. this .. }	{ .. it .. }
return result of lambda	with / run	let
return receiver	apply	also

```
receiver.apply {
    this.actions()
}
```

```
receiver.also {
    moreActions(it)
}
```

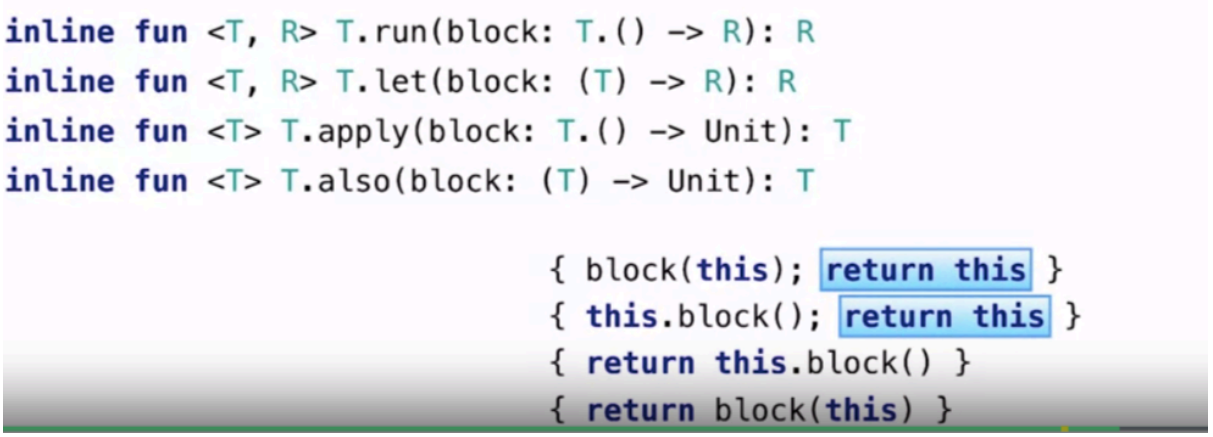
Exercise

Without looking back at the first slide with all the function decorations, you may practice your own understanding and find the correspondence between this function decorations and their implementations



The table is very useful. At first, let's focus on the functions **returning the receiver as the result**. They should return explicit to this because this refers to the receiver inside the lambda.

	{ .. this .. }	{ .. it .. }
return result of lambda	run	let
return receiver	apply	also



The other two functions **return the result of the lambda**

	{ .. this .. }	{ .. it .. }
return result of lambda	run	let
return receiver	apply	also

```
inline fun <T, R> T.run(block: T.() -> R): R
inline fun <T, R> T.let(block: (T) -> R): R
inline fun <T> T.apply(block: T.() -> Unit): T
inline fun <T> T.also(block: (T) -> Unit): T

                                { block(this); return this }
                                { this.block(); return this }
                                { return this.block() }
                                { return block(this) }
```

We now distinguish at least pairs of functions, after that we can go on and focus on which lambda do the functions take as an argument. Two functions **take lambda with receiver,** they expect a parameter of extension function type. **This parameter of extension function type must be called as an extension function**. If we call this block as an extension, then block corresponds to lambda with a receiver.

	{ .. this .. }	{ .. it .. }
return result of lambda	run	let
return receiver	apply	also

```
inline fun <T, R> T.run(block: T.() -> R): R
inline fun <T, R> T.let(block: (T) -> R): R
                                { return this.block() }
                                { return block(this) }

inline fun <T> T.apply(block: T.() -> Unit): T
inline fun <T> T.also(block: (T) -> Unit): T
                                { block(this); return this }
                                { this.block(); return this }
```

The other two functions take a regular lambda and then call a variable of a function type as a regular function.

	{ .. this .. }	{ .. it .. }
return result of lambda	run	let
return receiver	apply	also

```
inline fun <T, R> T.run(block: T.() -> R): R
inline fun <T, R> T.let(block: (T) -> R): R
                                { return this.block() }
                                { return block(this) }

inline fun <T> T.apply(block: T.() -> Unit): T
inline fun <T> T.also(block: (T) -> Unit): T
                                { block(this); return this }
                                { this.block(); return this }
```

That gives us the result.

	{ .. this .. }	{ .. it .. }
return result of lambda	run	let
return receiver	apply	also

```
inline fun <T, R> T.run(block: T.() -> R): R { return this.block() }
inline fun <T, R> T.let(block: (T) -> R): R { return block(this) }
inline fun <T> T.apply(block: T.() -> Unit): T { this.block(); return this }
inline fun <T> T.also(block: (T) -> Unit): T { block(this); return this }
```

Types

Basic Types

Primitives & wrapper types

Kotlin	Java	Kotlin	Java
Int	int	Int?	java.lang.Integer
Double	double	Double?	java.lang.Double
Boolean	boolean	Boolean?	java.lang.Boolean

Generic arguments

Kotlin	Java
List<Int>	List<Integer>

Array of primitive types

Kotlin	Java
Array<Int>	Integer[]
IntArray	int[]

String

Kotlin	Java
kotlin.String	java.lang.String

hides some confusing methods

```
"one.two.".replaceAll(".", "*")                    ********
"one.two.".replace(".", "*")                       one*two*
"one.two.".replace(".".toRegex(), "*")             ********
```

Type hierarchy

Unit instead of void

In Kotlin, Unit is **used always instead of void**. **Whenever you use void in Java, you use Unit in Kotlin**. It denotes that **no meaningful value is returned** from this function. These two syntactic forms are equivalent. You can either specify unit explicitly, which no one does, or you can simply omit it. At the Bytecode, unit is replaced with Java void. In Kotlin, you always use it instead of void for the function that has no return value

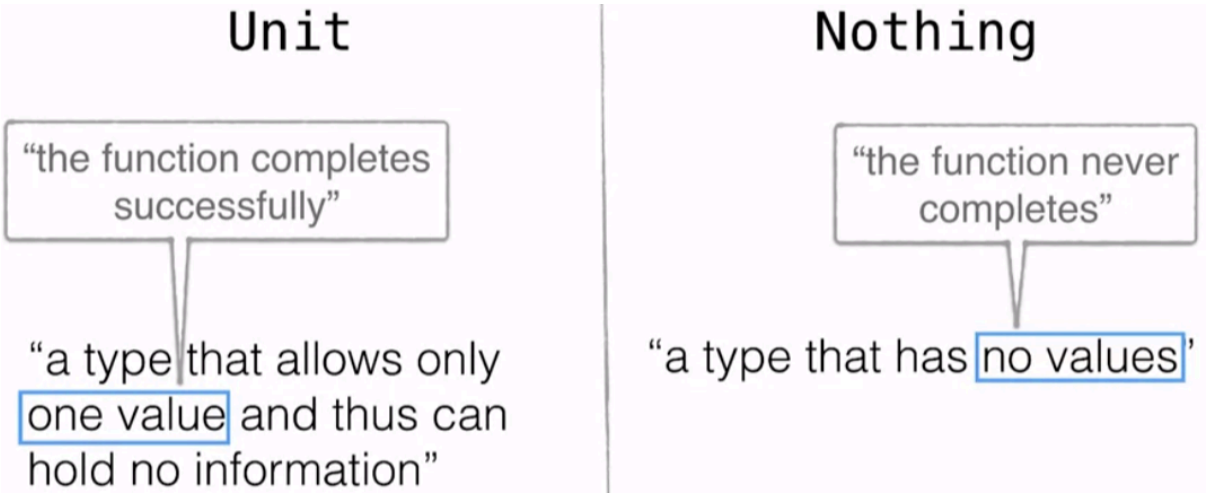
```
fun f() { /*...*/ }

fun f(): Unit { /*...*/ }
```

Nothing is different Unit/Void

Nothing is different to Unit and void. It means, **the function never returns**. You specify Nothing return type for the function that only throws an exception. In Kotlin, you can define such a function and use it instead of throwing an exception. Imagine there is a specific for your use case exception, and you're always thrown to read different details in messages. You can extract throwing this exception into a function and later call this function instead of all places where you had to throw an exception explicitly. If you specify that this function returns Nothing type, then the Kotlin compiler uses the information that this function can't be completed normally for type inference and while locating the dead code.

```
fun fail(message: String): Nothing {
    throw IllegalStateException(message)
}
```

It means “this function never returns”



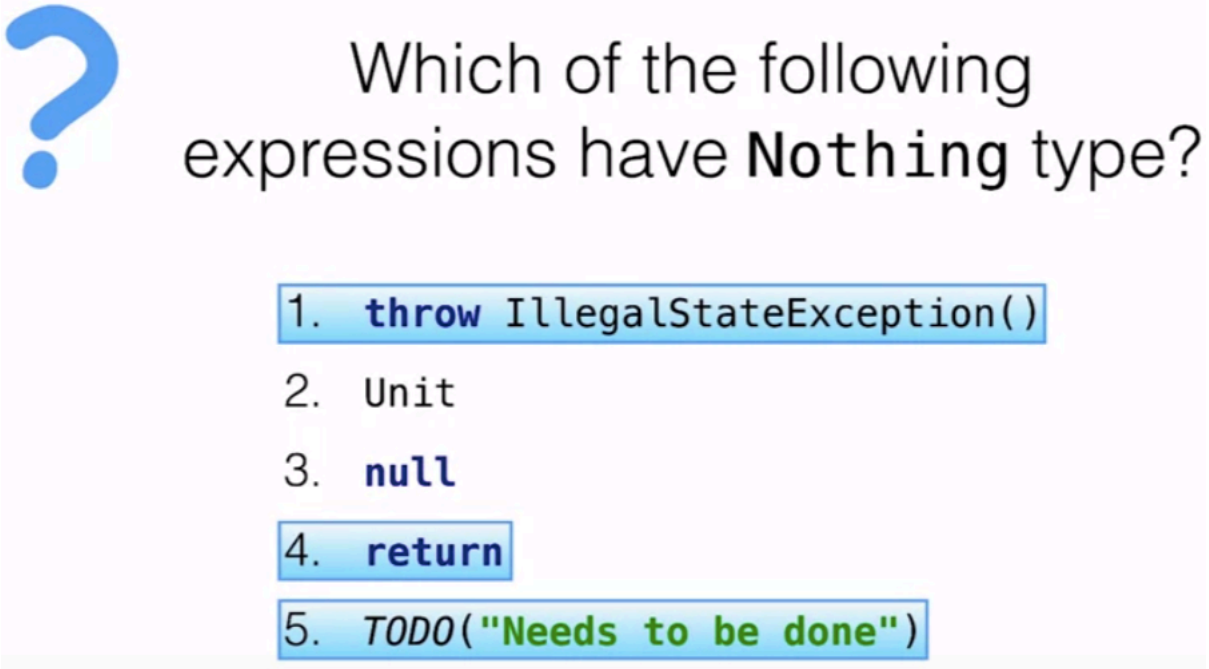
Expression of Nothing type

TODO is a function defined in the standard library. It throws node implemented error. You can use throw as a part of other expressions.

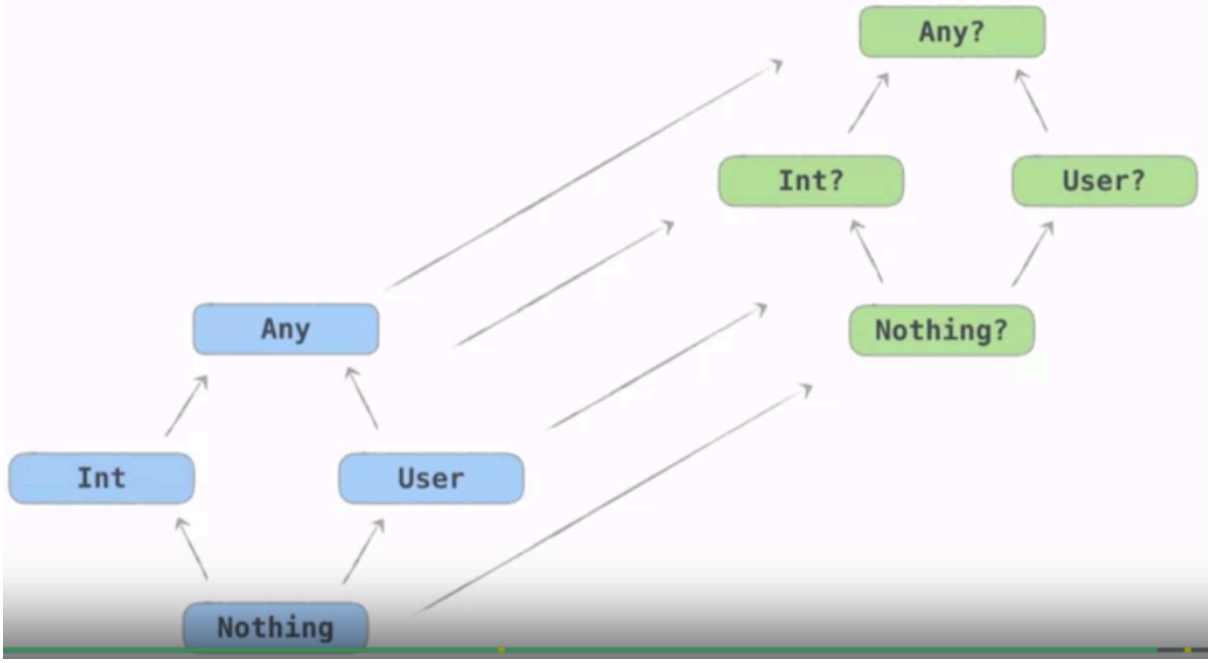
```
val answer: Int = if (timeHasPassed()) {
    42
} else {
    TODO("Needs to be done")
}
```

```
inline fun TODO(reason: String): Nothing =
    throw NotImplementedError("An operation is not implemented: $reason")
```

Question



Type hierarchy



Collection Types

List & Mutable List

Read-only != Inmutable

Note that we specifically used the broad read-only to describe the interface list in Kotlin. It's **not the same as immutable**. It's not immutable data structure which cannot be changed. No. A read-only interface, just like mutating methods. The actual list **can be modified via a different reference.**