{ this is Kotlin }

Functions

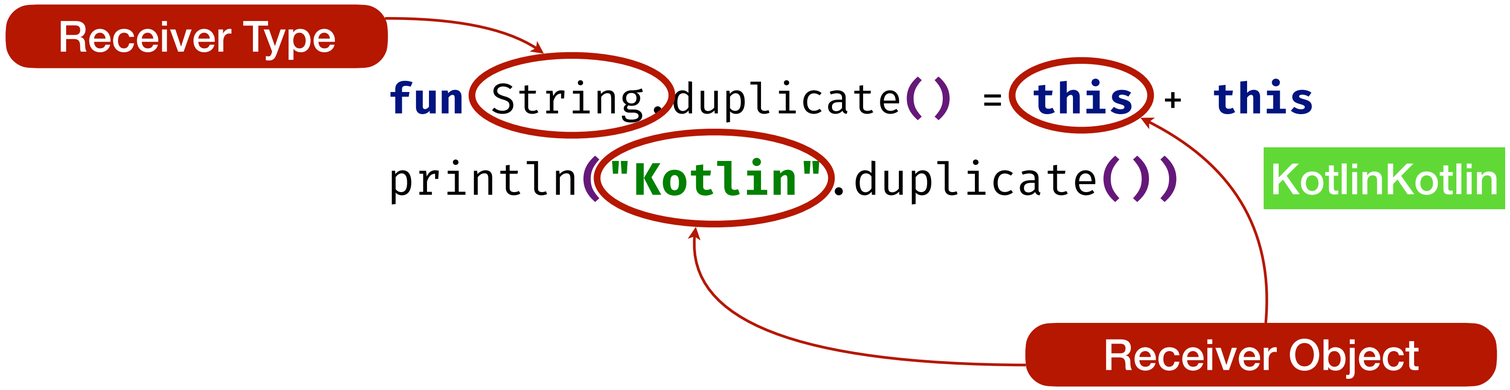
Tiberiu Tofan

Extension Function

declaration

```
fun duplicate(s: String) = s + s
println(duplicate("Kotlin"))
```

KotlinKotlin



Extension Function

default/named params

```
fun String.duplicate(separator: String = ""): String = "$this$separator$this"

println("Kotlin".duplicate(separator = " vs "))
```

Kotlin vs Kotlin

Extensions Scope

top level

```
package dev.school.extensions

fun String.isPalindrome() = this == this.reversed()
```

```
import dev.school.extensions.isPalindrome

fun main() {
    println("radar".isPalindrome())
}
```

Extensions Scope

member

```
class StringAlgorithms {
    fun String.isPalindrome() = this == this.reversed()

    fun printResult(s: String) = println(s.isPalindrome())
}


fun main() {
    val alg = StringAlgorithms()
    alg.printResult("radar") true
    println("radar".isPalindrome())
}                            compilation error: Unresolved reference: isPalindrome
```

Extensions Scope

local

```
fun main() {
    fun solution(): Boolean {
        fun String.isPalindrome() = this == this.reversed()
        return "radar".isPalindrome()
    }
    println(solution())
    println("radar".isPalindrome())
}
```

true

compilation error: Unresolved reference: isPalindrome

Statically Resolved

- extensions do not modify the classes they extend
- only make new functions and properties callable with the dot-notation
- dispatched *statically*

Statically Resolved

calling from Java

```kotlin
//Kotlin file: extensions.kt
package dev.school.extensions

fun String.duplicate(separator: String = ""): String
```

```java
//Java
import dev.school.extensions.ExtensionsKt;

public class ExtensionsDemo {
    public static void main(String[] args) {
        System.out.println(ExtensionsKt.duplicate("Kotlin", "_"));
    }
}
```

Higher order functions

- Does at least one of:

✓ takes a function as parameter

```
fun applyTwice(i: Int, f: (Int) → Int): Int = f(f(i))
```

✓ returns a function

```
fun addTwo(): (Int) → Int {
    return { i → i + 2 }
}
```

💻 How should this function be implemented?

```
fun twice(f: (Int) → Int): (Int) → Int = TODO()
```

Function types

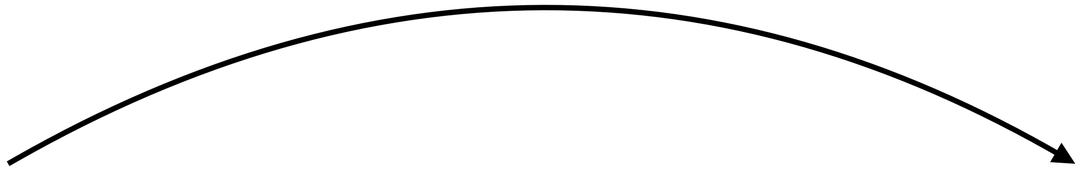
```
(Int) → String
```

```
(A, B) → C
```

- function with two parameters
- the first parameter has the type **A**
- the second parameter has the type **B**
- the function evaluates to a value of type **C**

Function types
naming function types

```
typealias UnaryInt = (Int) -> Int
```



```
fun twice(f: (Int) → Int): (Int) → Int
```

```
fun twice(f: UnaryInt): UnaryInt
```

Lambda expressions

an anonymous function that can be used as an expression

Lambda expressions

```
{ s: String -> s.length }

{ a: Int, b: Int ->
    println("$a + $b")
    a + b
}
```

- a lambda expression is always surrounded by curly braces
- parameter declarations in the full syntactic form go inside curly braces and have optional type annotations
- the body goes after an `->` sign
- if the inferred return type of the lambda is not `Unit`, the last (or possibly single) expression inside the lambda body is treated as the return value

Lambda expressions

are expressions

```
{ a: Int, b: Int -> a + b }
```

Lambda expressions

are expressions

```
val sum = { a: Int, b: Int -> a + b }
```

Lambda expressions

are expressions

```
val sum : (Int, Int) -> Int = { a: Int, b: Int -> a + b }
```

Lambda expressions

feel like language constructs

```
fun applyTwice(i: Int, f: (Int) → Int): Int
```

```
applyTwice(2, { n -> n + 2 })
```

last parameter outside parentheses

```
applyTwice(2) { n -> n + 2 }
```

default param name

```
applyTwice(2) { it + 2 }
```

Lambda expressions

HOFs that return a function

```
fun twice(f: (Int) -> Int): (Int) -> Int = { f(f(it)) }
```

```
: (Int) → Int
val quartic = twice { it * it }
val res = quartic(3) 81
```

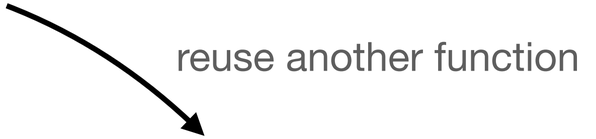
single liner →

```
val res = twice { it * it }.invoke(3)
val res = twice { it * it }(3)
```

Function references

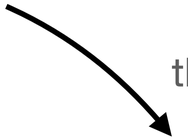
::

```
fun applyTwice(i: Int, f: (Int) → Int): Int
fun addTwo(i: Int): Int = i + 2

applyTwice(2) { it + 2 }
```



reuse another function

```
applyTwice(2) { addTwo(it) }
```

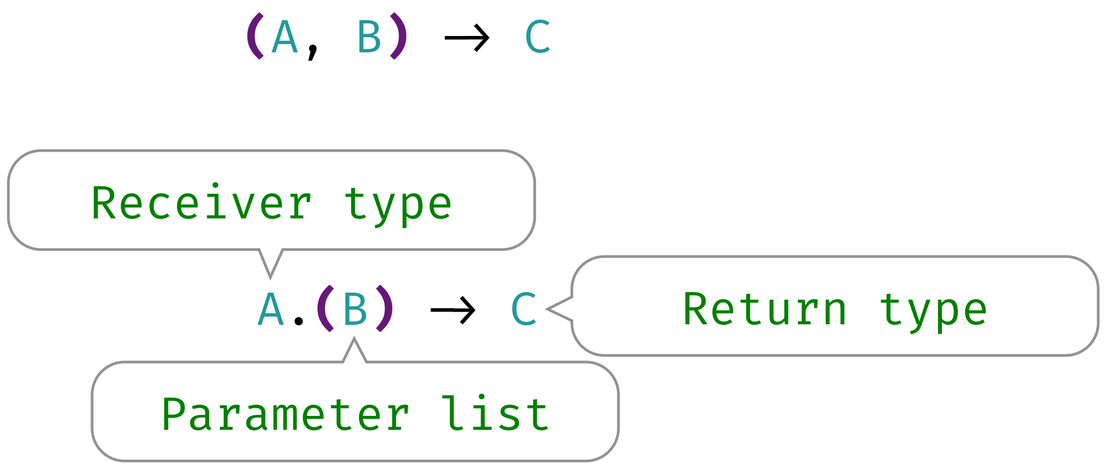


the lambda is just a proxy to another function

```
applyTwice(2, ::addTwo) 6

applyTwice(2, Int::dec) 0
```

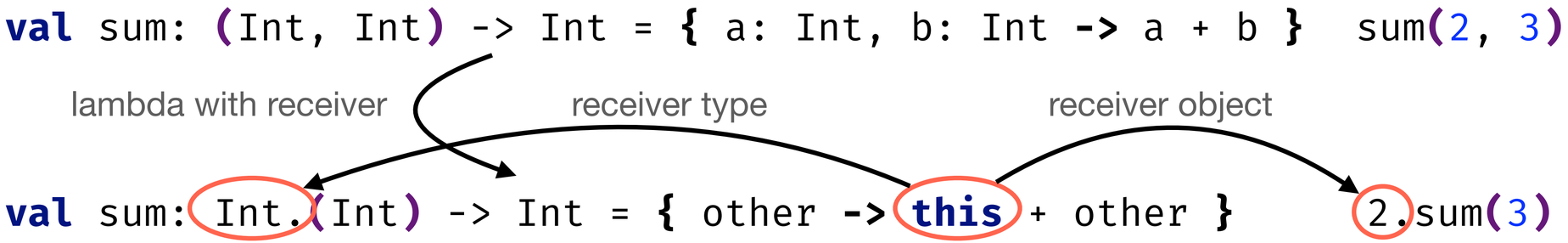
Functional literal with receiver

special form of functional literals



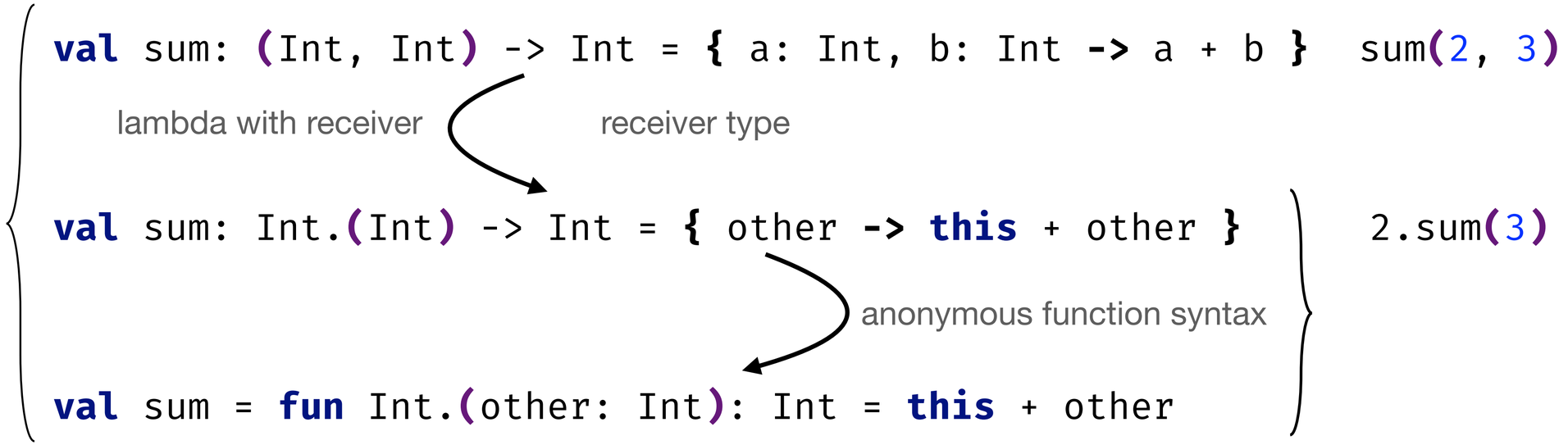
Functional literal with receiver

special form of functional literals



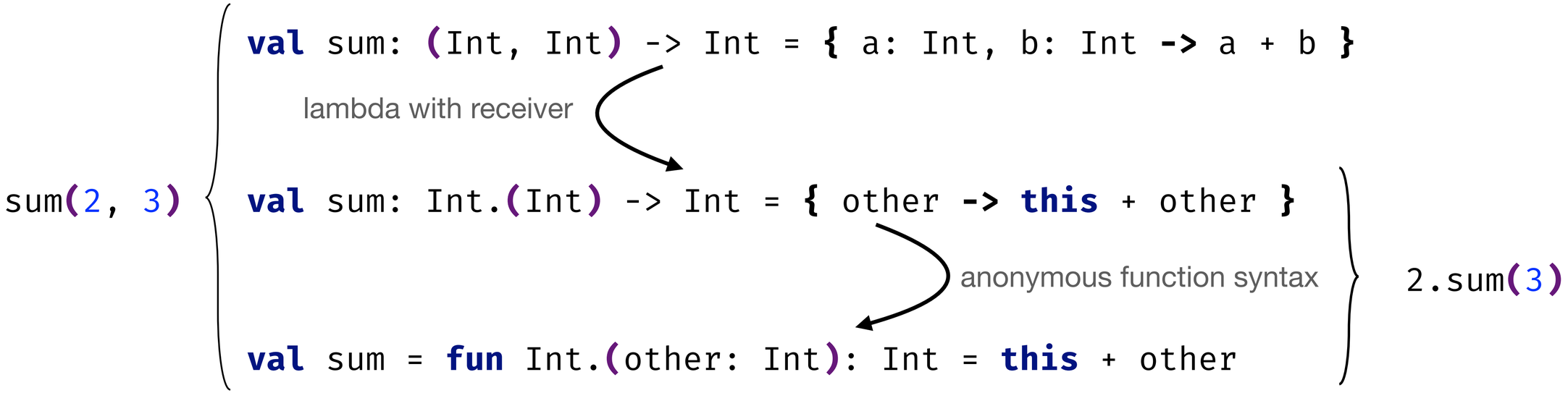
Functional literal with receiver

special form of functional literals



Functional literal with receiver

special form of functional literals



Functional literal with receiver

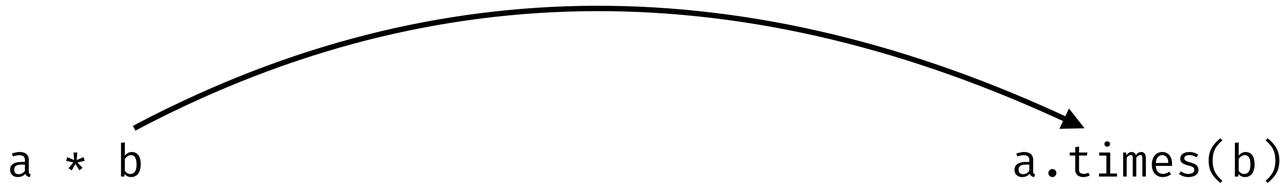
higher order functions

```
fun applyTwice(n: Int, f: (Int) → Int): Int = f(f(n))

applyTwice(2) { it + 1 }
```

```
fun applyTwice(n: Int, f: Int.() -> Int): Int = n.f().f()

applyTwice(2) { this + 1 }
```

- `f` can be seen as a local extension function on the receiver type
- the receiver type is `Int`
- the lambda has no param, the param becomes **`this`**
- the receiver object (`this`) is 2 on the first call and the result of `2.f()` (3) on the second call

Operator overloading




```
class Fraction(val numerator: Int, val denominator: Int) {
    override fun toString(): String = "$numerator/$denominator"
}

operator fun Fraction.times(other: Fraction) =
    Fraction(numerator * other.numerator, denominator * other.denominator)


println(Fraction(2, 3) * Fraction(2, 5)) 4/15
```

Operator overloading

rules

- each operator has a complementary function
- implement the function as a member or extension function
- operator precedence cannot be changed

❗ do not abuse operator overloading - only use it when the behaviour can be deduced intuitively

Operator overloading

overloadable operators

- unary operations

```
+a, -a, !a, a++, a—
```

- binary operations

```
a + b, a - b, a * b
a / b, a % b
a..b, a in b, a !in b
a += b, a -= b, a *= b
a /= b, a %= b
a < b, a > b
a >= b, a <= b
a == b, a != b
```

- invoke operator: `()`

```
a(), a(i), a(i, j),
a(i_1, …, i_n)
```

- indexed access operator: `[]`

```
a[i], a[i_1, …, i_n]
a[i] = b, a[i_1, i_n] = b
```

- property delegation operators *

* will be covered on the ***Delegation*** section of the course

Operator overloading

no restrictions on parameter type

```
operator fun Fraction.times(other: Int) = Fraction(numerator * other, denominator)

operator fun Int.times(other: Fraction) =
    Fraction(this * other.numerator, other.denominator)

println(Fraction(1, 3) * 2)
println(2 * Fraction(1, 3))
```

2/3

2/3

Operator overloading

standard library examples

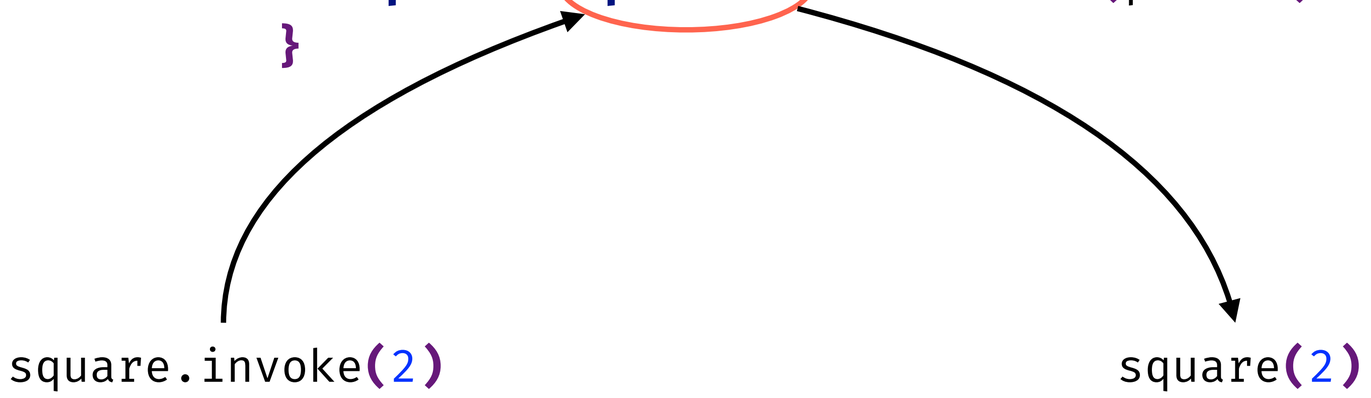
```
public inline operator fun BigDecimal.plus(other: BigDecimal): BigDecimal =
    this.add(other)

val balance = BigDecimal("10.2") + BigDecimal("1.1")
```

Operator overloading

standard library examples

```
val square = { n: Int -> n * n }
```

```
public interface Function1<in P1, out R> : Function<R> {
    /** Invokes the function with the specified argument. */
    public operator fun invoke(p1: P1): R
}
```



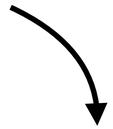
Infix notation

declaration

infix

```
infix fun String.writtenBy(author: String): String = "$this is written by $author"
```

```
"Programming Kotlin".writtenBy("Venkat Subramaniam")
```



```
"Programming Kotlin” writtenBy "Venkat Subramaniam"
```

Scope functions

Execute a block of code within the context of an object

Scope functions

```
val account = Account(
    iban = "RO20INGB1234567812345678",
    product = "Debit",
    currency = "EUR",
    balance = 100.0
)
```

Scope functions

let

```
val updatedAccount = account.let {
    Account(
        iban = it.iban,
        product = it.product,
        currency = it.currency,
        balance = it.balance + 300.0
    )
}
```

- extension function on all types
- the parameter is `account`
- returns the result of evaluating the lambda
- use cases:
  - ☑ execute a lambda on non-null objects
  - ☑ introduce an expression as a variable in the local scope

Scope functions

run

```
val updatedAccount: Account = account.run {
    Account(
        iban = iban,
        product = product,
        currency = currency,
        balance = balance + 300.0
    )
}
```

- extension function on all types
- `this` is `account` - lambda with receiver
- returns the result of evaluating the lambda
- use cases:
  - ☑ object configuration and computing the result
  - ☑ introduce an expression as this in the local scope

Scope functions

run (non-extension)

```
fun generateRandomNumber(): Int {
    val generated = Random.nextInt(100)
    println("I generated $generated")
    return generated
}
```

- returns the result of evaluating the lambda
- use cases:
  - ☑ running statements where an expression is required

```
fun generateRandomNumber(): Int = run {
    val generated = Random.nextInt(100)
    println("I generated $generated")
    generated
}
```

Scope functions

with

```
val updatedAccount: Account = with(account) {
    Account(
        iban = iban,
        product = product,
        currency = currency,
        balance = balance + 300.0
    )
}
```

- `this` is `account`
- returns the result of evaluating the lambda
- use cases:
  - ☑ group function calls or property access on an object

Scope functions

apply

```
class Config(var host: String, var protocol: String = "http", var port: Int = 80)
```

```
val config = Config("google.com").apply {
    protocol = "https"
    port = 443
}
```

- extension function on all types
- `this` is `Config("google.com")`
- returns the original object
- use cases:
  - ☑ object configuration

Scope functions

also

```
class Config(var host: String, var protocol: String = "http", var port: Int = 80)
```

```
val config = Config("google.com").also {
    println("configured for host ${it.host}")
}
```

- extension function on all types
- the parameter is `Config("google.com")`
- returns the original object
- use cases:
  - ☑ additional effects

Scope functions

Any function can be written as an expression using scope functions