Functions and Lambdas

About me

Trayan Iliev

– CEO of **IPT – Intellectual Products & Technologies**
http://www.iproduct.org

– Oracle® certified programmer 15+ Y

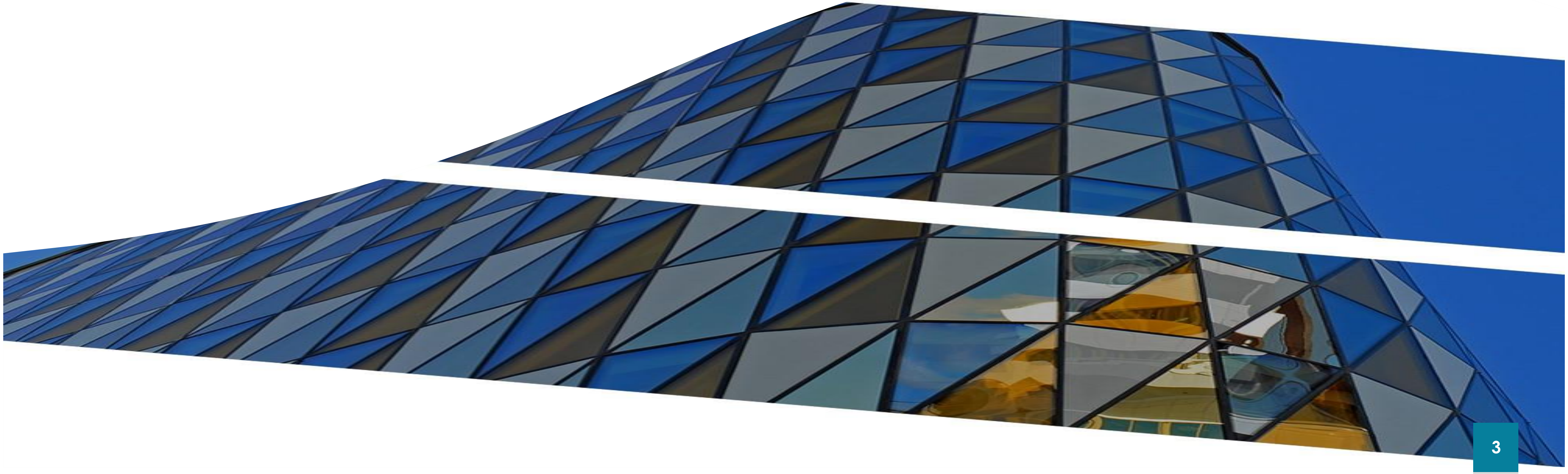
– end-to-end reactive fullstack apps with Java, ES6+, TypeScript, Angular, React and Vue.js

– 12+ years IT trainer: Spring, Java EE, Node.js, Express, GraphQL, SOA, REST, DDD & Reactive Microservices

– Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Organizer RoboLearn hackathons and IoT enthusiast

Functions



Functions

```
fun double(x: Int): Int {
    return 2 * x
}

fun powerOf(number: Int, exponent: Int): Int { /*...*/ return 42}
fun powerOf2(
    number: Int,
    exponent: Int, // trailing comma
) { /*...*/ }


fun main() {
    val result = double(2)
    Stream().read() // create instance of class Stream and call read()
}
```

Functions – default arguments

```
fun read(
    b: ByteArray,
    off: Int = 0,
    len: Int = b.size,
) { /*...*/ }
```

```
open class A {
    open fun foo(i: Int = 10) { /*...*/ }
}
class B : A() {
    override fun foo(i: Int) { /*...*/ }  // No default value is allowed.
}
fun foo(
    bar: Int = 0,
    baz: Int,
) { /*...*/ }
fun main() {
    foo(baz = 1) // The default value bar = 0 is used
}
```

Functions – lambda as last parameter

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


fun main() {
    foo(1) { println("hello") }     // Uses the default value baz = 1
    foo(qux = { println("hello") }) // Uses both default values bar = 0 and baz = 1
    foo { println("hello") }        // Uses both default values bar = 0 and baz = 1
}
```

Named arguments and varargs

- When you use named arguments in a function call, you can freely change the order they are listed in, and if you want to use their default values, you can just leave these arguments out altogether.

```
fun reformat(
    str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    divideByCamelHumps: Boolean = false,
    wordSeparator: Char = ' ',
) { /*...*/ }
reformat("String!", false, upperCaseFirstLetter = false,  divideByCamelHumps = true, '_')
reformat("This is a long String!")
reformat("This is a short String!", upperCaseFirstLetter = false, wordSeparator = '_')

//varargs
fun foo(vararg strings: String) { /*...*/ }
foo(strings = *arrayOf("a", "b", "c"))
```

Generic varargs

- Only one parameter can be marked as vararg. If a vararg parameter is not the last one in the list, values for the subsequent parameters can be passed using named argument syntax, or, if the parameter is function, by passing a lambda outside the parentheses.
- Inside a function, a vararg -parameter of type **T** is visible as an array of **T**, as in the example below, where the ts variable has type Array<out **T**>.

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

- When you call a vararg function, you can pass arguments individually, e.g. asList(1, 2, 3). If you have an array and want to pass its contents, use the spread operator (*):

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
val a2 = intArrayOf(1, 2, 3) // IntArray is a primitive type array
val list2 = asList(-1, 0, *a2.toTypedArray(), 4)
```

Unit returning functions

```kotlin
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello $name")
    else
        println("Hi there!")
    // `return Unit` or `return` is optional
}
```

```kotlin
fun printHello(name: String?) { /*...*/ }
```

Functions returning expressions

- When a function returns a single expression, the curly braces can be omitted and the body is specified after a **=** symbol.
- Explicitly declaring the return type is optional when this can be inferred by the compiler.

```
fun triple(x: Int): Int = x * 3
```

```
fun triple(x: Int) = x * 3
```

Infix notation

Functions marked with the **infix** keyword can also be called using the infix notation. Infix functions must meet the following requirements:

- They must be member functions or extension functions.
- They must have a single parameter.
- The parameter must not accept variable number of arguments and must have no default value.

```
infix fun Int.shl(x: Int): Int { /*...*/ }

// calling the function using the infix notation
    1 shl 2

// is the same as
    1.shl(2)
```

Infix notation

Functions marked with the **infix** keyword can also be called using the infix notation. Infix functions must meet the following requirements:

- They must be member functions or extension functions.
- They must have a single parameter.
- The parameter must not accept variable number of arguments or have default value.

```
infix fun Int.shl(x: Int): Int { /*...*/ }
1 shl 2  // calling the function using the infix notation
1.shl(2) // is the same as

class MyStringCollection {
    infix fun add(s: String) { /*...*/ }
    fun build() {
        this add "abc"   // Correct
        add("abc")       // Correct
        //add "abc"        // Incorrect: the receiver must be specified
    }
}
```

Functions - scope

- Top level functions
- Member functions

```
class Sample {
    fun foo() { print("Foo") }
}
fun main() {
    Sample().foo() // creates instance of class Sample and calls foo
}
```

- Local functions and closures

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)  dfs(v)
    }
    dfs(graph.vertices[0])
}
```

Generic functions

```
fun <T> singletonList(item: T): List<T> { /*...*/ }
```

Tail recursive functions

- Kotlin supports a style of functional programming known as tail recursion. For some algorithms that would normally use loops, you can use a recursive function instead without the risk of stack overflow. When a function is marked with the **tailrec** modifier and meets the required formal conditions, the compiler optimizes out the recursion, to a fast and efficient loop:
- To be eligible for the tailrec modifier, a function must call itself as the last operation it performs. You cannot use tail recursion when there is more code after the recursive call, and you cannot use it within try/ catch/ finally blocks.

```kotlin
val eps = 1E-10 // "good enough", could be 10^-15

tailrec fun findFixPoint(x: Double = 1.0): Double =
    if (Math.abs(x - Math.cos(x)) < eps) x else findFixPoint(Math.cos(x))
```

Higher-order functions (HOF)

- A higher-order function is a function that takes functions as parameters, or returns a function.

```
fun <T, R> Collection<T>.reduce(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

Higher-order functions (HOF) - II

```
fun main() {
    val items = listOf(1, 2, 3, 4, 5)
    // Lambdas are code blocks enclosed in curly braces.
    items.fold(0) {
        // When a lambda has parameters, they go first, followed by '->'
            acc: Int, i: Int ->
                print("acc = $acc, i = $i, ")
                val result = acc + i
                println("result = $result")
                // The last expression in a lambda is considered the return value:
                result
    }
    // Parameter types in a lambda are optional if they can be inferred:
    val joinedToString = items.fold("Elements:", { acc, i -> acc + " " + i })

    // Function references can also be used for higher-order function calls:
    val product = items.fold(1, Int::times)
}
```

Function types

- Kotlin uses function types, such as **(Int) -> String**, for declarations that deal with functions: **val onClick: () -> Unit = ...**
  - Function types: **(A, B) -> C** – denotes a type that represents functions that take two arguments of types A and B and return a value of type C. The list of parameter types may be empty, as in **() -> A**. Unit return cannot be omitted.
  - Function with receiver type: **A.(B) -> C** – represents functions that can be called on a receiver object A with a parameter B and return a value C. Function literals with receiver are often used along with these types.
  - Suspending functions: **suspend () -> Unit** or **suspend A.(B) -> C**
- The function type notation can optionally include names for the function parameters: **(x: Int, y: Int) -> Point**
- Nullable function type: **((Int, Int) -> Int)?**
- Combining function types: **(Int) -> ((Int) -> Unit)**

typealias ClickHandler = (Button, ClickEvent) -> Unit

Instantiating a function type - I

There are several ways to obtain an instance of a function type:

- Use a code block within a function literal, in one of the following forms:
  - a lambda expression: **{ a, b -> a + b }**
  - an anonymous function: **fun(s: String): Int { return s.toIntOrNull() ?: 0 }**
  - Function literals with receiver can be used as values of function types with receiver.
- Use a callable reference to an existing declaration:
  - a top-level, local, member, or extension function: **::isOdd**, **String::toInt**
  - a top-level, member, or extension property: **List<Int>::size**,
  - a constructor: **::Regex**
  - These include bound callable references that point to a member of a particular instance: **foo::toString**.

Instantiating a function type - II

There are several ways to obtain an instance of a function type:

- Using instances of a custom class that implements a function type as an interface:

```
class IntTransformer: (Int) -> Int {
    override operator fun invoke(x: Int): Int = TODO()
}

fun main() {
    val intFunction: (Int) -> Int = IntTransformer()
}
```

- The compiler can infer the function types for variables if there is enough information:

```
val a = { i: Int -> i + 1 } // The inferred type is (Int) -> Int
```

Instantiating function types with receivers

- Non-literal values of function types with and without a receiver are interchangeable, so the receiver can stand in for the first parameter, and vice versa. For instance, a value of type **(A, B) -> C** can be passed or assigned where a value of type **A.(B) -> C** is expected, and the other way around:

```
val repeatFun: String.(Int) -> String = { times -> this.repeat(times) }
val twoParameters: (String, Int) -> String = repeatFun // OK

fun runTransformation(f: (String, Int) -> String): String {
    return f("hello", 3)
}
val result = runTransformation(repeatFun) // OK
```

- A function type with no receiver is inferred by default, even if a variable is initialized with a reference to an extension function. To alter that, specify the variable type explicitly.

Invoking a function type instance

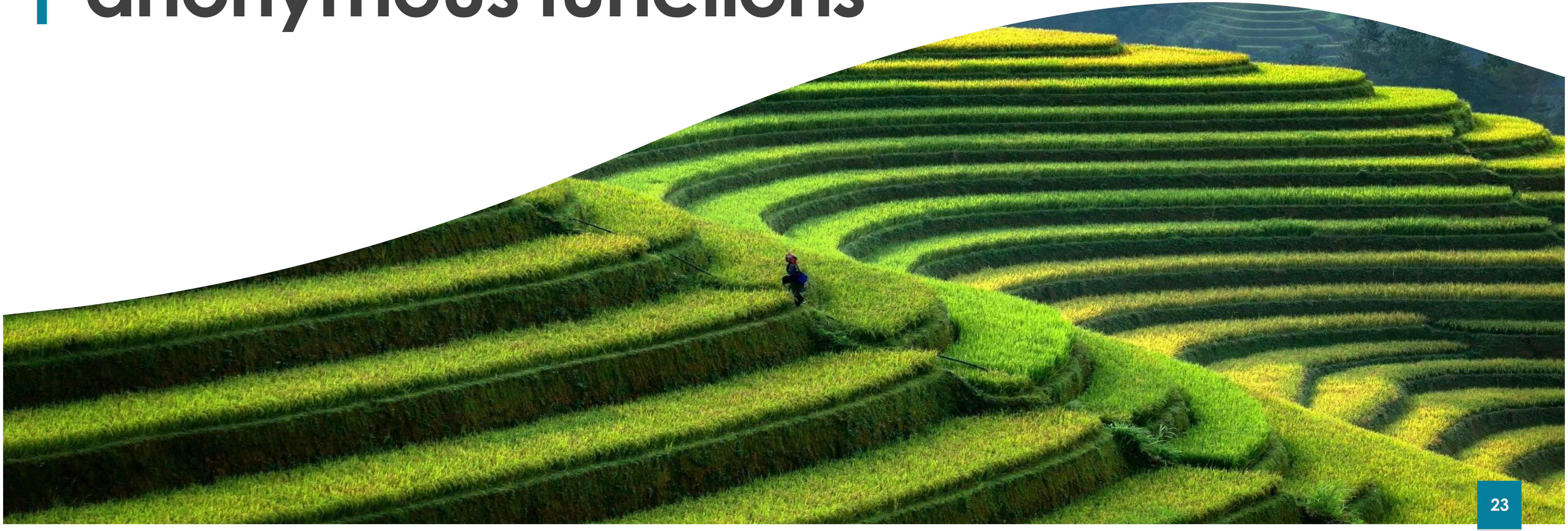
- A value of a function type can be invoked by using its **invoke(...)** operator: **f.invoke(x)** or just **f(x)**.
- If the value has a receiver type, the receiver object should be passed as the first argument. Another way to invoke a value of a function type with receiver is to prepend it with the receiver object, as if the value were an extension function: **1.foo(2)**

```
val stringPlus: (String, String) -> String = String::plus
val intPlus: Int.(Int) -> Int = Int::plus

println(stringPlus.invoke("<-", "->"))
println(stringPlus("Hello, ", "world!"))

println(intPlus.invoke(1, 1))
println(intPlus(1, 2))
println(2.intPlus(3)) // extension-like call
```

Lambda expressions and anonymous functions



Lambda expressions

- Lambda expressions and anonymous functions are function literals. Function literals are functions that are not declared but are passed immediately as an expression. Consider the following example:

```
val strings = listOf("Orange", "Banana", "Pineapple", "Papaya", "Apple", "Plum")
println(max(strings, { a, b -> a.length - b.length }))
```

- The function max is a higher-order function, as it takes a function value as its second argument. This second argument is an expression that is itself a function, called a function literal, which is equivalent to the following named function:

```
fun compare(a: String, b: String): Int = a.length - b.length
```

Lambda expressions syntax

- Lambda expressions full syntax:

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

- 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 the **->**.
- 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.

- If you leave all the optional annotations out, what's left looks like this:

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

Passing trailing lambdas

- According to Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses:

```
val product = items.fold(1) { acc, e -> acc * e }
```

- Such syntax is also known as **trailing lambda**.
- If the lambda is the only argument in that call, the parentheses can be omitted entirely:

```
run { println("...") }
```

it: implicit name of a single parameter

- It's very common for a lambda expression to have only one parameter.

- If the compiler can parse the signature without any parameters, the parameter does not need to be declared and **->** can be omitted. The parameter will be implicitly declared under the name **it**:

```
val ints = listOf(1, 2, 3)
ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```

Returning a value from a lambda expression

```
ints.filter {
    val shouldFilter = it > 0
    shouldFilter
}

ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}

strings.filter { it.length == 5 }.sortedBy { it }.map { it.uppercase() }
```

Underscore for unused variables

```
val map = mapOf(1 to "x", 2 to "y", 3 to "z")

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

```
x!
y!
z!
```

Destructuring in lambdas

- You can use the destructuring declarations syntax for lambda parameters. If a lambda has a parameter of the **Pair** type (or **Map.Entry**, or any other type that has the appropriate **componentN** functions), you can introduce several new parameters instead of one by putting them in parentheses:

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

- { a -> ... } // one parameter
- { a, b -> ... } // two parameters
- { (a, b) -> ... } // a destructured pair
- { (a, b), c -> ... } // a destructured pair and another parameter

```
map.mapValues { (_, value) -> "$value!" }
map.mapValues { (_, value): Map.Entry<Int, String> -> "$value!" }
map.mapValues { (_, value: String) -> "$value!" }
```

Anonymous functions

```
fun(x: Int, y: Int): Int = x + y
```

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

```
ints.filter(fun(item) = item > 0)
```

Function literals with receiver: A.(B) -> C

- As mentioned above, Kotlin provides the ability to call an instance of a function type with receiver while providing the receiver object.
- Inside the body of the function literal, the receiver object passed to a call becomes an implicit this, so that you can access the members of that receiver object without any additional qualifiers, or access the receiver object using a this expression.
- This behavior is similar to that of extension functions, which also allow you to access the members of the receiver object inside the function body.
- Here is an example of a function literal with receiver along with its type, where plus is called on the receiver object:

```
val sum2: Int.(Int) -> Int = { other -> plus(other) }
val sum3 = fun Int.(other: Int): Int = this + other
```

Function literals with receiver: A.(B) -> C

- As mentioned above, Kotlin provides the ability to call an instance of a function type with receiver while providing the receiver object.
- Inside the body of the function literal, the receiver object passed to a call becomes an implicit this, so that you can access the members of that receiver object without any additional qualifiers, or access the receiver object using a this expression.
- This behavior is similar to that of extension functions, which also allow you to access the members of the receiver object inside the function body.
- Here is an example of a function literal with receiver along with its type, where plus is called on the receiver object:

- val sum: Int.(Int) -> Int = { other -> plus(other) }map.*mapValues* **{** entry **->** "${entry.value}!" **}**
  map.*mapValues* **{** **(**key, value**)** **->** "$value!" **}**
- { a -> … } // one parameter

Type-safe builders

```
class HTML {
    fun body() { /*...*/ }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()  // create the receiver object
    html.init()        // pass the receiver object to the lambda
    return html
}

fun main() {
    html {       // lambda with receiver begins here
        body()   // calling a method on the receiver object
    }
}
```

Typesafe HTML Builder Exercise

https://github.com/iproduct/course-kotlin/blob/master/07-functions-lambdas/src/main/kotlin/html-builder.kt

Learn Kotlin by Example & Kotlin idioms

https://play.kotlinlang.org/byExample/

https://kotlinlang.org/docs/idioms.html

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD