



Functions and Lambdas

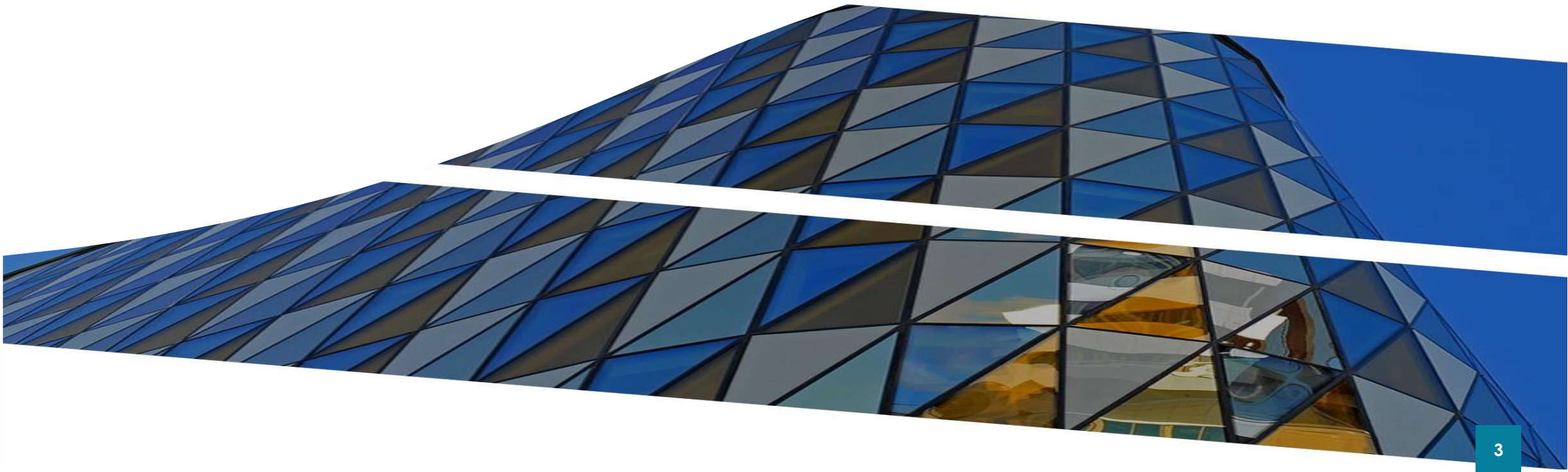
About me



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

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



```
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.

```
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**

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>