СПб АУ НОЦНТ РАН

Kotlin 02

30.10.2017

Анонимые функции

```
fun foo(block: (Int) -> String) = block(123)

foo(fun(x: Int) { return x.toString() })
foo({ x: Int -> x.toString() })
foo({ x -> x.toString() })
foo({ it.toString() })
foo { it.toString() }
foo(Int::toString)
```

Extension function

```
fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()
```

Member extensions

```
class A {
    fun String.spaceToCamelCase() {
        this.length // this@spaceToCamelCase.length
        this@A.foo()
    }

    fun foo() {
        "Convert this to camelcase".spaceToCamelCase()
    }
}
```

Extension properties

```
val String.numberOfA get() = count { it == 'A' }
val String.abc = 1 // error
```

Generic extensions

```
public fun <T> Iterable<T>.filter(p: (T) -> Boolean): List<T>

public fun <T, R> T.let(block: (T) -> R): R = block(this)

fun foo() = a.nullableMethod()?.let { it.length + it.length * 2 }
```

Members vs. Extensions

```
val String.length get() = 1
println("".length) // ??
```

Members vs. Extensions

```
list.forEach { e -> println(e) }
```

Operator conventions

```
data class Point(val x: Int, val y: Int)

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

val point = Point(10, 20)
println(-point)  // prints "(-10, -20)"
```

Operator conventions

```
data class Point(val x: Int, val y: Int)

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

val point = Point(10, 20)
println(point.unaryMinus())
```

Operator conventions

- unaryMinus/unaryPlus/not
- plus/minus/div/rem/times
- inc/dec/plusAssign/*Assign
- set/get – arraylike access
- compareTo
- https://kotlinlang.org/docs/reference/operator-overloading.html

in/in!

- a in b // a.contains(b)
- a !in b // !a.contains(b)

Равенство

- a === b // ‘a == b‘ в Java
- a == b // a?.equals(b) ?: (b === null)
- a != b // !(a?.equals(b) ?: (b === null))

Компоненты

```
val (x, y) = a

val x = a.component1()
val y = a.component2()

for ((x, y) in ...) {}

data class A(val x: Int, val y: String) // has components
```

invoke convention

```
operator fun String.invoke(x: Int) {
    // calling function by name through reflection
}

"abc"(1) // "abc".invoke(1)
```

Инфиксные функции

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

val pair = 1 to ""
```

Использование

```
val ml: MutableList<String>
ml += 1
(ml + ml).size
"abc" in ml
ml[1] = ml[0] + ""

val (first, second) = ml

for ((k, v) in mapOf("1" to 2)) {
    println("$k $v")
}
```

Delegated properties

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

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

Delegated properties

```
class A {
    val lazyValue: String by d
}

operator fun getValue(
    thisRef: R,
    property: KProperty<*>
): Type
// Each call to a.lazyValue compiled as
d.getValue(this, A::lazyValue /* KProperty (reflection)*/)
```

Delegated properties

```
operator fun Map<String, Int>.getValue(
        x: Any?,
        kProperty: KProperty<*>
): Int = this[kProperty.name]!!

val map = mapOf("abcd" to 123)
val abcd: Int by map
// The same as
//val abcd: Int
//    get() {
//        return map.getValue(null, ::abcd)
//    }

println(abcd) // prints "123"
```

Inline функции

```
inline fun <T> Iterable<T>.filter(
    predicate: (T) -> Boolean
): List<T> {
    val destination = arrayListOf<T>()
    for (element in this) {
        if (predicate(element)) {
            destination.add(element)
        }
    }
    return destination
}

fun foo(x: List<Int>) {
    // works just as if you had a loop here
    println(x.filter { it > 0 })
}
```

Возврат из лямбды

```
fun foo(x: List<Int>) {
    x.forEach {
        // exit from lambda (like continue)
        if (it == 42) return@filter
        // non-local return from foo (works only for inline lambda)
        if (it == 56) return
        it > 0
    }
}
```

Reified type parameters

```
inline fun <reified T> Iterable<Any?>.filterIsInstance(): List<T> {
    val destination = arrayListOf<T>()
    for (element in this) {
        if (element is T) {
            destination.add(element)
        }
    }
    return destination
}

fun foo(x: List<Any?>) {
    println(x.filterIsInstance<String>().size)
}
```

Dependency injection

```
interface Container {
    fun <T : Any> getInstance(klass: KClass<T>): T
}

inline operator fun <reified T : Any> Container.getValue(
        thisRef: Any?, kProperty: KProperty<*>
): T = getInstance(T::class)

fun doSomething(c: Container) {
    val myComp: MyComponent by c

    // c.getInstance(MyComp::class).hashCode()
    myComp.hashCode()
}
```

Coding conventions

- Про имена и базовое форматирование все примерно также как в Java
  - Фигурные скобки у if'а не пишутся, когда это выражение
- Предпочтения кратким формам записи (если это не вредит читаемости)
  - Первичные конструкторы
  - Функции из одного выражения
  - Вывод типов, где это очевидно
  - Использование функций стандартной библиотеки
  - Многое другое
- Обращаем внимание на подсказки IDE

Long class header

```
class Person(
    val id: Int,
    val name: String,
    val surname: String
) : Human(id, name),
    KotlinMaker {

    // ...
}
```

Long function header

```
fun longMethodName(
    argument: ArgumentType = defaultValue,
    argument2: AnotherArgumentType
): ReturnType {
    // body
}
```

Полезные шаблоны

```
fun parseBody(): Result? {
    val parser = buildParser() ?: return null

    // fast return
    parser.parseShortBody()?.let { return it }

    return parser.parseLongBody() ?: error("Can't parse")
}
```

Safe casts

```
// throws CCE when outerValue is not Int
val x: Int = outerValue as Int

// returns null when outerValue is not Int
val x: Int? = outerValue as? Int
```

Локальные функции

```
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])
}
```

Домашнее задание

- Ветка “02-fun-interpreter”
- Количество баллов будет зависеть от наличия предупреждений в IDE и соблюдения coding conventions