

АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 5

ПЛАН

- Параметры функций
- Наследование
- Операторы

ИМЕНОВАННЫЕ ПАРАМЕТРЫ

- Бывает так, что у функции есть несколько параметров одного типа
- И их порядок неочевиден
- Соглашений о порядке не сложилось
- Легко перепутать

ИМЕНОВАННЫЕ ПАРАМЕТРЫ

- Например:

```
createUser("username", "id-123")
```

VS

```
createUser("id-123", "username")
```

- Или:

```
new User("username", "id-123")
```

VS

```
new User("id-123", "username")
```

ИМЕНОВАННЫЕ ПАРАМЕТРЫ

- В Java для конструкторов проблема решается через шаблон "Builder"
- Для методов - примерно никак

JAVA: BUILDER

```
1 User user = User.builder()  
2           .user("username")  
3           .id("id-123")  
4           .build();
```

BUILDER

- Builder сам себя не напишет
- Либо однотипный boilerplate писать
- Либо библиотеки на аннотациях - Lombok и т.п.

ЧТО ДАЕТ KOTLIN

- Именованные параметры
- Любой параметр можно обозначить по имени в точке вызова
- Если смесь именованных и неименованных, то сначала неименованные
- В именованной части порядок произвольный

ПРИМЕР

```
1 fun createUser(name: String, id: String) {  
2     println("$name, $id")  
3 }  
4  
5 fun main() {  
6     createUser("vasya", "id-1234")  
7     createUser("id-1234", "vasya")  
8     createUser(name="vasya", "id-1234")  
9     createUser("vasya", id="id-1234")  
10    createUser(name="vasya", id="id-1234")  
11    createUser(id="id-1234", name="vasya")  
12    //createUser(id="id-1234", "vasya")  
13    //createUser("id-1234", name="vasya")  
14 }
```

ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

- В Java перегрузка часто используется для параметров по умолчанию
- Пишется самый подробный вариант
- И много сокращенных - которые вызывают самый подробный вариант
- Писать так на Kotlin - плохой стиль

ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

- Пример: `createUrl`
- Можно создавать из компонентов, многие из которых имеют значения по умолчанию
- А можно - разрешать путь относительно базового URL
- Для первого случая - параметры по умолчанию
- Для второго - перезагрузка

ПРИМЕР

```
1 fun createUrl(protocol: String="https", host: String,  
2               port: Int=443, path: String="",  
3               query: String=""): URL {  
4     return URL(  
5         protocol, host, port,  
6         if (query.isEmpty()) path else "$path?$query"  
7     )  
8 }  
9  
10 fun createUrl(base: URL, path: String): URL {  
11     return URL(base, path)  
12 }
```

VARARG

- Определили свою структуру данных
- Типа коллекции
- Хотим инициализировать в общем Kotlin-стиле
- Как-то так:

```
binomialHeapOf("hello", "vasya")
```

VARARG

- Параметр можно пометить как множественный: `vararg`
- Добавим ключевое слово перед параметром
- Он примет в себя переменное число значений
- Они будут видны как `Array` обычных типов
- Или `IntArray/LongArray` и т.п. - для примитивных

ПРИМЕР

```
1 fun main() {  
2     println(uniqueOf("qqq", "asd", "qqq", "12345"))  
3 }  
4  
5 fun uniqueOf(vararg words: String): List<String> {  
6     val found = mutableSetOf<String>()  
7     return words.filter {  
8         val result = it !in found  
9         found.add(it)  
10        result  
11    }  
12 }
```

НЮАНСЫ

- `vararg` может встречаться только 1 раз в определении
- Даже если разные типы - нельзя
- Можно сочетать с другими параметрами, не `vararg`
- Но те, которые после `vararg` - надо указывать по имени
- Или использовать значения по умолчанию

НЮАНСЫ

- В точке вызова `vararg` жаден
- Если идут значения его типа - заберет все
- Если идет значение не его типа - ни себе, ни людям
- Будет ошибка компиляции

ПРИМЕР

```
1 fun main() {  
2     //f()      - понятно, что так нельзя  
3     //f("a")  - но и так - тоже  
4     //f("a", "abc") - и так  
5     f("a", v="abc")  
6 }  
7  
8 fun f(vararg data: String, v: String) = v in data
```

НЮАНСЫ

- Константы/переменные вставляются прямым перечислением
- Массив можно вставить через *
- Не любую коллекцию, только массив
- Можно несколько раз

ПРИМЕР

```
1 uniqueOf(*arrayOf("123", "234"))  
2 uniqueOf(*arrayOf("vasya", ""), "123", *arrayOf("1234"))  
3 // uniqueOf(*listOf("1234")) - нельзя
```

КОНСТРУКТОРЫ

- В целом - как функции и методы
- С особенностями синтаксиса
- Начнем с первичного конструктора
- Официальный вариант синтаксиса предполагает слово `constructor`

ПРИМЕР

```
1 class C private constructor(val v: Int) {  
2     companion object {  
3         fun createInstance(): C {  
4             // ... - логика билдера  
5             return C(10)  
6         }  
7         fun createInstance(s: String): C {  
8             // ... - логика билдера  
9             return C(s.toInt())  
10        }  
11    }  
12 }  
13  
14 // .....
```

ПРИМЕР

```
1 // .....  
2  
3 fun main() {  
4     C.createInstance()  
5     C.createInstance("111")  
6     //C(5)  
7 }
```

КОНСТРУКТОРЫ

- В теле класса можно определить вторичные конструкторы
- `constructor` вместо `fun` и без имени
- И делегирование к другому конструктору
- Не в коде, а через отдельный синтаксис

ПРИМЕР

```
1 data class Point(val a: Int, val b: Int) {  
2     constructor(p: Pair<Int, Int>):  
3         this(p.first, p.second)  
4 }  
5  
6 fun main() {  
7     println(Point(5, 6))  
8     println(Point(Pair(1, 2)))  
9 }
```

КОНСТРУКТОРЫ

- Первичный конструктор - особый случай
- Если не нужны модификаторы, 'constructor' можно опустить
- Вместо this - отсылка на конструктор суперкласса
- Если не по умолчанию

ОБЯЗАТЕЛЬНОСТЬ ДЕЛЕГИРОВАНИЯ

- В Java можно написать несколько независимых конструкторов
- Каждый из которых сам инициализирует поля
- И каждый сам отдельно отвечает за инициализацию `final`-полей
- Можно делегировать другому, если это удобно
- Но это один из вариантов

ОБЯЗАТЕЛЬНОСТЬ ДЕЛЕГИРОВАНИЯ

- Это может иметь значение
- По JVM "заморозка" `final` происходит при выходе из конструктора
- Если он вызывается через делегирование - после выхода из вложенного вызова
- Кто делегировал - `final`-поля подстроить уже не сможет

ПРИМЕР

```
1  class C {  
2      private final int v;  
3  
4      C() {  
5          // ....  
6          // common initialization  
7          // ...  
8          v = 0;  
9      }  
10 // .....
```

ПРИМЕР

```
1 // .....
2
3     C(int v) {
4         this.v = v;
5     }
6
7     C(int v1, int v2) {
8         this();
9         // this.v = v1 + v2;    - уже нельзя
10    }
11 }
```

ОБЯЗАТЕЛЬНОСТЬ ДЕЛЕГИРОВАНИЯ

- Kotlin принуждает к делегированию
- Не обязательно делегироваться к первичному непосредственно
- Но цепочка делегирований обязана привести к нему

ОБЯЗАТЕЛЬНОСТЬ ДЕЛЕГИРОВАНИЯ

- А в теле вторичного конструктора - уже инициализированный объект
- Можно какой-то код исполнить между вызовом вторичного конструктора
- И того, кому он делегирует
- Но там много ограничений и неудобств

ОБЯЗАТЕЛЬНОСТЬ ДЕЛЕГИРОВАНИЯ

- Потому что в этот момент над `this` не вызван конструктор суперкласса
- А JVM к этому моменту относится особенно трепетно
- Нельзя вызвать даже свои приватные методы
- И передать такой `this` компаньону - тоже

ОБЩИЙ ПОДХОД

- Первичный конструктор - максимально подробный
- Если часто полные подробности не нужны, то используем параметры по умолчанию
- А вторичные конструкторы - когда нужна другая сигнатура, не сводящаяся к первичному через параметры по умолчанию

ВАРИАЦИЯ

- Можем решить, что в первичном совсем мелкие детали
- И приложению незачем о них знать - даже потенциально
- Тогда первичные делаем приватным
- И заводит несколько вторичных, которые будут декораторами приватного первичного
- Максимально используем значения по умолчанию

НЕ ВСЕГДА ЭТО ПОМОЖЕТ

- Вот создали класс URL
- И придумали кучу полей на самый детальный вариант
- Через параметры по умолчанию и вторичные можно дать кучу удобных вариантов сокращенного вызова
- Но еще хочется создавать относительный URL по контексту и относительному пути

НЕ ВСЕГДА ЭТО ПОМОЖЕТ

- Мы не можем просто надергать property
- Потому что потому конфигурация их изменений зависит от второго параметра
- В этом случае помогает companion
- В нем определить factory-метод
- Там сделать предвычисления - и в конце вызвать конструктор

ПРИМЕР

```
1 data class Point(val a: Double, val b: Double) {
2     constructor(p: Pair>Double, Double>):
3         this(p.first, p.second)
4 }
5 data class Line(val a: Double, val b: Double,
6                 val c: Double=0.0) {
7     companion object {
8         fun horizontal(v: Double) = Line(0.0, v, -v)
9         fun vertical(v: Double) = Line(v, 0.0, -v)
10        fun simple(a: Double, b: Double): Line {
11            TODO()
12        }
13    }
14 }
```

ПЕРЕОПРЕДЕЛЕНИЕ ОПЕРАТОРОВ

- Определили свой тип
- Хотим выражать операции над ним как операции
- А не как вызовы методов
- Например, определили класс Rational
- Хотим выражать через '+'

OPERATOR

- Определяем метод класса Rational
- Называем его plus
- Используем ключевое слово operator
- `operator fun plus(that: Rational)`

OPERATOR

- Набор операторов - фиксированный
- Если хочется чего-то своего - только через infix
- И оператор будет буквенным
- Например для создания дроби: 3 over 4

СООТВЕТСТВИЕ

$a + b$	<code>a.plus(b)</code>
---------	------------------------

$a - b$	<code>a.minus(b)</code>
---------	-------------------------

$a * b$	<code>a.times(b)</code>
---------	-------------------------

a / b	<code>a.div(b)</code>
---------	-----------------------

$a \% b$	<code>a.rem(b)</code>
----------	-----------------------

СООТВЕТСТВИЕ

a .. b	a.rangeTo(b)
--------	--------------

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

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

СООТВЕТСТВИЕ

$a[i]$

$a.get(i)$

$a[i, j]$

$a.get(i, j)$

$a[i_1, i_2, \dots, i_k]$

$a.get(i_1, i_2, \dots, i_k)$

$a[i] = b$

$a.set(i, b)$

$a[i, j] = b$

$a.set(i, j, b)$

$a[i_1, i_2, \dots, i_k] = b$

$a.set(i_1, i_2, \dots, i_k, b)$

СООТВЕТСТВИЕ

$a()$	$a.\text{invoke}(i)$
-------	----------------------

$a(i)$	$a.\text{invoke}(i)$
--------	----------------------

$a(i, j)$	$a.\text{invoke}(i, j)$
-----------	-------------------------

$a(i_1, i_2, \dots, i_k)$	$a.\text{invoke}(i_1, i_2, \dots, i_k)$
---------------------------	---

ООТВЕТСТВИЕ

$a += b$	<code>a.plusAssign(b)</code>
----------	------------------------------

$a -= b$	<code>a.minusAssign(b)</code>
----------	-------------------------------

$a *= b$	<code>a.timesAssign(b)</code>
----------	-------------------------------

$a /= b$	<code>a.divAssign(b)</code>
----------	-----------------------------

$a \% = b$	<code>a.remAssign(b)</code>
------------	-----------------------------

СООТВЕТСТВИЕ

$a == b$	$a?.equals(b) \text{ ? : } (b === \text{null})$
----------	---

$a != b$	$!(a?.equals(b) \text{ ? : } (b === \text{null}))$
----------	--

$a > b$	$a.compareTo(b) > 0$
---------	----------------------

$a < b$	$a.compareTo(b) < 0$
---------	----------------------

$a \geq b$	$a.compareTo(b) \geq 0$
------------	-------------------------

$a \leq b$	$a.compareTo(b) \leq 0$
------------	-------------------------

СООТВЕТСТВИЕ

+a	a.unaryPlus()
----	---------------

-a	a.unaryMinus()
----	----------------

!	a.not(b) > 0
---	--------------

a++	a.inc()
-----	---------

b++	a.dec()
-----	---------

ИНТЕРФЕЙСЫ И НАСЛЕДОВАНИЕ

- В целом напоминает Java
- Вместо extends - двоеточие
- Вместо implements - тоже
- override - ключевое слово

ПРИМЕР

```
1 interface Runnable {  
2     fun run()  
3 }  
4  
5 class Printer(private val s: String): Runnable {  
6     override fun run() {  
7         println(s)  
8     }  
9 }
```

ПЕРЕКРЫТИЕ СИГНАТУР

- Само по себе - не проблема
- Особенно если без реализации по умолчанию
- И без значений параметров по умолчанию

