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:

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

vs

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

ИМЕНОВАННЫЕ ПАРАМЕТРЫ

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

JAVA: BUILDER

```
User user = User.builder()
                .user("username")
                .id("id-123")
                .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 компонетов, многие из которых имеют значения по умолчанию
- А можно - разрешать путь относительно базового URL
- Для первого случая - параметры по умолчанию
- Для второго - перегрузка

ПРИМЕР

```kotlin
fun createUrl(protocol: String="https", host: String,
              port: Int=443, path: String="",
              query: String=""): URL {
    return URL(
        protocol, host, port,
        if (query.isEmpty()) path else "$path?$query"
    )
}

fun createUrl(base: URL, path: String): URL {
    return URL(base, path)
}
```

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

```kotlin
fun main() {
    println(uniqueOf("qqq", "asd", "qqq", "12345"))
}

fun uniqueOf(vararg words: String): List<String> {
    val found = mutableSetOf<String>()
    return words.filter {
        val result = it !in found
        found.add(it)
        result
    }
}
```

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

```
class C private constructor(val v: Int) {
    companion object {
        fun createInstance(): C {
            // ... – логика билдера
            return C(10)
        }
        fun createInstance(s: String): C {
            // ... – логика билдера
            return C(s.toInt())
        }
    }
}

// ..........
```

ПРИМЕР

```
// ............

fun main() {
    C.createInstance()
    C.createInstance("111")
    //C(5)
}
```

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

```
// ..............

    C(int v) {
        this.v = v;
    }

    C(int v1, int v2) {
        this();
        // this.v = v1 + v2;  – уже нельзя
    }
}
```

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 момень над this не вызван конструктор суперкласса
- А JVM/Kotlin к этому моменту относится особенно трепетно
- Нельзя вызвать даже свои приватные методы
- И передать такой this компаньону - тоже

ЧЕРЕЗ SUPER

- Можно из вторичного вызвать конструктор суперкласса
- Но тогда нельзя полноценно использовать первичный
- Сначала исполнится конструктор суперкласса
- Потом - вырожденный первичный, и потом вторичный со связанными руками

ОБЩИЙ ПОДХОД

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

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	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)

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[i1, i2, ..., ik]	a.get(i1, i2, ..., ik)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i1, i2, ..., ik] = b	a.set(i1, i2, ..., ik, b)

СООТВЕТСТВИЕ

a()	a.invoke(i)
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i1, i2, ..., ik)	a.invoke(i1, i2, ..., ik)

СООТВЕТСТВИЕ

a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b)

СООТВЕТСТВИЕ

a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a ≥ b	a.compareTo(b) ≥ 0
a ≤ b	a.compareTo(b) ≤ 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

```
interface Runnable {
    fun run()
}

class Printer(private val s: String): Runnable {
    override fun run() {
        println(s)
    }
}
```

ПЕРЕКРЫТИЕ СИГНАТУР

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