АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 3

ПЛАН ЛЕКЦИИ

- Управляющие конструкции
- Классы: продолжение

WHILE - ЦИКЛЫ

- Классический while и do-while
- Фигурные скобки в обоих можно опускать
- Официально выражениями не являются
- Но местами ведут себя как выражения со значением типа Unit
- В условии do-while можно использовать локальную переменную из блока цикла

ПРИМЕРЫ

```
1 var curr = Pair(0, 1)
2 var i = 0
3 while (i++ < 10) {
4     println(curr.first)
5     curr = Pair(curr.second, curr.first + curr.second)
6 }
7
8 i = 0
9 while (i < 10) println(i++)
```

ПРИМЕРЫ

```
 1 var curr = Pair(0, 1)
 2 do {
 3     println(curr.first)
 4     curr = Pair(curr.second, curr.first + curr.second)
 5 } while (curr.first < 100)
 6
 7 var i = 0
 8 do println("hello") while (i++ < 20)
 9
10 do {
11     val v = Math.random()
12 } while (v < 0.5)
13
14 // no v variable here
```

КАК ВЫРАЖЕНИЯ

```
 1 fun value() = 5
 2 val f1: () -> Int = {
 3     value()
 4 }
 5 val f2: () -> Unit = {
 6     value()
 7     println()
 8 }
 9 val f3: () -> Unit = {
10     value()
11     while (Math.random() < 0.5) println()
12 }
13 // val v: Unit = while (Math.random() < 0.5)
14 //     println() // не выйдет
```

FOR-ЦИКЛЫ

- ```
  for (v in data) { println(v) }
  ```
- По любому итератору
- Можно без фигурных скобок
- Классический for делается через range
  ```

ПРИМЕРЫ

```
 1 for (i in 0 ..< 10) {
 2     println(i)
 3 }
 4
 5 for (i in 0 .. 10) println(i) // включая 10
 6
 7 for (v in setOf("hello", "world")) {
 8     println(v)
 9 }
10
11 for (c in "hello") {
12     println(c)
13 }
```

ПРИМЕРЫ

```
1 // ужас, никаких оправданий
2 for (i in 0 ..< args.size) {
3     println(args[i])
4 }
5
6 // Правильно
7 for (v in args) {
8     println(args)
9 }
```

ПРИМЕРЫ

```
1 // есть оправдание, но все равно ужас
2 for (i in 0 until args.size) {
3     println("$i: ${args[i]}")
4 }
5
6 for ((i, v) in args.withIndex()) {
7     println("$i: ${args[i]}")
8 }
```

BREAK/CONTINUE

- Как везде с маленькой добавкой
- Можно помечать цикл:

```
label@ while ...
```

- Тогда во вложенном можно явно указать, к кому относится break/continue
-

```
break@label
```

IF/ELSE

- Общая форма и семантика - как везде
- Нет сокращенного elif - для этого есть when
- Без else в чем-то похож на циклы
- В смысле - формально выражением не является
- Местами фактически может пониматься как возвращающий Unit

IF/ELSE КАК ВЫРАЖЕНИЕ

- С else трактуется как выражение
- Типичная форма:

```
fun abs(v: Int) = if (v >= 0) v else -v
```

- Замена тернарного оператора
- Но блоки могут быть с фигурными скобками
- И большими по содержимому

IF/ELSE КАК ВЫРАЖЕНИЕ

- Значением ветки является значение последнего выражения
- Или Unit, если последнее предложение - не выражение
- Типом будет наиболее точный супертип типов результатов веток

ПРИМЕР

```
1 fun prettyView(s: String, size: Int) =
2     if (size >= s.length) s else {
3         var result: String = "TODO"
4         // tricky calculations
5         result
6     }
```

ПРИМЕР

```
1 fun p(v: Any): Any {
2     println(v)
3     return v
4 }
5 fun isPrintable(v: Any) = v is String
6
7 // Хочется, но нельзя
8 fun printPossible(v: Any) = if (isPrintable(v)) p(v)
```

ПРИМЕР

```
1  fun p(v: Any): Any {
2      println(v)
3      return v
4  }
5  fun isPrintable(v: Any) = v is String
6
7  // Можно, но меняется возвращаемый тип
8  fun printPossible(v: Any) { if (isPrintable(v)) p(v) }
9
10 // Я б согласился, но тоже нельзя
11 fun printPossible(v: Any): Any {
12     if (isPrintable(v)) p(v)
13 }
```

ПРИМЕР

```
1 // И так нельзя
2 fun printPossible(v: Any): Any {
3     if (isPrintable(v)) return p(v)
4 }
5
6 // И так тоже
7 fun printPossible(v: Any): Any {
8     if (isPrintable(v)) return p(v)
9 }
```

ПРИМЕР

```
1 // Можно, но громоздко
2 fun printExceptHello(v: Any): Any {
3     if (isPrintable(v)) return p(v)
4     return Unit
5 }
```

ПРИМЕР

```
1 // Можно вот так
2 fun printPossible(v: Int) = if (v > 0) p(v) else Unit
3
4 // Или так
5 fun printPossible(v: Int): Any = if (v > 0) p(v) else Unit
```

ЗАБАВНОЕ

```
1 val v: Int = 5
2 val f: () -> Int = { 5 }
3
4 val r1: Int = if (Math.random() < 0.5) v else v
5 val r2: Int = if (Math.random() < 0.5) 5 else 5
6 val r3: () -> Int = if (Math.random() < 0.5) f else f
7 val r4: Int = if (Math.random() < 0.5) { 5 } else { 5 }
```

ЗАБАВНОЕ

- В Kotlin нет конструкции "блок, возвращающий значение"
- Чтобы как-то так:

```
val v: Int = {println("assign"); 5}
```

- Можно только так:

```
val v: Int = {println("assign"); 5}()
```

- Но это выльется в создание анонимной функции

WHEN

- Kotlin-версия case
- По-особому подсахаренная
- Можно указать выражение и расписать варианты значений
- А можно без общего выражения перебирать разношерстные условия (в духе elif)

ПРИМЕР

```
1 fun nDays(month: Int) = when (month) {
2     2 -> 28
3     4, 6, 9, 11 -> 31
4     1, 2, 5, 7, 8, 10, 12 -> 31
5     else -> -1
6 }
```

WHEN И BREAK

- В when не нужен никакой break
- Опасно, когда когда when внутри цикла
- "По инерции" написанный break может оказаться формально корректным
- И выйти из цикла, если when в цикле

ПРИМЕР

```
 1 fun printNDays(months: IntArray) {
 2     for (month in months) {
 3         when (month) {
 4             2 -> {
 5                 println(28)
 6                 break
 7             }
 8             4, 6, 9, 11 ->  {
 9                 println(30)
10                 break
11             }
12
13 // to be contunued
```

ПРИМЕР

```
// ....
            1, 2, 5, 7, 8, 10, 12 ->   {
                println(30)
                break
            }
        }
    }
}

fun main() {
    printNDays(intArrayOf(2, 10))
}
```

ЕЩЕ ДЕТАЛИ

- Сравнивать не обязательно с константой
- Можно удобно проверять на in, !in, is, !is
- when может быть полноценным выражением
- Надо, чтобы не было необработанных вариантов
- И это было статически понятно (else, enum)

ПРИМЕР

```
when (x) {
    s.toInt() -> print("s encodes x")
    else -> print("s does not encode x")
}

when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

ПРИМЕР

```
when {
    x.isOdd() -> print("x is odd")
    y.isEven() -> print("y is even")
    else -> print("x is even, y is odd")
}

fun Request.getBody() =
    when (val response = executeRequest()) {
        is Success -> response.body
        is HttpError ->
            throw HttpException(response.status)
    }
```

SMART CAST

- Частый шаблон в Java: проверить instanceof, привести тип
- Kotlin уничтожает этот boilerplate
- Часто можно не писать явного преобразования
- И компилятор начинает воспринимать объект с уточненным типом

SMART CAST

- Не работает, если есть противоречие в возможных вариантах
- Или его разрешение не по силам компилятору
- Не работает над нелокальными var-ами
- Не работает над свойствами со своим get()
- Не работает везде, где нет гарантий "атомарности"

КЛАССЫ

- Бывает так, что параметры конструктора нужны только для инициализации
- И они не нужны во время жизни объекта
- Можно убрать val при их объявлении
- Например, при определении рационального числа

ПРИМЕР

```
 1 private fun gcd(a:Int, b: Int): Int =
 2     if (b == 0) a else gcd(b, a % b)
 3
 4 class Rational(a: Int, b: Int) {
 5       val num: Int
 6       val denom: Int
 7
 8       init {
 9           val gcd = gcd(a, b)
10           num = a / gcd
11           denom = b / gcd
12       }
13 }
```

СВОЙСТВА С ПОЛЯМИ И БЕЗ ПОЛЕЙ

- val с начальным значением подразумевает наличие final-поля
- Аналогично - val из конструктора
- Доступ к нему - через JVM-метод
- А еще можно сделать свой get-метод
- Тут есть варианты

СВОЙСТВА С ПОЛЯМИ И БЕЗ ПОЛЕЙ

- Иногда хочется иметь реальное "физическое" поле
- И что-то делать при доступе к нему
- Например, при включенном отладочном режиме собрать статистику чтений
- А иногда - вернуть что-то, зависящее от имеющихся свойств
- Не храня это поле в памяти

ПРИМЕР С ПОЛЕМ

```
 1 class Point(x_: Double, y_: Double) {
 2     private val xCounter = AtomicLong()
 3     private val yCounter = AtomicLong()
 4
 5     val x: Double = x_
 6         get() {
 7             xCounter.incrementAndGet()
 8             return field
 9         }
10 // ......
```

ПРИМЕР С ПОЛЕМ

```
1  // .......
2
3      val y: Double = y_
4          get() {
5              yCounter.incrementAndGet()
6              return field
7          }
8
9      fun stats() = Pair(xCounter, yCounter)
10 }
```

ПРИМЕР БЕЗ ПОЛЯ

```
1 class Line(val a: Point, val b: Point) {
2     val length: Double
3         get() {
4             val dx = a.x - b.x
5             val dy = a.y - b.y
6             return sqrt(dx * dx + dy * dy)
7         }
8 }
```

ОТЛИЧИЯ

- В варианте с полем обращаемся к нему через 'field'
- Без поля - field не упоминается
- В варианте с полем обязана быть инициализация
- Без поля - нельзя инициализировать

FIELD

- Обращение в get к полю field - признак наличия поля
- Бывает, что в классе есть свойство с именем field
- И хочется к нему обратиться в get-методе
- Тогда надо через this.field

РЕКУРСИЯ

- Не запрещено в get-методе обратиться к "себе" через имя свойства
- Но это будет рекурсия
- Вряд ли разумная и так и задуманная
- Никто не помешает получить косвенную рекурсию
- Или бесконечную цепочку вызовов

ПРИМЕР РЕКУРСИИ

```
 1 class C1 {
 2     val p: Int
 3         get() = C2().p
 4 }
 5
 6 class C2 {
 7     val p: Int = 5
 8         get(): Int {
 9             println(field)
10             return C1().p
11         }
12 }
```

DATA-КЛАССЫ

- Часто нужны классы, чтобы просто вместе держать несколько полей
- И обычные Kotlin-классы во многом упрощают создание таких классов
- Но хочется чуть большего
- Например, разумной реализации hashCode/equals
- toString тоже неплохо бы

DATA-КЛАССЫ

- А еще хочется уметь копировать
- Создать новый объект из старого, поменяв пару полей
- Чтобы не руками и не через аннотации
- Все будет, если написать data class

ПРИМЕР

```kotlin
class PersonOrdinary(val name: String, val age: Int)

data class PersonData(val name: String, val age: Int)

fun main(args: Array<String>) {
    val po1 = PersonOrdinary("vasya", 23)
    val po2 = PersonOrdinary("vasya", 23)
    val po3 = PersonOrdinary("petya", 25)

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

ПРИМЕР

```
1 // .........
2
3     val pd1 = PersonData("vasya", 23)
4     val pd2 = PersonData("vasya", 23)
5     val pd3 = pd2.copy(name="petya")
6
7     println(po3)
8     println(pd3)
9     println(po1 == po2)
10    println(pd1 == pd2)
11 }
```

КРАТКО ПРО СТАТИКУ

- В Kotlin-классах нет "статических полей"
- И статических классов - тоже
- Для статических сущностей есть три формы существования
- Простейшая - внеклассовые функции и свойства

СВОЙСТВА УРОВНЯ ФАЙЛА

- val/var могут существовать на уровне файла
- Это аналог статических полей
- Можно реализовать свой get/set
- И это может быть свойство с полем и без поля

CONST-СВОЙСТВА

- Можно перед val указать const
- Это аналог static final
- Применимо только в статическом контексте
- Никаких своих get

CONST-СВОЙСТВА

- Инициализируется только статически известными значениями
- Можно позволить себе выражения, даже конкатенацию
- Аргументами могут быть другие const val

ПРИМЕР

```
1  const val FLAG_ADVANCED_MODE = 0x0001
2  const val FLAG_PENDING = 0x0002
3  const val STATE_FLAGS =
4      FLAG_ADVANCED_MODE or FLAG_PENDING
5
6  fun printDetails(v: Int) {
7      if (v and FLAG_ADVANCED_MODE != 0) {
8          println("ADVANCED")
9      }
10     if (v and FLAG_PENDING != 0) {
11         println("PENDING")
12     }
13     println("state flags: ${STATE_FLAGS and v}")
14 }
```

ОРГАНИЗАЦИЯ В РАМКАХ ПРОЕКТА

- В Kotlin есть понятие package
- Но нет требования, чтобы структура исходников соответствовала структуре пакетов
- Классы могут жить где угодно
- Важно, чтобы компилятор обнаружил файл с классом
- И конфликтов не возникло

ОРГАНИЗАЦИЯ В РАМКАХ ПРОЕКТА

- В начале файла обычно указывает package
- Можно не указывать, но это рекомендуется
- IDEA мягко подталкивает
- Можно в разных файлах указать один и тот же package
- И эти файлы могут жить в разных каталогах

ОРГАНИЗАЦИЯ В РАМКАХ ПРОЕКТА

- Все, что определяется в разных файлах одного package-а, "видит" друг друга
- И разделяет пространства имен
- Статику таких файлов можно представлять как виртуальный Util-класс, определенный для пакета

ОРГАНИЗАЦИЯ В РАМКАХ ПРОЕКТА

- Если пакет не указан, то это специальный пакет "умолчанию"
- Из других пакетов его элементы должны быть явно импортированы