АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 4

ПЛАН ЛЕКЦИИ

- Управляющие конструкции
- Строки
- Функции

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

```
var curr = Pair(0, 1)
var i = 0
while (i++ < 10) {
    println(curr.first)
    curr = Pair(curr.second, curr.first + curr.second)
}

i = 0
while (i < 10) println(i++)
```

ПРИМЕРЫ

```
var curr = Pair(0, 1)
do {
    println(curr.first)
    curr = Pair(curr.second, curr.first + curr.second)
} while (curr.first < 100)

var i = 0
do println("hello") while (i++ < 20)

do {
    val v = Math.random()
} while (v < 0.5)

// no v variable here
```

КАК ВЫРАЖЕНИЯ

```
fun value() = 5
val f1: () -> Int = {
    value()
}
val f2: () -> Unit = {
    value()
    println()
}
val f3: () -> Unit = {
    value()
    while (Math.random() < 0.5) println()
}
// val v: Unit = while (Math.random() < 0.5)
//    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 вложненном 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

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

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

```
fun printNDays(months: IntArray) {
    for (month in months) {
        when (month) {
            2 -> {
                println(28)
                break
            }
            4, 6, 9, 11 ->  {
                println(30)
                break
            }

// 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"

СТРОКОВЫЕ ЛИТЕРАЛЫ

- В двойных кавычах - "как в Java"
  - Стандартные \-последовательности
- В трех двойных кавычках - много на несколько строк
  - \ ничего не значит

ПРИМЕР "ДЛИННОЙ" СТРОКИ

```
fun p() {
    val s = """
        Однажды в студеную зимнюю пору
        Я из лесу вышел, был сильный мороз
        Гляжу – поднимается медленно в гору
        Лошадка, везущая хворосту воз
    """

    println(s)
}
```

ЧТО ПОЛУЧИМ

- Переводы строки сохранятся
  - Отступы - тоже
- Синтаксического способа убрать их - нет

  Но есть полезные методы

ВАРИАНТ 1

```
fun p() {
    val s = """
        Однажды в студеную зимнюю пору
        Я из лесу вышел, был сильный мороз
        Гляжу – поднимается медленно в гору
        Лошадка, везущая хворосту воз
    """.trimIndent()

    println(s)
}
```

ВАРИАНТ 2

```
fun p() {
    val s = """
    |    Однажды в студеную зимнюю пору
    |    Я из лесу вышел, был сильный мороз
    |    Гляжу – поднимается медленно в гору
    |    Лошадка, везущая хворосту воз
    """.trimMargin()

    println(s)
}
```

ВАРИАНТ 3

```
fun p() {
    val s = """
    ***    Однажды в студеную зимнюю пору
    ***    Я из лесу вышел, был сильный мороз
    ***    Гляжу – поднимается медленно в гору
    ***    Лошадка, везущая хворосту воз
    """.trimMargin("***")

    println(s)
}
```

ОПЕРАЦИИ НАД СТРОКАМИ

- Символ по индексу через квадратные скобки
- И не только для строки - для любого CharSequence
- Что верно для всех расширений строки

ОПЕРАЦИИ НАД СТРОКАМИ

- Есть бинарная операция a in b
- Сводится к b.contains(a)
- Есть ее отрицание a !in b
- Это идиоматичнее, чем not(a in b)

РАСШИРЕНИЯ

- Добавлены тонны новых методов
- Общая идея - унифицировать строку
- Считать ее разновидностью коллекции
- Невозможно про все рассказать
- Что-то я расскажу, остальное ищем в документации

РАСШИРЕНИЯ

- Местами добавлены варианты с типами в духе Kotlin-а
- Например, substring(IntRange)
- Надо стремиться использовать методы в функциональном стиле
- Худший вариант - делать что-то через while

РАСШИРЕНИЯ

- Чуть лучше - пройти for-ом по индексам
- И реализовать логику в теле цикла
- Лучший вариант - свести к функциональным методам
- Или их цепочке

ВОЗМОЖНЫ ВАРИАНТЫ

- Если сложная логика перебора
- Если "неправилыный" вариант сильно быстрее правильного - и это важно
- Важно не нарваться на квадратичную сложность

ПРИМЕР

```
1  fun countDuplicates(s: String) = s.zipWithNext()
2      .count { pair -> pair.first == pair.second }
3
4  fun countDuplicateVowels(s: String) {
5      val vowels = "aeiouy".toSet()
6      s.zipWithNext()
7          .filter { pair -> pair.first == pair.second }
8          .map { pair -> pair.first }
9          .count(vowels::contains)
10 }
```

ПЛОХОЙ ПРИМЕР

```
1 fun naiveReverse(s: String): String = if (s.isNotEmpty())
2     naiveReverse(s.drop(1)) + s.first()
3     else s
4
5 // Метод reversed уже есть,
6 // делегируется к StringBuilder.reverse
7
8 // Если бы нужно было – можно через
9 // свертку с состоянием в StringBuilder
```

ИНТЕРПОЛЯЦИЯ

- Если в строке есть переменная часть, пишем ${}
- И в фигурных скобках - переменное выражение
- Если выражение - одна переменная, можно без фигурных скобок
- В фигурных скобках могут быть свои фигурные скобки
- И даже вложенная интерполяция

ПРИМЕР ВЛОЖЕННОЙ ИНТЕРПОЛЯЦИИ

```
1 fun bigrams(s: String): Map<String, Int> = s
2     .zipWithNext()
3     .map { String(charArrayOf(it.first, it.second)) }
4     .groupBy { it }
5     .mapValues { it.value.count() }
6
7 fun main(args: Array<String>) {
8     println("${args[0]}: ${bigrams("#${args[0]}#")}")
9 }
```

StringBuilder и StringBuffer

- Два класса динамических строк в Java
- StringBuffer - синхронизированный
- StringBuilder - несинхронизированный (более быстрый)
- Оба доступны, но StringBuilder предпочтительнее

STRINGBUILDER И STRINGBUFFER

- StringBuilder определен в kotlin.text
- В StringBuilder определен метод set - как расширение
- Можно писать sb[5] = 'a'
- В StringBuffer - нет

ОПРЕДЕЛЕНИЕ ФУНКЦИИ

- Функция может определяться вне класса
- Функция может определяться в классе - тогда это Kotlin-метод
- Функция может определяться внутри функции
- И даже внутри блока

СХЕМА ОПРЕДЕЛЕНИЯ ФУНКЦИИ

- Ключевое слово fun
- Имя функции
- Список параметров в скобках
- А дальше - варианты

ВАРИАНТ "КОРОТКОЙ" ФУНКЦИИ

- Необязательно так буквально
- Функция состоит из одного выражения
- Тогда ставим знак равенства и пишем выражение
- Но оно может быть длинным (if, when, ...)

ВЫВЕДЕНИЕ ВОЗВРАЩАЕМОГО ТИПА

- В короткой функции отсутствие типа приводит к его выведению
- При рекурсии выведение типов не работает
- В публичных методах выведение типов сильно нежелательно
- А функции по умолчанию превращаются в публичные методы

"ДЛИННЫЕ" ФУНКЦИИ

- После параметров - возвращаемый тип
- И тело функции в фигурных скобках
- Возвращение - через явный return
- Если возвращаем Unit, можно просто дойти до конца

"ДЛИННЫЕ" ФУНКЦИИ

- Выведения типов нет
- Тип может быть не указан
- Но это означает тип Unit
- Указывать Unit явно можно, но не принято

АНОНИМНЫЕ ФУНКЦИИ

- Могут быть без параметров
- Тогда это просто фигурные скобки с предложениями
- Если в конце выражение, то его тип будет возвращаемым типом анонимной функции
- Если в конце не выражение, то анонимная функция возвращает Unit

ПРИМЕРЫ ОПРЕДЕЛЕНИЙ

```
val f1: () -> Int = { 5 }
val f2: () -> Unit = {  }
val f3: () -> Unit = {
    println("hello")
}
val f4: () -> Unit = {
    if (Math.random() < 0.5) println("hello")
}
val f5: () -> Double = Math::random
val f6: () -> String = ::produceString
// если produceString определена как "не-метод"
```

ВЫЗОВ АНОНИМНОЙ ФУНКЦИИ

- Через скобки: f1()
- Через invoke: f1.invoke()
- Под капотом каждая функция - анонимный класс с методом invoke (интерфейс Function0)
- Разницы между прямым вызовом и через invoke нет

ПРИМЕНЕНИЕ АНОНИМНОЙ ФУНКЦИИ БЕЗ АРГУМЕНТОВ

- Ленивый параметр функции
- Например, есть схема логирования
- Есть уровни логирования
- trace - самый подробный

ПРИМЕНЕНИЕ АНОНИМНОЙ ФУНКЦИИ БЕЗ АРГУМЕНТОВ

- Другие уровни: info, error, warn, debug
- Желательный уровень выставляется в конфигурации
- Возможно, для каждого класса свой
- По коду расставлены вызовы:

```
logger.info("value of variable: " + variable)
```

ПРИМЕНЕНИЕ АНОНИМНОЙ ФУНКЦИИ БЕЗ АРГУМЕНТОВ

- Вызовов log.trace очень много
- Уровень trace включаем, когда нужна ну совсем подробная информация
- Для отладки чего-то особо непонятного
- Казалось бы - поставили уровень ниже, и trace-логи не мешают

ПРИМЕНЕНИЕ АНОНИМНОЙ ФУНКЦИИ БЕЗ АРГУМЕНТОВ

- Но мы вынуждены считать строковые аргументы
- Которые в итоге не нужны
- А их вычисление может быть существенным
- Выход - передать анонимную функцию без параметров

КОВАРНЫЙ СЛУЧАЙ

```
fun bigrams(s: String) = {
    s.zipWithNext()
        .map { String(charArrayOf(it.first, it.second)) }
        .groupBy { it }
        .mapValues { it.value.count() }
}

fun main(args: Array<String>) {
    println(bigrams(args[0]))
}
```

ЧТО ПРОИСХОДИТ

- Программа компилируется и работает
- Но печатает странное
- Функция bigram написана неправильно
- Но получается синтаксически корректная конструкция
- А выведенные типы не выявляют проблемы
- Потому что контекст применения типа очень свободный

С ОДНИМ ПАРАМЕТРОМ

- После фигурной скобки указываем имя параметра и тип
- Потом "стрелка" и тело функции

```
val incr = { v -> v + 1 }
```

- Такие функции часто передаются параметрами в функции обработки коллекций

БЕЗ ОБЪЯВЛЕНИЯ ПАРАМЕТРА

- Можно:

```
val f: (Int) -> Int = {it * 2}
```

- Нельзя:

```
val f = {it * 2}
```

- Можно:

```
val f = { it: Int -> it * 2 }
```

БЕЗ ОБЪЯВЛЕНИЯ ПАРАМЕТРА

- Если передаем параметром, то контекст понятен, то есть можно коротко
- Идиома: определять последним параметром, при вызове помещать за скобками
- Синтаксис разрешает, code-style рекомендует

ПРИМЕР

```kotlin
fun countDuplicates(s: String) = s.zipWithNext()
    .count { it.first == it.second }

fun countDuplicateVowels(s: String) {
    val vowels = "aeiouy".toSet()
    s.zipWithNext()
        .filter { it.first == it.second }
        .map { it.first }
        .count(vowels::contains)
}
```

МНОГО ПАРАМЕТРОВ

```
intArrayOf(123, 234)
    .fold(Pair(Int.MIN_VALUE, Int.MAX_VALUE)) {acc, curr ->
        Pair(acc.first.coerceAtLeast(curr),
             acc.second.coerceAtMost(curr)
        )
    }

val sum = {a: Int, b: Int -> a + b}
```