АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 2

ПЛАН ЛЕКЦИИ

- Строки
- Основы определения функций
- Классы

СТРОКОВЫЕ ЛИТЕРАЛЫ

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

ПРИМЕР "ДЛИННОЙ" СТРОКИ

```
 1 fun p() {
 2     val s = """
 3         Однажды в студеную зимнюю пору
 4         Я из лесу вышел, был сильный мороз
 5         Гляжу - поднимается медленно в гору
 6         Лошадка, везущая хворосту воз
 7     """
 8
 9     println(s)
10 }
```

ЧТО ПОЛУЧИМ

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

  Но есть полезные методы

ВАРИАНТ 1

```
 1 fun p() {
 2     val s = """
 3         Однажды в студеную зимнюю пору
 4         Я из лесу вышел, был сильный мороз
 5         Гляжу - поднимается медленно в гору
 6         Лошадка, везущая хворосту воз
 7     """.trimIndent()
 8
 9     println(s)
10 }
```

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

```
 1 fun p() {
 2     val s = """
 3     ***    Однажды в студеную зимнюю пору
 4     ***    Я из лесу вышел, был сильный мороз
 5     ***    Гляжу - поднимается медленно в гору
 6     ***    Лошадка, везущая хворосту воз
 7     """.trimMargin("***")
 8
 9     println(s)
10 }
```

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

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

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

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 возращает Unit

ПРИМЕРЫ ОПРЕДЕЛЕНИЙ

```
 1 val f1: () -> Int = { 5 }
 2 val f2: () -> Unit = {  }
 3 val f3: () -> Unit = {
 4     println("hello")
 5 }
 6 val f4: () -> Unit = {
 7     if (Math.random() < 0.5) println("hello")
 8 }
 9 val f5: () -> Double = Math::random
10 val f6: () -> String = ::produceString
11 // 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

```
 1 fun bigrams(s: String) = {
 2     s.zipWithNext()
 3         .map { String(charArrayOf(it.first, it.second)) }
 4         .groupBy { it }
 5         .mapValues { it.value.count() }
 6 }
 7
 8 fun main(args: Array<String>) {
 9     println(bigrams(args[0]))
10 }
```

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

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

МНОГО ПАРАМЕТРОВ

```
1 intArrayOf(123, 234)
2     .fold(Pair(Int.MIN_VALUE, Int.MAX_VALUE)) {acc, curr ->
3         Pair(acc.first.coerceAtLeast(curr),
4              acc.second.coerceAtMost(curr)
5         )
6     }
7
8 val sum = {a: Int, b: Int -> a + b}
```

КЛАССЫ

- Простейшее определение:

```
class C
```

- Чуть посложнее:

```
class C {}
```

- Еще посложнее:

```
class C() {}
```

- И еще:

```
class D(val value: Int)
```

ЭКЗЕМПЛЯРЫ КЛАССА

- Нет слова new
- ```
  val c = C()
  ```
- ```
  val d = D(10)
  ```
- Очень просто получить значение:

  ```
  d.value
  ```

ПОЙМЕМ, ЧТО ПРОИСХОДИТ

- Сразу описываем публичный конструктор
- И сразу создаем поле с именем и типом, равным параметру конструктора
- Kotlin порождает конструктор, который копирует параметр в поле
- И порождает get-метод
- А поле будет final в JVM

МЕТОДЫ

- Создадим класс Binom
- Хотим знать разложение на множители
- Определим метод
- Все - как для функций, только внутри класс

ПРИМЕР

```
 1 class Binom(val a: Double, val b: Double, val c: Double) {
 2     fun roots(): Pair<Double, Double> {
 3         val d = sqrt(b * b - 4 * a * c)
 4         val q = 2 * a
 5         return Pair((-b + d) / q, (-b - d) / q)
 6     }
 7 }
 8
 9 fun main(args: Array<String>) {
10     println(Binom(1.0, 2.0, 1.0).roots())
11 }
```

БОЛЕЕ ИДИОМАТИЧНО

```
 1 class Binom(val a: Double, val b: Double, val c: Double) {
 2       val roots: Pair<Double, Double>
 3           get() {
 4               val d = sqrt(b * b - 4 * a * c)
 5               val q = 2 * a
 6               return Pair((-b + d) / q, (-b - d) / q)
 7           }
 8 }
 9
10 fun main(args: Array<String>) {
11     println(Binom(1.0, 2.0, 1.0).roots)
12 }
```

ЕЩЕ ВАРИАНТ

```
1 class Binom(val a: Double, val b: Double, val c: Double) {
2     val roots: Pair<Double, Double>
3     init {
4         val d = sqrt(b * b - 4 * a * c)
5         val q = 2 * a
6         roots =  Pair((-b + d) / q, (-b - d) / q)
7     }
8 }
```

ДРУГОЙ КЛАСС

```
1 class Rectange(val a: Double, val b: Double) {
2     val diag: Double = sqrt(a * a + b * b)
3 }
```

ЛОГИКА КОНСТРУКТОРА

- Логика порожденного контруктора формируется из кусочков
- Сначала - инициализация полей, отмеченных как val в списке параметров
- А дальше идут кусочки, соответствующие val-объявлениям из тела класса
- И init-блокам
- В том порядке, как они идут

В ТЕРМИНАХ KOTLIN

- Тело класса "исполняется" при создании объекта
- Выражения при val-инициализациях и init-блоки
- Есть еще разные вариации
- О них - в следующий раз