АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 2

ПЛАН ЛЕКЦИИ

- ООП в Kotlin

КРУПНЫМ ПЛАНОМ

- Борьба с boilerplate-ом
- Поддержка некоторых паттернов в языке
- Ограничения на конструкторы
- Переосмысление статики
- Унаследованные от JVM проблемы

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
- Как функция, только внутри класса

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 }
```

В ПОИСКАХ СОВЕРШЕНСТВА

```
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

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

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 конструктора 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
- Нюанс: нет конструкции "блок, возвращающий значение"

ПРОСТО ПАРАМЕТРЫ КОНСТРУКТОРА

- 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

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

    val y: Double = y_
        get() {
            yCounter.incrementAndGet()
            return field
        }

    fun stats() = Pair(xCounter, yCounter)
}
```

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

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

    val pd1 = PersonData("vasya", 23)
    val pd2 = PersonData("vasya", 23)
    val pd3 = pd2.copy(name="petya")

    println(po3)
    println(pd3)
    println(po1 == po2)
    println(pd1 == pd2)
}
```

КРАТКО ПРО СТАТИКУ

- В Kotlin-классах нет "статических полей"
- И статических классов - тоже
- Для статических сущностей есть разные 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

ВЕРНЕМСЯ К СТАТИКЕ

- Один вариант - внеклассовые свойства и функции
- Типичное применение: вспомогательные функции уровня пакета и выше
- Константы времени компиляции - его разновидность
- Другая разновидность - вспомогательные методы для "внешних" классов

МОТИВИРУЮЩИЙ ПРИМЕР

- Пишем GUI-приложение
- Дошло до использования библиотеки AWT
- Базовое GUI для JVM-платформы
- API разработано в прошлом веке

ПРИМЕР

```
1 val point = Point(100, 200)
2 println(point.x)
3 println(point.y)
4 point.move(10, 20)
5 println(point)
```

РАЗБЕРЕМ

- Что не устраивает
  - move - изменяющий метод
  - А хочется создавать новый объект
  - И в ООП стиле

ПРИМЕР: НЕ ООП

```
1  fun moved(p: Point, dx: Int, dy: Int) =
2      Point(p.x + dx, p.y + dy)
3
4  fun scaled(p: Point, c: Int) =
5      Point(p.x * c, p.y * c)
6
7      val newPoint =
8          moved(
9              scaled(
10                 moved(point, 10, 20),
11                 4),
12             3, 4
13         )
```

АЛЬТЕРНАТИВА

- По-честному "встроиться" в чужой класс нельзя
- Но можно поддержать синктаксически
- Можно к имени функции приписать имя класса - слева через точку
- И потом вызывать этот дополнительный "метод"

ПРИМЕР: ООП

```
1  fun Point.moved(dx: Int, dy: Int) =
2      Point(this.x + dx, this.y + dy)
3
4  fun Point.scaled(c: Int) =
5      Point(this.x * c, this.y * c)
6
7  fun main() {
8      val point = Point()
9
10     point.moved(10, 20)
11         .scaled(3)
12         .moved(3, 4)
13 }
```

КАК ЭТО РАБОТАЕТ

- Создается статический метод
- В месте определения расширения
- Тип первого параметра - расширяемый класс
- Остальные параметры - из расширения

ВАЖНОЕ СЛЕДСТВИЕ

- Нет полноценного полиморфизма
- Мы можем класс и подкласс расширить одним методом
- С тем же именем и теми же параметрами
- Но это не будет "перекрытием"

ВАЖНОЕ СЛЕДСТВИЕ

- Внутри расширения нет слова super
- И выбор между методами делается статически
- И это опасная ошибка

ПРИМЕР

```
1  open class C1
2  class C2() : C1()
3
4  fun C1.m() = println("C1.m")
5  fun C2.m() {
6      println("C2.m")
7  }
8  fun f(c1: C1) = c1.m()
9
10 fun main() {
11     f(C1())
12     f(C2())
13 }
```

ЧТО ЕЩЕ СО СТАТИКОЙ

- Вспомогательная логика для обычного класса
- Не привязанная к экземплярам
- И, возможно, что спрятанная от других объектов