АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 6

ПЛАН

- Интерфейсы
- Наследование
- Права доступа
- Замыкания

РЕАЛИЗАЦИЯ ПО УМОЛЧАНИЮ

- Примерно как в Java
- В контексте синтаксиса Kotlin - как метод

```
1 interface URLFactory {
2     fun newURL(): URL = URL("http://www.ya.ru")
3 }
```

СТАТИКА И ИНТЕФЕЙСЫ

- В том же духе, как и в классах
- Статических методов в прямом смысле нет
- Но можно завести компаньона

ПРИМЕР

```
interface Iface {
    companion object {
        fun f() = 5
        const val FLAG = 0x123
    }
}
object Use {
    val v1 = Iface.FLAG
    val v2 = Iface.f()
}
```

ВАРИАНТ ИСПОЛЬЗОВАНИЯ

- Интерфейс описывает контракт
- Есть несколько реализаций
- В компаньон помещаем factory-метод
- Который по параметрам определяет подходящую реализацию

СВОЙСТВА В ИНТЕРФЕЙСЕ

- В интерфейсе могут встречаться свойства
- Но их нельзя материализовывать
- Но можно определить get-метод
- Будет как метод по умолчанию

ПРИМЕР

```
interface Base {
    val url: String
}

class C: Base {
    override val url: String = "https://www.yandex.ru"
}
```

ПЛОХОЙ ПРИМЕР

```
interface Base {
    val url: String = "https://www.yandex.ru"
    // так нельзя
}

interface Base {
    val url: String
        get() = field.substring(1) // и так тоже
}
```

ХОРОШИЙ ПРИМЕР

```
interface Base {
    val url: String
        get() = "https://www.yandex.ru"
        // А так – можно
}


class C1: Base {
    override val url = ""
}

class C2: Base {
    init {
        println(url)
    }
```

НЮАНСЫ

- В определении метода по умолчанию могут быть параметры со значением по умолчанию
- Более того - у метода без тела тоже может быть параметр по умолчанию
- В месте определения параметр надо указать
- Но свое значение по умолчанию указать нельзя

ПРИМЕР

```
interface Base {
    fun createUrl(host: String="www.yandex.ru"): String =
        "https://${host}"
}

object O : Base

fun main() {
    println(O.createUrl())
}
```

ПРИМЕР

```
interface Base {
    fun createUrl(host: String="www.yandex.ru"): String =
        "https://${host}"
}
object O : Base {
    override fun createUrl(host: String) = "ftp://${host}"
}
fun main() {
    println(O.createUrl())
}
```

РЕАЛИЗАЦИЯ

- Форме без параметра соответствует синтетический статический интерфейсный метод
- Ему передается параметром объект
- Он вызывает над объектом его реализацию метода
- И передает значение по умолчанию

ДЕТАЛИЗАЦИЯ

- Можно обратиться к конкретному методу по умолчанию
- Указываем super и тип в угловых скобках
- Помогает в разрешении неоднозначностей
- Или в вызове "скрытого" метода

ПРИМЕР

```
interface I1 {
    fun m() = println("I - 1")
}
interface I2 {
    fun m() = println("I - 2")
}
class C: I1, I2 {
    override fun m() {
        super<I1>.m()
        super<I2>.m()
    }
}
```

НА ПОНИМАНИЕ

```kotlin
interface I1 {
    fun m(v: Int = 10) = println("I - 1: $v")
}
interface I2 : I1 {
    override fun m(v: Int) = println("I - 2: $v")
}
class C: I1, I2 {
    init {
        super<I1>.m(1)
        super<I2>.m(2)
        m()
        //super<I1>.m() - так нельзя
    }
}
```

УМОЛЧАНИЯ

- По умолчанию классы финальны
- Методы классов - тоже
- Методы интерфейсов - нет (было бы странно)
- Ключевое слово `open` отменяет финальность

УМОЛЧАНИЯ

- override подразумевает open
- Но иногда хочется положить этому конец
- Можно перед override указать final

МОДИФИКАТОРЫ ДОСТУПА

- public доступность везде
- По умолчанию все - public (странная идея)
- protected - видимость в наследниках
- Только в наследниках

МОДИФИКАТОРЫ ДОСТУПА

- Вне класса protected нет
- private-элемент класса виден только в классе
- private-элемент файла - только в файле
- Внешний класс не видит private-элементы своих внутренних классов

МОДИФИКАТОРЫ ДОСТУПА

- Это отличает Kotlin от Java
- Нельзя использовать приватное поле в equals
- В этом что-то есть
- Но JVM не проверяет - поэтому можно обходить

JAVA

```java
class C1 {
    class C2 {
        private int f;
    }

    void f(C2 v) {
        System.out.println(v.f);
    }
}
```

KOTLIN

```kotlin
class C {
    protected class C1 {
        private val q = 5
    }

    fun f(v: Int) {
        v.q
    }
}
```

РЕГУЛЯРНОСТЬ

- Kotlin-модификаторы более регулярны по сравнению с Java
- Нет странного правила "protected-элементы доступны из других классов того же пакета"
- И нет странного зверя "package private"

РЕГУЛЯРНОСТЬ

- Элементы - public, private, protected - в классическом понимании
- Классы - public и private (то, что в Java называется "package private class")
- В плюс Java - закрытость класса по умолчанию

INTERNAL

- Обозначает видимость в рамках единицы сборки
- Например, собираем библиотеку
- Класс нужен много где в библиотеке
- Но он служебный

ВНУТРЕННИЕ КЛАССЫ

- Примерно как в Java
- Но смещены понятия
- По умолчанию - статические
- Умолчание меняется ключевым словом inner

ЧТО НАПЕЧАТАЕМ ?

```
1  class C {
2      private inner class C2 {
3          val v: String = vv
4      }
5
6      private val c = C2()
7      private val vv = "hello"
8      val vvv: String
9          get() = c.v
10 }
11
12 fun main() {
13     println(C().vvv)
14 }
```

KOTLIN

```kotlin
class Outer {
    class Static

    inner class Inner {
        init {
            println(this)
            println(this@Inner)
            println(this@Outer)
        }

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

KOTLIN

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

        inner class AnotherInner {
            init {
                println(this)
                println(this@Inner)
                println(this@Outer)
                println(this@AnotherInner)
            }
        }
    }
// ..................
```

KOTLIN

```kotlin
// ..............
    init {
        println(this)
        println(this@Outer)
    }
}

fun main() {
    Outer()
    println("------")
    Outer().Inner()
    println("+++++")
    Outer().Inner().AnotherInner()
}
```

SEALED-КЛАССЫ

- Начнем с мотивирующего примера
- Что-то типа второй домашки
- Допустим, решаем ее через when (на то могут быть причины)

KOTLIN

```kotlin
abstract class Expr

data class Value(val v: Int) : Expr()
data class Sum(val e1: Expr, val e2: Expr): Expr()

fun eval(e: Expr): Int = when (e) {
    is Value -> e.v
    is Sum -> eval(e.e1) + eval(e.e2)
    else -> throw IllegalArgumentException("why ?")
}
```

SEALED-КЛАССЫ

- Хотим убедить компилятор в том, что других подклассов нет
- Это делается с помощью ключевого слова sealed
- Все подклассы должны быть в том же файле
- Нужно, чтобы был не анонимный package

УБЕДИЛИ

```
1 sealed class Expr {
2     class Value(val v: Int) : Expr()
3     class Sum(val e1: Expr, val e2: Expr) : Expr()
4 }
5
6 fun eval(e: Expr): Int = when (e) {
7     is Expr.Value -> e.v
8     is Expr.Sum -> eval(e.e1) + eval(e.e2)
9 }
```

КОНСТРУКТОР СУПЕРКЛАССА

- Связь задается при объявлении класса
- Не в описании конструктора
- Страдает single-responsibility
- Меньше вариантов стыковки
- Вторичные связываются с суперклассом через первичного

ИСКЛЮЧЕНИЯ

- Бросить - throw
- Поймать - try/catch/finally
- Нет понятия 'checked exception'
- try - выражение

CHECKED EXCEPTION

```
1  class C {
2      public static void m(String s) throws IOException {
3          FileInputStream fis = new FileInputStream(s);
4          ///
5      }
6  }
7
8  class C2 {
9      public static void m2(List<String> lines) {
10         lines.forEach(C::m);  // так нельзя
11     }
12 }
```

CHECKED EXCEPTION

```
1 public static void m2(List<String> lines) {
2     lines.forEach(e -> {
3         try {   // можно, но хочется покомпактнее
4             C.m(e);
5         } catch (IOException exc) {
6             throw new UncheckedIOException(exc);
7         }
8     });
9 }
```

CHECKED EXCEPTION

```
1  // Так можно, но толку мало
2  public static Consumer<String> wrap(Consumer<String> f) {
3      return e -> {
4          try {
5              f.accept(e);
6          } catch (Exception exc) {
7              throw new RuntimeException(exc);
8          }
9      };
10 }
```

CHECKED EXCEPTION

```
1 // Потому что так – все равно нельзя
2
3 public static void m3(List<String> lines) {
4     lines.forEach(wrap(C1::m1));
5 }
```

USE

- try-with-resources в явном виде нет
- Потому что это умножение сущностей
- Есть метод use
- Как расширение Closeable

РЕАЛИЗАЦИЯ

```kotlin
public inline fun <T : Closeable?, R> T.use(
    block: (T) -> R
): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    var exception: Throwable? = null
    try {
        return block(this)
    } catch (e: Throwable) {
        exception = e
        throw e
    } finally {
// ..........
```

РЕАЛИЗАЦИЯ

```
// ...........
        when {
            apiVersionIsAtLeast(1, 1, 0) ->
                this.closeFinally(exception)
            this == null -> {}
            exception == null -> close()
            else -> try {
                close()
            } catch (exc: Throwable) {
 // cause.addSuppressed(exc) - to keep legacy behaiviour
            }
        }
    }
}
```

WITH/APPLY

- Хотим получить строку через StringBuilder
- Можно в лоб завести переменную-билдер
- Что-то с ней поделать
- Вернуть результат
- Kotlin позволяет сделать красивее

BASELINE

```
1 fun alphabet(): String {
2     val result = StringBuilder()
3     for (letter in 'A'..'Z') {
4         result.append(letter)
5     }
6     result.append("\nNow I know the alphabet!")
7     return result.toString()
8 }
```

KOTLIN-STYLE

```kotlin
fun alphabet() = with(StringBuilder()) {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
    this.toString()
}
```

WITH ВНУТРИ

```kotlin
public inline fun <T, R> with(
    receiver: T, block: T.() -> R
): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return receiver.block()
}
```

РАЗБЕРЕМ

- T.() -> R - "lambda with receiver"
- Формально - анонимная функция, в которой можно использовать this
- Явно или неявно
- И которая в конечном итоге вызывается как расширение

ПРИМЕР

```
1 fun main() {
2     val f: String.(Int) -> Int = {
3         it + length
4     }
5
6     println("hello".f(1))
7 }
```

ЭКВАВАЛЕНТЫ ПО РЕАЛИЗАЦИИ

```
1 fun main() {
2     val f1: (Pair<String, Int>) -> Int = {
3         it.second + it.first.length
4     }
5
6     println(f1(Pair("hello", 1)))
7 }
```

ЭКВАВАЛЕНТЫ ПО РЕАЛИЗАЦИИ

```
1 fun main() {
2     val f1: (String, Int) -> Int = { s, n ->
3         n + s.length
4     }
5
6     println(f1("hello", 1))
7 }
```

В ЧЕМ ПОЛЬЗА

- Помимо сахара
- Разделяем монолит на переменную и действия
- Можем комбинировать и выбирать независимо
- Можно преобразовывать действия
- Способствует DSL-стилю

APPLY

- Близкий аналог with
- Метод, а не функция
- Возвращает this
- Вариант использования: быстрая отладка

ПРИМЕР: JAVA

```java
public static IntFunction create(int delta) {
    return n -> n + delta;
}
public static void main(String[] args) {
    IntFunction incr = create(1);
    IntFunction incr10 = create(10);

    System.out.println(incr.apply(5));
    System.out.println(incr10.apply(2));
}
```

ИЗМЕНЕНИЕ СОСТОЯНИЯ

- Изменять состояние переменной не можем
- Потому что значение неизменяемой переменной примитивного типа можно добавить в замыкание
- А с адресом - сложнее, он на стеке
- А объект - можно. Он в куче

ОБХОД ОГРАНИЧЕНИЯ

```
1 public static Pair<IntConsumer, IntSupplier> create() {
2     AtomicInteger v = new AtomicInteger();
3     return new kotlin.Pair<>(n ->
4         v.addAndGet(n), () -> v.get()
5     );
6 }
7 // ..........
```

ОБХОД ОГРАНИЧЕНИЯ

```
// ...........
public static void main(String[] args) {
    Pair<IntConsumer, IntSupplier> s1 = create();
    Pair<IntConsumer, IntSupplier> s2 = create();

    s1.getFirst().accept(1);
    s1.getFirst().accept(22);
    s2.getFirst().accept(14);
    System.out.println(s1.getSecond().getAsInt());
    System.out.println(s2.getSecond().getAsInt());
    s2.getFirst().accept(-2);
    System.out.println(s2.getSecond().getAsInt());
}
```

FINAL

- И даже если не меняем - еще требуется финальность
- Не обязательно явная
- Достаточно фактической
- Называется 'effectively final'

ТАК НЕЛЬЗЯ

```
1 public static Pair<IntFunction, IntFunction>
2                createPair(int delta) {
3     IntFunction f1 = n -> n + delta; // ошибка компиляции
4     delta *= 2;
5     IntFunction f2 = n -> n + delta; // ошибка компиляции
6     return new Pair<>(f1, f2);
7 }
```

ТАК МОЖНО

```
1 public static Pair<IntFunction, IntFunction>
2                createPair (final int delta) {
3     IntFunction f1 = n -> n + delta;
4     int delta2 = delta * 2;
5     IntFunction f2 = n -> n + delta2;
6     return new Pair<>(f1, f2);
7 }
```

KOTLIN

- Все похоже
- Только функциональные типы унифицированы
- И можно работать с переменной "напрямую"

ПРИМЕР

```
1  fun createIncr(delta: Int = 1) = {n : Int ->
2      n + delta
3  }
4
5  fun main() {
6      val incr = createIncr()
7      val add5 = createIncr(5)
8
9      println(incr(4))
10     println(add5(12))
11 }
```

ПРИМЕР С ПЕРЕМЕННЫМИ

```
1 fun create(): Pair<(Int) -> Unit, () -> Int> {
2     var v = 0
3     return Pair({ n ->  v += n}, { v })
4 }
5 // ......
```

ПРИМЕР С ПЕРЕМЕННЫМИ

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

fun main() {
    val s1 = create()
    val s2 = create()

    s1.first(1)
    s1.first(22)
    s2.first(14)
    println(s1.second())
    println(s2.second())
    s2.first(-2)
    println(s2.second())
}
```

РЕАЛИЗАЦИЯ

- Под капотом - примерно то, что сделано в Java-примере
- Только тип - не атомарный
- Класс называется IntRef
- Можно даже использовать явным образом

ТРЕТИЙ ПРИМЕР

- Аналог третьего примера - тоже не откомпилируется
- Но по другой причине
- Потому что параметр функции в Kotlin - val по умолчанию
- Возможна неприятность в борьбе за компилируемость

ПЛОХОЙ ПРИМЕР

```
fun createPair(delta: Int):
        Pair<(Int) -> Int, (Int) -> Int> {

    val f1 = { n: Int -> n + delta }
    delta = delta * 2
    val f2 = { n: Int -> n + delta }
    return Pair(f1, f2)
}
```

ИСПРАВЛЕНИЕ КУРИЛЬЩИКА

```kotlin
fun createPair(delta: Int):
        Pair<(Int) -> Int, (Int) -> Int> {

    var d = delta
    val f1 = { n: Int -> n + d }
    // Kotlin не ругается на то, что это
    // не 'effective final'
    d = d * 2
    val f2 = { n: Int -> n + d }
    return Pair(f1, f2)
}

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

ИСПРАВЛЕНИЕ КУРИЛЬЩИКА

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

fun main() {
    val q = createPair(10)
    println(q.first(5))
}
```

ИСПРАВЛЕНИЕ ЗДОРОВОГО ЧЕЛОВЕКА

```
1  fun createPair(delta: Int):
2          Pair<(Int) -> Int, (Int) -> Int> {
3
4      val f1 = { n: Int -> n + delta }
5      val d = delta * 2
6      val f2 = { n: Int -> n + d }
7      return Pair(f1, f2)
8  }
9
10 fun main() {
11     val q = createPair(10)
12     println(q.first(5))
13 }
```