АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 7

ПЛАН

- Замыкания
- Делегирование
- Nullable

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

```
1  // ...........
2  public static void main(String[] args) {
3      Pair<IntConsumer, IntSupplier> s1 = create();
4      Pair<IntConsumer, IntSupplier> s2 = create();
5
6      s1.getFirst().accept(1);
7      s1.getFirst().accept(22);
8      s2.getFirst().accept(14);
9      System.out.println(s1.getSecond().getAsInt());
10     System.out.println(s2.getSecond().getAsInt());
11     s2.getFirst().accept(-2);
12     System.out.println(s2.getSecond().getAsInt());
13 }
```

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

```
 1 // ..............
 2
 3 fun main() {
 4     val s1 = create()
 5     val s2 = create()
 6
 7     s1.first(1)
 8     s1.first(22)
 9     s2.first(14)
10     println(s1.second())
11     println(s2.second())
12     s2.first(-2)
13     println(s2.second())
14 }
```

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

```
1 fun createPair(delta: Int):
2         Pair<(Int) -> Int, (Int) -> Int> {
3
4     val f1 = { n: Int -> n + delta }
5     delta = delta * 2
6     val f2 = { n: Int -> n + delta }
7     return Pair(f1, f2)
8 }
```

ИСПРАВЛЕНИЕ КУРИЛЬЩИКА

```
 1 fun createPair(delta: Int):
 2         Pair<(Int) -> Int, (Int) -> Int> {
 3
 4     var d = delta
 5     val f1 = { n: Int -> n + d }
 6     // Kotlin не ругается на то, что это
 7     // не 'effective final'
 8     d = d * 2
 9     val f2 = { n: Int -> n + d }
10     return Pair(f1, f2)
11 }
12
13 // ......................
```

ИСПРАВЛЕНИЕ КУРИЛЬЩИКА

```
1 // ...........
2
3 fun main() {
4     val q = createPair(10)
5     println(q.first(5))
6 }
```

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 }
```

ДЕЛЕГИРОВАНИЕ

- В классическом ООП многое строилось на наследовании
- Одна из проблем - у сущностей зоопарк атрибутов/поведения
- Они относятся к разным смысловым группам
- И объект каждую такую группу может наследовать от разных объектов
- И уж совсем точно - не от более абстрактного понимания самого себя

ДЕЛЕГИРОВАНИЕ

- Это можно решать с помощью интерфейсов
- Создавать интерейсы для смысловых групп атрибутов/поведения
- Что-то вроде Runnable, Chargable, Storable, Observable

ДЕЛЕГИРОВАНИЕ

- Но если этим ограничиться - может потребоваться писать повторяющиеся реализации
- А если ограничиться реализацией по умолчанию в интерфейсе - не получится наследовать разное поведение от разных объектов

ДЕЛЕГИРОВАНИЕ

- Можем принимать в конструкторе ссылки на другие объекты
- И в реализации методов интерфейсов ставить вызов метода другого объекта
- Это нормально работает
- Но требует boilerplate-кода
- Kotlin поддерживает этот шаблон в языке

НА ПРИМЕРЕ: ИНТЕРФЕЙСЫ

```
1 interface Chargeable {
2     fun charge(value: Int)
3 }
4
5 interface Employer: Chargeable, Named {
6     val employees: List<Employee>
7 }
8
9 // ..................
```

НА ПРИМЕРЕ: ИНТЕРФЕЙСЫ

```
1 // ..............
2
3 interface Employee: Chargeable, Named {
4     val employer: Employer
5 }
6
7 interface Named {
8     val name: String
9 }
```

НА ПРИМЕРЕ: КЛАСС

```
1  class Company(override val name: String):
2          Named, Chargeable, Employer {
3      override val employees: List<Employee>
4
5      init {
6          employees = listOf(OrdinaryAdult(
7              "Ivan Petrov", this)
8          )
9      }
10
11     override fun charge(value: Int) {
12         println("ok")
13     }
14 }
```

НА ПРИМЕРЕ: ЕЩЕ КЛАССЫ

```
1  class OrdinaryAdult(
2      override val name: String,
3      override val employer: Employer
4  ): Chargeable, Employee, Named {
5
6      override fun charge(value: Int) {
7          if (value > 20000) {
8              println("let me think")
9          } else {
10             println("ok")
11         }
12     }
13 }
```

НА ПРИМЕРЕ: ЕЩЕ КЛАССЫ

```
1 // ............
2
3 class TinAger(
4     override val name: String,
5     fundProvider: Chargeable
6 ): Named, Chargeable by fundProvider
7
8 data class EmpInBusinessTrip(val who: Employee):
9     Chargeable by who.employer, Named by who
```

ИСПОЛНЯЕМ

```
1 fun main() {
2     val company = Company("yandex")
3     val farther = company.employees[0]
4     val vasya = TinAger("Vasya", farther)
5     vasya.charge(100)
6     vasya.charge(100000)
7
8     EmpInBusinessTrip(farther).charge(100)
9 }
```

МИНУС ДЕЛЕГИРОВАНИЯ

- При наследовании метод суперкласса может реализовать крупную схему поведения
- Для элементов которой вызывать другие protected-методы
- Либо абстрактные, либо переопределяемые
- Используя факт наследования
- При делегировании это в лучшем случае усложняется

NULLABLE

- В JVM ссылочный тип может хранить null
- Попытки его разыменовать порождают исключение
- Проблема 1: это происходит во время исполнения
- Проблема 2: диагностика часто не дает понимания
- Хотя и получше, чем в более ранних версиях

ПРИМЕР

```
 1 public class Main {
 2     public static int m(URL first, URL second) {
 3         return first.getQuery().length()
 4                         + first.getQuery().length();
 5     }
 6
 7     public static void main(String[] args)
 8             throws Exception {
 9
10         m(new URL("http://ya.ru"),
11           new URL("http://ya.ru/?qqq=123")
12         );
13     }
14 }
```

ВАРИАНТЫ РЕШЕНИЯ

- Можно вставлять явные проверки
- Улучшать диагностику
- Но это boolerplate-код
- И это сложно проверить статически

ВАРИАНТЫ РЕШЕНИЯ

- Есть вариант Option-типа в двумя вариантами: Some(value) и None
- Давно придуман, хорошо проработан
- Заставляет обрабатывать null-вариант
- Следит за этим статически
- Стыкуется с generic-типизацией и с коллекциями

OPTIONAL В JVM

- В JVM есть Optional
- Доступен в Kotlin
- Реализован на троечку
- В силу этого - является скорее умножением сущностей

ПУТЬ KOTLIN

- Kotlin пошел своим путем
- Для каждого типа есть nullable-модификация
- Обозначается знаком вопроса после типа
- Например, String? - обнуляемая строка

ПУТЬ KOTLIN

- Смысл в том, чтобы изолировать преобразования типов данных
- От тех точек, в которых может быть получен null
- Например, при получении значения по ключу (если ключа нет в словаре)
- Или при неудачном чтении файла

ПРИМЕР

```kotlin
data class Person(val id: Long, val name: String)

object Registry {
    val data: Map<String, Person>

    init {
        val vasya = Person(123, "vasya")
        data = mapOf(vasya.name to vasya)
    }

    fun byName(query: String): Person? = data[query]
}

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

ПРИМЕР

```
1 // ..............
2
3 fun main() {
4     println(Registry.byName("vasya"))
5     println(Registry.byName("dima"))
6 }
```

ХОТИМ ПОЛУЧИТЬ ID ПО ИМЕНИ

- Первый вариант по семантике: получить значение типа Int?
- Второй вариант по семантике: получить значение типа Int с маркером отсутствия
- Первый вариант по реализации: явная проверка
- Второй вариант по релизации: встроенная конструкция

ПРИМЕР

```
object Registry {
    val data: Map<String, Person>

    init {
        val vasya = Person(123, "vasya")
        data = mapOf(vasya.name to vasya)
    }

    fun byName(query: String): Person? = data[query]

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

ПРИМЕР

```
1  // ..............
2      fun idByName1(query: String): Long? {
3          val p = byName(query)
4          return if (p == null) null else p.id
5      }
6
7
8      fun idByName2(query: String): Long? =
9          byName(query)?.id
10 }
11 // ..............
```

ПРИМЕР

```
1 // ..............
2
3 fun main() {
4     println(Registry.idByName1("vasya"))
5     println(Registry.idByName2("vasya"))
6     println(Registry.idByName1("dima"))
7     println(Registry.idByName2("dima"))
8 }
```

ПРИМЕР

```
object Registry {
    // ........

    fun idByName3(query: String): Long {
        val p = byName(query)
        return if (p == null) -1 else p.id
    }

    fun idByName4(query: String): Long = byName(query)?.id?
}
// ............
```

ПРИМЕР

```
1 // ............
2
3 fun main() {
4     println(Registry.idByName3("vasya"))
5     println(Registry.idByName4("vasya"))
6     println(Registry.idByName3("dima"))
7     println(Registry.idByName4("dima"))
8 }
```

ПРИЕМЫ РАБОТЫ С NULLABLE

- При явной проверке на null в ветке, где значение гарантированно не null - оно рассматривается как значение обычного типа
- С теми же оговорками, как для умного приведения
- Safe call (?.) - проверка на null
- И обращение к методу/свойству, если не null

ПРИЕМЫ РАБОТЫ С NULLABLE

- Safe call удобно выстравается в цепочки
- Когда идем вглубь вложенных структур по nullable-полям
- Или по цепочке методов, возвращающих nullable
- Как-то так:

```
order?.date?.month
```

LET

- let - стандартное расширение
- Применение блока к данному объекту
- Результат блока - результат let
- Идиоматично применяется в связке с safe call

LET

- Часто - с Unit-результатом
- Например - напечатать не-null элемент
- Не Unit - когда надо не просто свойство/метод взять
- А как-то похитрее преобразовать null-значение

ПРИМЕР

```
1  data class Person(val id: Long, val name: String,
2                    val details: String?)
3
4  object Registry {
5      val data: Map<String, Person>
6      init {
7          val vasya = Person(123, "vasya", "qwerty")
8          val petya = Person(234, "petya", null)
9          data = mapOf(vasya.name to vasya,
10                      petya.name to petya)
11     }
12
13     fun byName(query: String): Person? = data[query]
14 }
```

ПРИМЕР

```
1  // ............
2  fun f(names: List<String?>) = names.map{ n ->
3      n?.let(Registry::byName)
4          ?.details
5          ?.let {
6              println("name: $n: $it")
7          }
8  }
9
10 fun main() {
11     f(listOf("vasya", "dima", null, "petya"))
12 }
```

ЧТО ЕЩЕ

- run - синоним let
- also - как let, но без возврата значения
- apply - примерно как also
- Только не-null значение предстает как this

ПРИМЕР

```
1 fun f(names: List<String?>) = names.map{ n ->
2     n?.run(Registry::byName)
3         ?.details
4         ?.apply {
5             println("name: $n: $this")
6         }
7 }
```

ЧТО ЕЩЕ

- ?.takeIf - дополнительная фильтрация
- Если не выполнено условие, получаем null
- ?.takeUnless - инверсия

ТОНКОСТЬ

- Пусть it - String?
- Есть it?.let и есть it?.map
- Есть it?.takeIf и есть it?.filter
- Первое - про строку в целом, второе - про символы

ПОСЛОЖНЕЕ

- Есть список ключей
- Хотим для каждого ключа обратиться в словарь
- Получится список элементов типа Person?
- Хотим сделать map для непустых

ПОСЛОЖНЕЕ

- Можно сделать в лоб фильтрацию по

```
it != null
```

- И отдельно map
- Но с точки зрения типов в map все равно будет nullable-тип
- Если сделать elvis - получим nullable-тип в результате

ПОСЛОЖНЕЕ

- Есть отдельная конструкция - !!
- Принудительное приведение nullable в обычный тип
- NPE - в случае, если null
- Технически подойдет, но плохой стиль

ПОСЛОЖНЕЕ

- Есть целевые функции - filterNotNull и mapNotNull
- Они решают проблему
- Но сами по себе кажутся умножением сущностей
- С ними ничего не поделаешь - издержки подхода

ПЛЮСЫ И МИНУСЫ

- Плюс: Концептуальная близость JVM-терминологии
- Плюс: Отсутствие runtime-издержек
- Минус: изобретение велосипеда относительно Option-модели
- Минус: "угловатость" конструкций, невписываемость в обобщаемые модели
- Минус: неразличение "порядка" отсутствия