АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 7

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 интерфейсы 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

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

interface Employee: Chargeable, Named {
    val employer: Employer
}

interface Named {
    val name: String
}
```

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

```
class OrdinaryAdult(
    override val name: String,
    override val employer: Employer
): Chargeable, Employee, Named {

    override fun charge(value: Int) {
        if (value > 20000) {
            println("let me think")
        } else {
            println("ok")
        }
    }
}
```

НА ПРИМЕРЕ: ЕЩЕ КЛАССЫ

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

class TinAger(
    override val name: String,
    fundProvider: Chargeable
): Named, Chargeable by fundProvider

data class EmpInBusinessTrip(val who: Employee):
    Chargeable by who.employer, Named by who
```

ИСПОЛНЯЕМ

```
fun main() {
    val company = Company("yandex")
    val farther = company.employees[0]
    val vasya = TinAger("Vasya", farther)
    vasya.charge(100)
    vasya.charge(100000)

    EmpInBusinessTrip(farther).charge(100)
}
```

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

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

fun main() {
    println(Registry.byName("vasya"))
    println(Registry.byName("dima"))
}
```

ХОТИМ ПОЛУЧИТЬ ID ПО ИМЕНИ

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

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

```
// ..............
    fun idByName1(query: String): Long? {
        val p = byName(query)
        return if (p == null) null else p.id
    }


    fun idByName2(query: String): Long? =
        byName(query)?.id
}
// ..............
```

ПРИМЕР

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

fun main() {
    println(Registry.idByName1("vasya"))
    println(Registry.idByName2("vasya"))
    println(Registry.idByName1("dima"))
    println(Registry.idByName2("dima"))
}
```

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

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

fun main() {
    println(Registry.idByName3("vasya"))
    println(Registry.idByName4("vasya"))
    println(Registry.idByName3("dima"))
    println(Registry.idByName4("dima"))
}
```

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

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

object Registry {
    val data: Map<String, Person>
    init {
        val vasya = Person(123, "vasya", "qwerty")
        val petya = Person(234, "petya", null)
        data = mapOf(vasya.name to vasya,
                     petya.name to petya)
    }

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

ПРИМЕР

```
// .............
fun f(names: List<String?>) = names.map{ n ->
    n?.let(Registry::byName)
        ?.details
        ?.let {
            println("name: $n: $it")
        }
}

fun main() {
    f(listOf("vasya", "dima", null, "petya"))
}
```

ЧТО ЕЩЕ

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

ПРИМЕР

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

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

GENERICS

- Похожая конструкция есть в Java
- Но менее развитая
- И с грузом legacy

ОБЩЕЕ ОПИСАНИЕ

- Применяется к классам и статическим функциям/методам
- Простейшая форма - именованный тип-параметр
- Тоже в угловых скобках
- Пока как в Java

ПРИМЕР

```
class Holder<T>(val value: T)

fun main() {
    val intHolder = Holder<Int>(123)
    val stringHolder = Holder<String>("string")

    val intHolder2 = Holder(123)
    val stringHolder2 = Holder("string")
}
```

ОТЛИЧИЕ ОТ JAVA

- Все применения generic-типа требуют указания типа
- Нет legacy-варианта "generic без типовых параметров"
- intHolder2/stringHolder2 - это просто type inference
- Так уже нельзя:

```
var intHolder2: Holder
```

ПРИМЕР

```
fun <T> findDuplicates(data: List<T>): Set<T> {
    val result = mutableSetOf<T>()
    val found = mutableSetOf<T>()

    data.forEach {
        val target = if (it in found) result else found
        target += it
    }

    return result.toSet()
}
// ..............
```

ПРИМЕР

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

fun main() {
    println(findDuplicates(listOf("a", "a", "b")))
    println(findDuplicates(
        listOf("bb", "b", "a", "a", "bb", "a", "cc")
    ))
}
```

ПРИМЕР

```
data class Holder<T>(val value: T) {
    fun <T2> map(f: (T) -> T2): Holder<T2> =
        Holder(f(value))
}

fun <E> holderOf(v: E) = Holder(v)

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

ПРИМЕР

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

fun main() {
    val intHolder = holderOf(123)
    val stringHolder = Holder<String>("string")

    println(intHolder)
    println(stringHolder)
    println(stringHolder.map { it.length })
}
```

ПРИМЕР

```
1  sealed interface Either<L, R> {
2      fun isLeft(): Boolean
3      fun isRight(): Boolean
4      fun swaped(): Either<R, L>
5  }
6
7  data class Left<L, R>(val left: L): Either<L, R> {
8      override fun isLeft(): Boolean = true
9      override fun isRight(): Boolean = false
10     override fun swaped(): Either<R, L> = Right(left)
11 }
12
13 // ............
```

ПРИМЕР

```
// ..............
data class Right<L, R>(val right: R): Either<L, R> {
    override fun isLeft(): Boolean = false
    override fun isRight(): Boolean = true
    override fun swaped(): Either<R, L> = Left(right)
}
fun main() {
    val v1: Either<String, Int> = Left("")
    val v2: Either<String, Int> = Right(5)
    println(v1)
    println(v1.swaped())
    println(v2)
    println(v2.swaped())
}
```

GENERIC-МЕТОДЫ

- У них могут быть свои типы-параметры
- И типы-параметры объемлющего класса
- При совпадении имен внутреннее приоритетнее внешнего
- В функциях могут определяться локальные классы с параметрами-типами
- Или в локальных классах использоваться параметры-типы функции

ПРИМЕР

```
fun <T> f(s: T): Callable<List<T>> {
    class C(private val v: T): Callable<List<T>> {
        override fun call(): List<T> = listOf(v, v)
    }

    return C(s)
}

fun main() {
    println(f("hello").call())
    println(f(123).call())
}
```

GENERIC В РАСШИРЕНИЯХ

- Никто не запрещает
- И в составе параметра
- И в качестве расширяемого типа

ПРИМЕР

```
fun <T> T.printIf(f: (T) -> Boolean) {
    if (f(this)) {
        println(this)
    }
}

fun main() {
    1.printIf {it % 2 == 1}
    "123".printIf {it.length > 3}
    "12345".printIf {it.length > 3}
}
```

ОСОБЕННОСТИ

- Конкретные расширения имеют приоритет над обобщенными
- Если возникает неоднозначность - ошибка компиляции
- Если дважды встречается один тип-параметр - подразумеваются значения одного типа
- Если в реальности разные - ищется общий

ПРИМЕР

```
1 fun <T> T.m(other: T) {
2     println("1")
3 }
4
5 fun <T> T.m(other: String) {
6     println("2")
7 }
8
9 // ..........
```

ПРИМЕР

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

fun <T> Int.m(other: T) {
    println("3")
}

fun Int.m(other: String) {
    println("4")
}

fun main() {
    1.m("hello")
}
```

РЕАЛИЗАЦИЯ

- Как обычное расширение - только тип-параметр меняется на java.lang.Object
- Как для Any
- Generic-расширение может сочетаться с одноименным расширением конкретного типа
- Но если этим типом будет Any - получим конфликт и ошибку компиляции

NULLABLE

- Nullable в сигнатурах методов реализуется добавлением аннотации
- И добавкой проверки на null для необнуляемых параметров
- Это означает невозможность иметь два метода с одним именем, отличающихся только обнуляемостью параметра

NULLABLE

- Например:

```
fun m(s: String)
```

и

```
fun m(s: String?)
```

NULLABLE

- Nullable-тип может быть подставлен в тип-параметр
- Знак вопрос можно добавить к типу-параметру
- Синтаксически можно получить "двойной nullable"
- Если используется что-то типа T?
- A в T подставляется String?

ПРИМЕР

```
1  fun <T> m1(a: T?) {
2      println(a)
3  }
4
5  fun <T> m2(a: T) {
6      println(a)
7  }
8
9  fun mm1(s: String) {
10     m1(s)
11     m1(null)
12 }
13
14 // ..................
```

ПРИМЕР

```
1  // ...................
2
3  fun mm2(s: String?) {
4      m1(s)
5      m1(null)
6  }
7
8  fun main() {
9      mm1("hello")
10     m2("hello")
11     m2(null)
12     mm2("hello")
13 }
```

NOTHING

- Nothing - отдельный тип
- Одна из интерпретаций - тип вычисления, которое не завершится
- Из-за гарантированно бесконечного цикла или брошенного исключения
- Надо какой-то тип подставить
- Но по сути - нам все равно

NOTHING

- Формально он приводим к любому типу
- Результат функции, которая не завершится штатно, можно присвоить обычной типизированной переменной
- Или объявить функцию как что-то возвращающую - и при этом бросить исключение

NOTHING

- Обратно - привести нельзя
- Нельзя объявить функцию как возвращающую Nothing
- И попытаться вернуть 15, "hello", Unit или Any

ПРИМЕР

```
1 fun f1(): Nothing {
2     throw RuntimeException()
3 }
4
5 fun f2(): Nothing {
6     while (true) {}
7 }
8 // ...............
```

ПРИМЕР

```
1 // ..............
2
3 fun main() {
4     val q1: Int = f1()
5     println(q1)
6
7     val q2 = f2()
8     //println(q2) // не откомпилируется
9 }
```

В КОНТЕКСТЕ ОБОБЩЕННЫХ ТИПОВ

- Пример: хотим определить тип Option<T>
- Интерфейс с двумя реализациями: Some и None
- Для Some ничего не придумаешь лучше, чем Some<T>
- Для None хотелось бы иметь объект

NOTHING

- Объект не может быть generic
- Нужно, чтобы он был какого-то типа
- На эту роль подходит Nothing
- Чтобы любому Option<T> присваивать наследника Option<Nothing>

НЕКРАСИВО

```
1  sealed interface Option<out T>
2  data class None<T>(private val dummy: Int=0): Option<T>
3  data class Some<Q>(val value: Q): Option<Q>
4
5
6  fun main() {
7      val strOpt = Some("hello")
8      val intOpt = Some(112233)
9      val strOpt2: Option<String> = None()
10     val intOpt2: Option<Int> = None()
11 }
```

ПОЧТИ ТО, ЧТО НАДО

```kotlin
sealed interface Option<T>
object None: Option<Nothing>
data class Some<Q>(val value: Q): Option<Q>

fun main() {
    val strOpt = Some("hello")
    val intOpt = Some(112233)
//    val strOpt2: Option<String> = None
//    val intOpt2: Option<Int> = None
}
```

КОВАРИАНТНОСТЬ

- Хотим, чтобы Option<A> был подтипом Option<B>
- Если A - подтип B
- Это важно, чтобы заработала схема с Nothing
- И само по себе неплохо

КОВАРИАНТНОСТЬ

- Добавим ключевое слово out - и оно заработает

```
1  sealed interface Option<out T>
2  object None: Option<Nothing>
3  data class Some<Q>(val value: Q): Option<Q>
4
5  fun main() {
6      val strOpt = Some("hello")
7      val intOpt = Some(112233)
8      val strOpt2: Option<String> = None
9      val intOpt2: Option<Int> = None
10 }
```

ВОЗНИКАЮТ ВОПРОСЫ

- Почему такого нет по умолчанию ?
- Почему тут ключевое слово именно out ?
- Почему в Java такого нет совсем ?
- Рассмотрим другую ситуацию

СИТУАЦИЯ

- Сделали изменяемый Option
- Как-то так:

```
data class Some<T>(var value: T): Option<T>
```

- Написали функцию, принимающую параметром Option<Any>
- И передали туда Option<String>

СИТУАЦИЯ

- Изнутри функции Option<String> понимается как Option<Any>
- И статически нет никаких препятствий привоить 123 (число) в value
- Возникает неприятный выбор
- Либо динамически падать при таком присваивании, либо серьезно портить Option-объект

СИТУАЦИЯ

- В JVM будет падение в момент присваивания
- Если совсем точно - не в момент исполнения, а при верификации байткода
- Java контролирует на уровне языка путем более ограничительных правил
- Kotlin коктролирует гибче, но усложняются концепции

ЧТО ИМЕЕМ И КУДА ДВИЖЕМСЯ

- Можем отношение наследования перенести из типа-параметра в объемлющий тип
- Если поля этого типа-параметра неизменяемы
- А если это не так - тогда "как в Java"
- Но можно добавить гибкости

ДОБАВИМ ДЕТАЛЕЙ

- Пусть есть чистый set-метод
- Он только меняет значение свойства value
- Объявить параметр как Option<Any> и передавать Option<String> нельзя
- А наоборот - можно
- Нет ничего плохого в присваивании строки полю объекта типа Option<Any>

ПОЧЕМУ OUT

- Многое зависит от того, что мы делаем с сущностями типа-параметра
- Это ярко проявляется в типе Function
- Возьмем вариант с одним параметром и результатом

ПОЧТИ ТО, ЧТО НАДО

```
fun main() {
    f1(::asString)
    f1(::asStringBuilder)
    f1(::asString2)
    f1(::asStringBuilder2)
}

fun asString(v: Any): String = v.toString()

fun asStringBuilder(v: Any): StringBuilder =
    StringBuilder(v.toString())

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

ПОЧТИ ТО, ЧТО НАДО

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

fun asString2(v: CharSequence): String =
    v.length.toString()

fun asStringBuilder2(v: CharSequence): CharSequence =
    StringBuilder(v.toString())

fun f1(f: (CharSequence) -> CharSequence) {
    f("hello")
    f(StringBuilder("hello"))
    f(f("hello"))
}
```

РАЗБЕРЕМ

- Параметр f1 - функция типа

```
(CharSequence) -> CharSequence
```

- Можем передать функцию с точным соответствием сигнатуры
- Можем передать функцию с параметром более широкого типа
- Или с результатом более узкого типа

РАЗБЕРЕМ

- Параметр f1 - функция типа

```
Function1<CharSequence, CharSequence>
```

- Первый параметр in, второй out

FUNCTION1

```
1 public interface Function1<in P1, out R>:
2     Function<R> {
3
4     // Invokes the function with the specified argument
5     public operator fun invoke(p1: P1): R
6 }
```

ПРАВИЛА

- Есть понятия "ковариантная позиция" и "контравариантная позиция"
- Ковариантая - чтение значения в широком смысле
- Контравариантная - запись в широком смысле

ПРАВИЛА

- Если тип-параметр всегда используется в ковариантных позициях - можно сделать ковариантным
- И желательно так сделать
- Если только в контравариантных - сделать контравариантным
- Если и так, и так - инвариантным (как в Java всегда)

ПРАВИЛА

- Речь шла про вариантность на уровне типа
- Бывают локальные уточнения
- Если функция только меняет значение данного типа - его можно сделать локально контравариантным

ПРИМЕР

```
1 fun <T> changer(opt: Some<in T>, v: T) {
2     opt.value = v
3 }
```