АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 8

ПЛАН

- Generics
- Ковариантность и контравариантность

ОБЩЕЕ ОПИСАНИЕ

- 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

```
1 class Holder<T>(val value: T)
2
3 fun main() {
4     val intHolder = Holder<Int>(123)
5     val stringHolder = Holder<String>("string")
6
7     val intHolder2 = Holder(123)
8     val stringHolder2 = Holder("string")
9 }
```

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

```
1 // ..................
2
3 fun main() {
4     println(findDuplicates(listOf("a", "a", "b")))
5     println(findDuplicates(
6         listOf("bb", "b", "a", "a", "bb", "a", "cc")
7     ))
8 }
```

ПРИМЕР

```
1 data class Holder<T>(val value: T) {
2     fun <T2> map(f: (T) -> T2): Holder<T2> =
3         Holder(f(value))
4 }
5
6 fun <E> holderOf(v: E) = Holder(v)
7
8 // ............
```

ПРИМЕР

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

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

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

ПРИМЕР

```kotlin
sealed interface Either<L, R> {
    fun isLeft(): Boolean
    fun isRight(): Boolean
    fun swaped(): Either<R, L>
}

data class Left<L, R>(val left: L): Either<L, R> {
    override fun isLeft(): Boolean = true
    override fun isRight(): Boolean = false
    override fun swaped(): Either<R, L> = Right(left)
}

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

ПРИМЕР

```kotlin
// ..............
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

```
 1 fun <T> f(s: T): Callable<List<T>> {
 2     class C(private val v: T): Callable<List<T>> {
 3         override fun call(): List<T> = listOf(v, v)
 4     }
 5
 6     return C(s)
 7 }
 8
 9 fun main() {
10     println(f("hello").call())
11     println(f(123).call())
12 }
```

GENERIC В РАСШИРЕНИЯХ

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

ПРИМЕР

```
 1 fun <T> T.printIf(f: (T) -> Boolean) {
 2     if (f(this)) {
 3         println(this)
 4     }
 5 }
 6
 7 fun main() {
 8     1.printIf {it % 2 == 1}
 9     "123".printIf {it.length > 3}
10     "12345".printIf {it.length > 3}
11 }
```

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

```
1  sealed interface Option<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 }
```

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

```
 1 fun main() {
 2     f1(::asString)
 3     f1(::asStringBuilder)
 4     f1(::asString2)
 5     f1(::asStringBuilder2)
 6 }
 7
 8 fun asString(v: Any): String = v.toString()
 9
10 fun asStringBuilder(v: Any): StringBuilder =
11     StringBuilder(v.toString())
12
13 // ..............
```

ПОЧТИ ТО, ЧТО НАДО

```
1  // ..............
2
3  fun asString2(v: CharSequence): String =
4      v.length.toString()
5
6  fun asStringBuilder2(v: CharSequence): CharSequence =
7      StringBuilder(v.toString())
8
9  fun f1(f: (CharSequence) -> CharSequence) {
10     f("hello")
11     f(StringBuilder("hello"))
12     f(f("hello"))
13 }
```

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
- Можно 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 }
```