Тема 2. Знакомство с объектно-ориентированными возможностями языка Kotlin

Романов Владимир Юрьевич
МГУ им. М.В.Ломоносова, ф-т ВМК
romanov.rvy@yandex.ru

Ноябрь 7, 2020

1.2. ЗНАКОМСТВО С ОБЪЕКТНО-ОРИЕНТИРОВАННЫМИ ВОЗМОЖНОСТЯМИ

Объявление интерфейсов

```
interface Move {
    val piece: Piece?

    @Throws(GameOver::class)
    fun doMove()

    fun undoMove() {}
}
```

- значение для свойства **piece** в интерфейсе не хранится

- метод интерфейса в Kotlin может иметь *реализацию по умолчанию.*
  Ключевое слово ***default*** (как в Java) не требуется.
  Достаточно наличия тела метода.

Наследование интерфейсов

```
interface ITransferMove : Move {
    /**
     * Вернуть клетку откуда пошла фигура.
     */
    val source: Square

    /**
     * Вернуть клетку куда пошла фигура.
     */
    val target: Square
}
```

- в интерфейсе-потомке могут быть объявлены новые свойства и функции
- часть свойств и функций в интерфейсе-потомке может быть реализована

Реализация интерфейсов

В языке Kotlin применение модификатора ***override*** обязательно

```
open class SimpleMove(
        override val source: Square,
        override val target: Square,
) : ITransferMove {
    override var piece: Piece? = null

    init { piece = source.getPiece()!! }

    override fun doMove() {
        piece?.moveTo(target)
    }
    override fun undoMove() {
        piece?.moveTo(source)
    }
}
```

Реализация свойств интерфейсов

- свойство интерфейса может быть реализовано как *переменная* ИЛИ как *метод доступа*

```
interface IPlayer {
    val name: String

    val authorName: String
}
class Neznaika : MovePiecePlayer() {
  override val name: String
            get() = "Незнайка"
  override val authorName: String
            = "Романов В.Ю."
}
```

Модификатор класса abstract

```
abstract
class Piece(var square: Square,
            var color: PieceColor) {
  abstract
  fun isCorrectMove(vararg squares: Square)
      : Boolean

  abstract
  fun makeMove(vararg squares: Square): Move
}
```

Модификатор класса abstract

```
abstract
class ChessPiece(square: Square,
                 color: PieceColor)
        : Piece(square, color)
{
  override fun isCorrectMove(
           vararg squares: Square): Boolean {
    val target = squares[0]
    val piece = target.getPiece() ?: return true

    return color !== piece.color
  }
}
```

Модификатор класса open

```
open class Capture(source: Square, target: Square)
    : SimpleMove(source, target), ICaptureMove {
    var capturedSquare: Square = target
    var capturedPiece: Piece = target.piece!!

    override val captured: List<Square>
        get() = listOf(capturedSquare)

    override fun doMove() {
        capturedPiece.remove()
        super.doMove()
    }
    override fun undoMove() {
        super.undoMove()
        capturedSquare.setPiece(capturedPiece)
    }
}
```

Модификатор класса final

Класс по умолчанию с модификатором ***final***

```
class Promotion(source: Square, target: Square)
    : SimpleMove(source, target), ICaptureMove {
    private val pawn: Piece = source.piece!!
    private var capturedPiece: Piece? = null
    private var promotedPiece: Piece? = null

    // ...
}
```

Модификатор видимости

Модификатор	Член класса	Объявление верхнего уровня
public (по умолчанию)	Доступен повсюду	Доступно повсюду
internal	Доступен только в модуле	Доступно в модуле
protected	Доступен	Доступно в подклассах
private	Доступен в классе	Доступно в файле

Модули Kotlin и модификатор internal

Модификатор видимости **internal** означает видимость в пределах модуля.

Модуль - множество файлов на языке Kotlin скомпилированных вместе:

- модуль в **IntelliJ**
- проект **Maven**
- множество исходных текстов в **Gradle**
- множестов файлов скомпилированных одни вызовом задачи **Ant**

Модификатор видимости Kotlin и Java

Kotlin	Java	причина
internal	public	В Java нет ограничение на модули
private	protected	В Java нет приватных классов

Особености модификаторов видимости в Java и Kotlin

- **Java**: член класса с модификатором ***protected***
  доступен в самом классе, в подклассах **И** в пакете класса
- **Kotlin**: член класса с модификатором ***protected***
  доступен **ТОЛЬКО** в самом классе и его подклассах.
- **Kotlin**: *функциям-расширениям* класса **НЕ** доступны
  члены класса с модификаторами ***protected*** или ***private***.
- **Kotlin**: к классам, функциям и свойствам верхнего уровня
  применим модификатор ***private***.
  Их объявления видны **ТОЛЬКО** в файле.
- **Kotlin**: внешний класс не видит приватных членов
  внутренних (вложенных) классов

Модификатор sealed

- Модификатор ***sealed*** в объявлении базового класса
- Ограничивает возможность создания подклассов
- Все прямые подклассы должны быть вложены в базовый класс

```
sealed class Expr {
    class Num( val value : Int)
        : Expr( )
    class Sum(val left: Expr, val right : Expr)
        : Expr()
}
```

Основной конструктор класса

```
class GamePanel
      protected constructor(val game: Game)
      : JPanel(BorderLayout())
{
    private var control: GameControlPanel
    private var adorned: AdornedBoard
            = AdornedBoard()
    private var history: MovesHistory
            = MovesHistory(game.board.history)
    init {
        add(history, BorderLayout.LINE_END)
    }
```

- первичный конструктор не может содержать код инициализации
- ключевое слово ***init*** - начало блока инициализации
- блоков инициализации может быть несколько

Основной конструктор класса

```
class GamePanel(val game: Game)
      : JPanel(BorderLayout())
{
    private var control: GameControlPanel
    private var adorned: AdornedBoard
            = AdornedBoard()
    private var history: MovesHistory
            = MovesHistory(game.board.history)

    init {
        add(history, BorderLayout.LINE_END)
    }
```

Вторичные конструкторы класса

```
open class Board {
    constructor(nV: Int, nH: Int) {
        // ...
    }
    constructor(board : Board) {
        // ...
    }
}
```

Вызов конструктора суперкласса

```
open class AsiaBoard : Board() {
    constructor(board : Board)
        : this(board.nV, board.nH)  {
        // ...
    }

    constructor(nV: Int, nH: Int)
        : super(board)  {
        // ...
    }
}
```

Свойства класса

```
class Board(var nV: Int = 0,
            var nH: Int = 0)
{
    var whitePlayer: IPlayer
        get() = players[PieceColor.WHITE]!!
        set(value) {
            players[PieceColor.WHITE] = value
            setBoardChanged()
        }
}
```

Реализация свойств, объявленных в интерфейсах

```
interface Move {
    val piece: Piece?
    // ...
}
interface ITransferMove : Move {
    val source: Square
    val target: Square
}
open class SimpleMove(
        override val source: Square,
        override val target: Square,
) : ITransferMove {
    override var piece: Piece? = null
}
```

Свойства с поздней инициализацией

- Свойства, объявленные как имеющие ненулевой тип, должны быть инициализированы в конструкторе.
- Если это неудобно, то может быть использован модификатор ***lateinit*** для более поздней инициализации.

```
public class Point {
    lateinit var x: Int
    lateinit var y: Int

    fun setup() {
        x = 1
        y = 1
    }
    fun dump() {
        if (x.isInitialized)
            println("x=$x")
    }
}
```

Обращение к полю из методов доступа

```
class User(val name : String) {
var address: String = "unspecified"
    set(value : String) {
        println("Address was changed to $value")
        field = value
    }
}
```

- в теле записи ***set*** для доступа к значению поля используется идентификатор **field**.
- в методе чтения ***get*** можно только прочитать значение, а в методе записи ***set*** - прочитать и изменить.
- для изменяемого свойства можно переопределить только один из методов доступа

Ограничение видимости свойств

- По умолчанию методы доступа имеют ту же видимость, что и свойство.
- Можно изменить видимость методы доступа добавив модификатор видимости перед ключевыми словами ***get*** и/или ***set***.
- Модифицировать свойство **moveColor** можно только внутри класса **Board**.

```
class Board(var nV: Int = 0, var nH: Int = 0) {
    /**
     * Цвет фигуры которая должна сделать ход.
     */
    var moveColor = PieceColor.WHITE
        private set
}
```

Внутренние и вложенные классы

- В Kotlin по умолчанию классы вложенные (как ***static*** - классы Java)
- вложенные классы не имеют доступа к экземпляру внешнего класса
- внутренние классы должны создаются с ключевым словом ***inner***

```
class Outer {
    inner class Inner {
        fun getOuterReference() : Outer
            = this@Outer
    }
}
```

Универсальные методы классов

- **toString()** - позволяет получить строковое представление экземпляра класса.
  По умолчанию это имя класса и адрес объекта в памяти.
- **equals()** - сравнение экземпляров классов.
  По умолчанию сравниваются адреса экземпляров в памяти.
- **hashCode()** - вместе с **equals** всегда должен переопределяться метод **hashCode**.
  Если два объекта равны, они должны иметь одинаковый хэш-код.
  Сравнение значений в **HashSet** оптимизировано: сначала сравниваются их хэш-коды, и только если они равны, сравниваются фактические значения.

Универсальные методы объектов-данных

```
data class Point(val x: Int, val y: Int)
```

Для объектов генерируются методы:
- **toString()** для создания строкового представления, показывающего все поля в порядке их объявления.
- **equals()** проверяет равенство значений всех свойств.
- **hashCode()** возвращает значение, зависящее от хэш-кодов всех свойств.

Методы **equals** и **hashCode** учитывают все свойства, объявленные в основном конструкторе. Свойства, не объявленные в основном конструкторе, не принимают участия в проверках равенства и вычислении хэш-кода.

Объекты - одиночки

```
object HomoSapience : IPlayer {
    override val name = "Homo sapience"
    override val authorName = "Романов"

    override fun doMove(board: Board, color: PieceColor) {}
    override fun toString() = name
}
```

- объявление объекта - это объявление класса и единственного экземпляра этого класса
- конструкторов нет

- имя класса и имя экземпляра совпадают
- возможно наследование от классов
- возможна реализация интерфейсов

Объекты - компаньоны

Могут использоваться для хранения статических свойств и методов:

```
class Square(
 // ...
 val vLetter: String
     get() = ALPHABET.substring(v, v + 1)

 companion object {
  private val ALPHABET
          = "abcdefghijklmnopqrstuvwxyz"
 }
}
```

Объекты - выражения

Могут использоваться для создания анонимных объектов:

```
class MoveLabel(kMove: Int,
                private val move: Move)
      : JLabel() {
init {
  val mListener = object : MouseListener {
    override fun mouseReleased(e: MouseEvent?) {}
    override fun mouseClicked(e: MouseEvent?) {}
    // ...
} }
```

- могут реализовывать неограниченное число интерфейсов.
- могут изменять переменные функций и классов в которых они расположены.

СИСТЕМА ТИПОВ ЯЗЫКА KOTLIN

Определение типа

- Тип данных - это множество значений и операций на этих значениях
- *Класс* и *интерфейс* являются *типами*
- Класс **String** - это класс и тип
- **List<String>** является типом, но не классом или интерфейсом

Поддержка значения *null*

- Система типов Kotlin поддерживает значения **null**
- Это дает возможность избегать *во время выполнения* программы исключения **NullPointerException**.
- Ошибки *времени выполнения* становаятся ошибками *времени компиляции*.

```
val a: String = null  // :( ошибка компиляции
val s: String? = null
val upper: String? = s.toUpperCase()
        // :( ошибка компиляции
```

- **String** - это класс и тип
- **String?** - это тип

Оператор безопасного вызова ?.

Для упрощения работы с типами принимающими значение **null** существует оператор **?.**

```
val a: String? = null
val b: String? = if (a != null) a.toUpperCase()
                 else null
val c: String? = b?.toUpperCase()

data class People(val name: String,
                  var parent: People? = null)

val man = People("Mark")
val parentName: String? = man.parent?.name
```

Оператор Элвис ?:

Оператор объединения со значением **null** (оператор *Элвис*) **?:** предназначен использования со значением **null**.

```
val a: String? = null
val b: String? = if (a != null) a.toUpperCase()
                 else null
val c: String = a?.toUpperCase() ?: ""

data class People(val name: String,
                  var parent: People? = null)

val man = People("Mark")
val parentName: String = man.parent?.name
                         ?: "Unknown"
```

Оператор Элвис ?: (проверка предусловий)

Оператор **?:** удобно использовать для проверки параметров функций на **null**:

```
override fun mousePressed(e: MouseEvent?) {
    e ?: return

    val s = getSquare(e) ?: return

    listener.mouseDown(s, e.button)
}
```

Проверка на null с помощью оператора !!

- Преобразует любое значение к типу, не поддерживающему значения **null**
- Если значение равно **null**, во время выполнения возникнет исключение.

```
override fun mousePressed(e: MouseEvent?) {
    val s = getSquare(e!!)!!

    listener.mouseDown(s, e.button)
}
```

Безопасное приведение типов: оператор as?

- При невозможности приведения типа оператор **as** выбрасывает ситуацию **ClassCastException**
- При невозможности приведения типа оператор **as?** возвращает результат **null**.

```
var p : Any?

val person = p as? Person ?: return false
```

Функция let

- **let** принимает в качестве параметра вызываемый объект и возвращает результат лямбда-выражения.
- Функция **let** предназначена для работы с *выражениями*, допускающими значение **null.**
- В одном коротком выражении позволяет вычислить выражение, проверить результат на **null** и сохранить его в переменной.
- Возможно использовать для передачи аргумента, который может оказаться равным **null**, в функцию, которая ожидает параметра, не равного **null**.

```
val numbers = listOf("one", "two", "three", "four")

val modifiedFirstItem = numbers.first().let {
    println("The first item of the list is '$it'")
    if (it.length >= 5) it else "!$it!"
}.toUpperCase()
```

Свойства с отложенной инициализацией

- Для всех **not null** свойств требуется инициализация в конструкторе
- Если свойство не может иметь значения **null** для инициализации необходимо указать значение.
- Если нет возможности предоставить такое значение,
  - приходится использовать тип с поддержкой **null**.
  - с помощью модификатора **lateinit** объявить, что свойство поддерживает *отложенную инициализацию*

```
class Sample {
    lateinit var name: String

    fun setup() {
        name = "Unknown"
    }
}
```

Расширение типов поддерживающих значение null

- Разрешить вызовы функций, где в роли получателя может быть тип допускающий **null**
- иметь дело со значением **null** *внутри этой функции*.
- Это возможно только для *функций-расширений*

```
fun String?.isNullOrBlank() : Boolean
    = this == null || this.isBlank()
```

Получатель - тип **String?**

Примитивные типы без поддержки null

- Пример: **Int, Boolean**
- Переменная *примитивного типа* содержит свое значение.
- Переменная *ссылочного типа* содержит ссылку на область памяти, где хранится объект.
- В языке Kotlin нет деления типов на *примитивные типы* и *обертки примитивных типов* как в Java

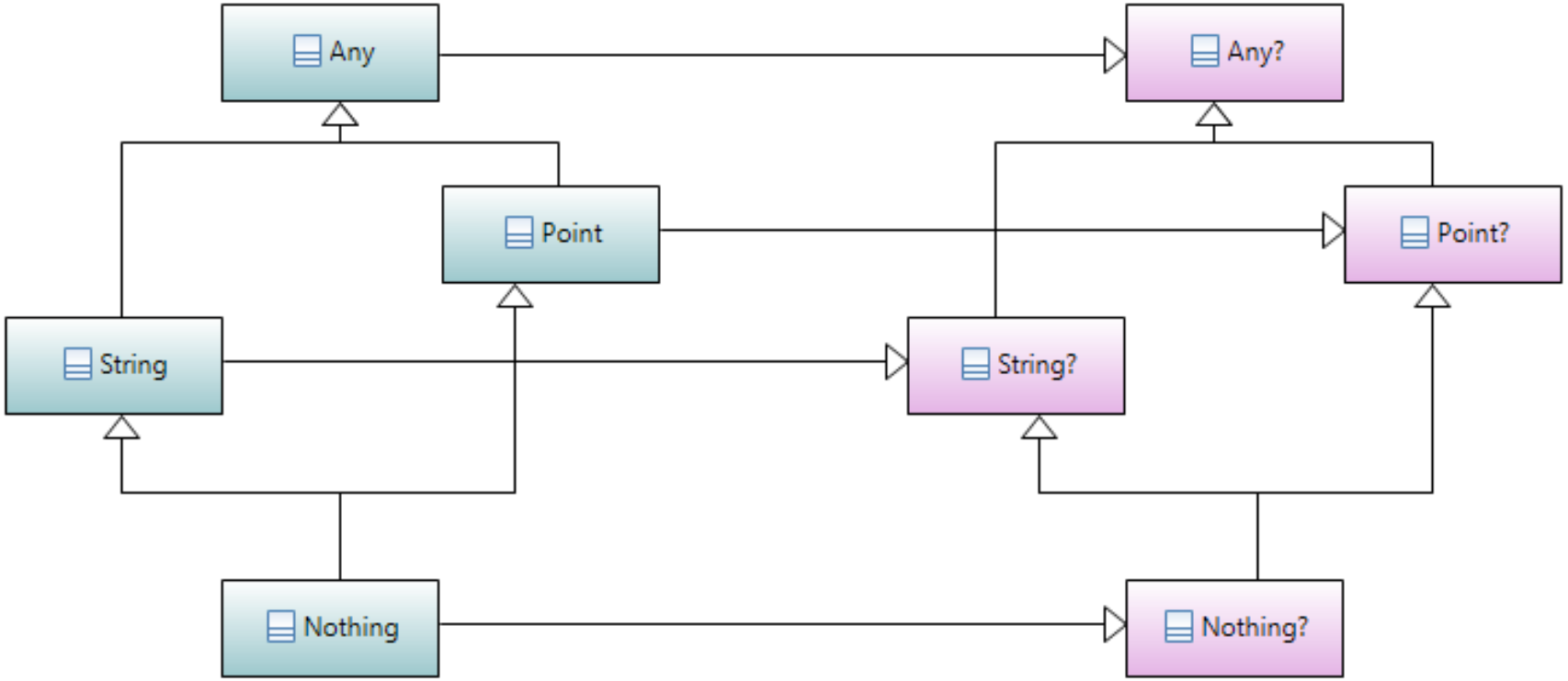
```
// Java
int i:  = 1
List<Integer> list = list0f(1, 2, 3)
// Kotlin
val i: Int = 1
val list: List<Int> = list0f(1, 2, 3)
```

Примитивные типы с поддержкой null

- Пример: **Int?**, **Boolean?**
- В языке Котлин для таких типов всегда используется класс-обертка. Например (**Integer** для **Int**)
- При сравнении двух значений примитивных типов с поддержкой **null** (например, **Int?**) выполняется проверка, что оба значения не равны **null**. После после этого компилятор работает с ними как обычно.
- JVM не поддерживает использования примитивных типов в качестве аргументов типа,
- Поэтому обобщенные классы Kotlin всегда должны использовать типы-обертки (**Integer** для **Int**).

```
val i: Int? = 1
val list: List<Int?> = list0f(1, 2, 3)
```

Иерархия типов языка Kotlin



Иерархия типов языка Kotlin

Типы и подтипы, классы и подклассы

- Тип переменной определяет её возможные значения
- Тип **B** - это подтип типа **A**, если значение типа **B** можно использовать везде, где ожидается значение типа **A**
- Обозначение отношения тип-подтип **A <: B**
- Класс **B** - это подтип класса **A**, если класс **B** - это подкласс класса **A**
- Класс **B** - это подтип интерфейса **A** если класс **B** - реализует интерфейс **A**

Корневые типы Any и Any?

- В Kotlin тип **Any** предок всех типов принимающих не нулевые значения, также и для примитивных типов (например **Int**).
- Тип **Any** не поддерживает значения **null**, поэтому переменная типа **Any** не может хранить **null**.
- Для хранения любого допустимого значение в Kotlin (и **null** тоже) требуется тип **Any?**.
- Все классы в Kotlin имеют три метода наследуемые от **Any**: **toString**, **equals** и **hashCode**
- Присваивание примитивного значения переменной типа **Any** вызывает автоматическую упаковку значения

```
val size: Any = 11
```

Приведение типов nullable и not nulable (Any и Any?)

- Тип **Any** подтип типа **Any?**
- Обозначение **Any? <: Any**

```
val any: Any = ""
val anyQ: Any? = ""

val any1: Any = anyQ // :( Ошибка компиляции
val anyQ1: Any? = any
```

Объявление класса Any

```
public open class Any {
    public open operator
    fun equals(other: Any?): Boolean

    public open
    fun hashCode(): Int

    public open
    fun toString(): String
}
```

Класса **Any?** не существует

Тип Unit для отсутствующего значения

- Тип **Unit** предназначен для отсутствующего возвращемого значения функции

```
public interface Union
      extends Annotated, SimpleTypeHost, TypedXmlWriter {
 // ...
}
```

- Возвращается *неявно*
- **return Unit** добавляется компилятором *неявно*.

```
fun f1() : Unit {
}
fun f2() {
}
```

Тип Nothing

- В языке Kotlin существует специальный тип возвращаемого значения **Nothing**

```
public class Nothing private constructor()
```

- Тип **Nothing** не имеет значений и используется как тип возвращаемого значения функции
- Компилятору сообщается что функция с таким типом не вернет управления
- Эта информация используется при анализе кода вызова функции.

```
public inline fun TODO(): Nothing
    = throw NotImplementedError()
```

Nullable типы как параметры родовых типов

```
fun dumpList(names: List<String?>) {
    val notNulls: List<String>
        = names.filterNotNull()
    notNulls.forEach(::println)
}
```

Изменяемые и неизменяемые коллекции

Интерфейсы для изменяемых (**MutableCollection**) коллекций и неизменяемых (**Collection**) коллекций отделены

```
public interface Collection<out E>
        : Iterable<E> {
    public val size: Int
    public fun isEmpty(): Boolean
    override fun iterator(): Iterator<E>
    // ...
}
public interface MutableCollection<E>
        : Collection<E>, MutableIterable<E> {
    override fun iterator(): MutableIterator<E>
    public fun add(element: E): Boolean
    public fun remove(element: E): Boolean
    // ...
}
```

ОБОБЩЕННЫЕ ТИПЫ (GENERIC)

Параметризованные типы

- В языке Kotlin имеется возможность определять *параметризованные типы*.
- Также используются термины *обобщенные типы* и *родовые типы*
- При создании экземпляра такого типа, параметр типа заменяется конкретным типом, который называется аргументом типа.

```
class Box<T>(t: T) {
    var value = t
}

val box1: Box<Int> = Box<Int>(1)
val box2: Box<Int> = Box(1)
```

Параметризованные классы

```
class MutableStack<E>(vararg items: E) {
  private val elements = items.toMutableList()
  fun push(element: E) = elements.add(element)
  fun peek(): E = elements.last()
  fun pop(): E = elements.removeAt(elements.size-1)
}

val stringStack1: MutableStack<String>
    = MutableStack<String>("one", "two", "three")

val stringStack2: MutableStack<Int>
    = MutableStack(1, 2, 3)
```

Параметризованные интерфейсы

```
interface Comparable<T> {
    fun compareTo(other: T) : Int
}

class String : Comparable<String> {
  override fun compareTo(other: String) : Int
          = // ...
}
```

Параметризованные функции

- Параметр функции после ключевого слова ***fun***:
- Тип параметра **E** будет определен во время подстановки

```
fun <E> mutableStackOf(vararg elements: E)
        = MutableStack(*elements)

fun main() {
  val stack = mutableStackOf(0.62, 3.14, 2.7)
  println(stack)
}
```

Функции с параметризованным получателем

- Параметр функции после ключевого слова ***fun***:
- Тип получателя **T** будет определен во время подстановки

```
public inline
fun <T> T.takeIf(predicate: (T) -> Boolean): T? {
    return if (predicate(this)) this else null
}

var k = 1
val t = k.takeIf{ it >= 0} ?: 0
```

Ограничение типовых параметров - класс

- Ограничение типового параметра **<T : Number>**
- **T** типовой параметр
- **Number** - граница типового параметра
- Типовым параметром может быть класс **Number** и его подтипы (подклассы)

```
fun <T : Number> List<T>.sum() : T
    = this.sum()

list0f(1, 2, 3).sum()
list0f(1.5, 2.5, 3.5).sum()
```

Ограничение типовых параметров - интерфейс

- Ограничение типового параметра **<T : Comparable<T>>**
- **T** типовой параметр
- **Comparable<T>** - граница типового параметра
- Типовым параметром может быть классы реализующие интерфейс **Comparable**

```
fun <T: Comparable<T>> max(first: T, second: T): T
    = if (first > second) first else second

val number: Int = max(1, 2)
val string: String = max("Атлантида",
                         "Антарктида")
```

Несколько ограничений типовых параметров

- С помощью конструкции **when** можно описать несколько ограничений на типовые параметр

```
fun <T> max(first : T, second: T) : T
    when T : Comparable<T>,
         T : Number
    = if (first > second) first else second

val number: Int = max(1, 2)
```

Запрет использования null типов в качестве типовых параметров

- Ограничение - типовым параметром может быть любой потомок класса **Any**
- Потомки классов принимающих значения **null** (потомки **Any?**) недопустимы

```
class SomeClass<T: Any> {
}
```

Обобщенные типы во время выполнения

- Обобщенные типы в Kotlin *не хранятся* во время выполнения.
- Экземпляр обобщенного класса *не хранит* информацию о типовых аргументах, использованных для создания этого экземпляра
- невозможно проверить что список состоит из строк, а не целых чисел

```
var list: Any = listOf("")

if (list is List<String>)
    // :( Cannot check for instance
    // of erased type: List<String>
    // ...
```

Проверка и приведение для обобщенных типов

```
var c: Any = listOf("")

if (c is List<*>)
    println(c)

val list: List<Int>? = c as? List<Int>
```

Обобщенные типы в функциях во время выполнения

- для функций проверка обобщенного типа *не возможна*

```
fun <T> isA(value : Any)
    = value is T
    // :( Error : Cannot check for instance
    // of erased
```

Обобщенные типы во встраиваемых функциях во время выполнения

- для встраиваемых (**inline**) функций проверка обобщенного типа *возможна*
- типовой параметр необходимо пометить как овеществляемый (**reified**)

```
inline
fun <reified T> isA(value : Any) = value is T
// Код компилируется, проверка выполняется
```

Типы и подтипы

- Тип переменной определяет её возможные значения
- Тип **B** - это подтип типа **A**, если значение типа **B** можно использовать везде, где ожидается значение типа **A**
- Обозначение отношения тип-подтип **A <: B**

Вариантность типов (variance)

- Термин ***вариантность (variance)*** описывает, как связаны между собой типы с одним базовым типом и разными типовыми аргументами.
- При присваиваивании значения переменной или передаче аргумента при вызове функции, компилятор проверяет наличие отношения тип-подтип
- Обобщенный класс **C** называют ***инвариантным*** по типовому параметру, если для любых разных типов **A** и **B** тип **C<A>** не является подтипом или супертипом **C<B>**.

Ковариантность

- Если **C<T>** является универсальным типом с параметром типа **T,** а **U** является подтипом **T**, тогда **C<U>** является подтипом **C<T>**
- Если **T <: U** то **C<T> <: C<U>**
- *Направление отношения тип - подтип сохраняется*
- Пример: **List<Int>** является подтипом **List<Number>,** потому что **Int** является подтипом **Number.**

Классы и интерфейсы ковариантные по типовому параметру

Этот интерфейс **Producer** объявлен ковариантным по типовому параметру **T** с помощью ключевого слова ***out***

```
interface Producer<out T> {
  fun produce() : T
}
```

- Объявление типа *ковариантным по типовому параметру* ограничивает возможные способы использования этого параметра в типе.
- Он может использоваться только в *исходящих позициях* (**out**): то есть тип может *производить (produce)* значения типа **T**, но *не потреблять (consume)* их.

Особености ковариантных типовых параметров

- Ключевое слово ***out*** применяется к типам, которые являются ***производителями*** или ***источниками*** значения типа **T**
- **T** появляется только в позиции ***out,***
  (в качестве возвращаемого типа функции).
- Разрешается *возвращать значения* этого типа из функций, когда типовой аргумент неточно соответствует типу параметра в определении функции

- *Параметры конструктора* не находятся ни во входящей, ни в исходящей позиции.
- Если *параметр типа* объявить как ***out***, можно использовать его в объявлениях параметров конструктора.
- Параметры приватных (***private***) методов *не находятся* ни в исходящей, ни во входящей позициях

Пример использования ковариантных типов

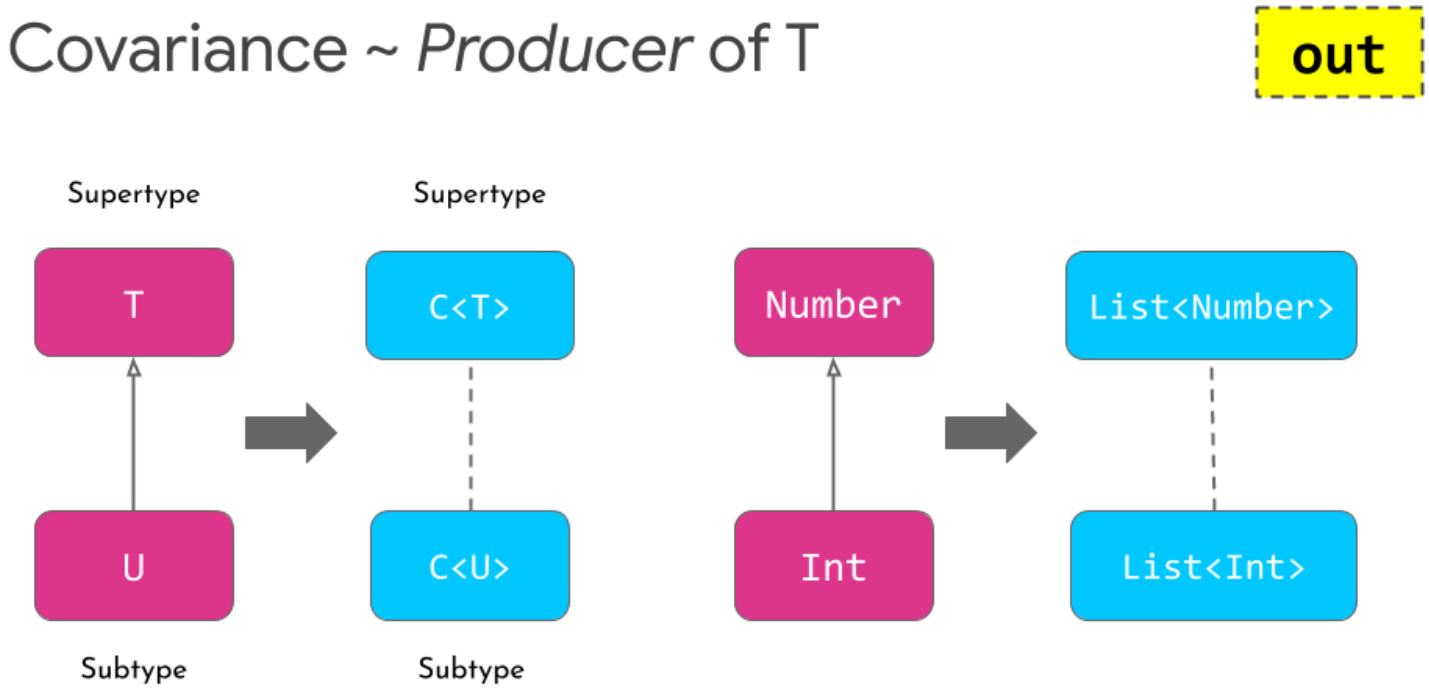
```
class CovarianceSample<out T>

fun main1() {
    val firstSample: CovarianceSample<Any>
            = CovarianceSample<Int>()
    // :) Int подтип типа Any

    val secondSample: CovarianceSample<Any>
            = CovarianceSample<String>()
    // :) String подтип типа Any

    val thirdSample: CovarianceSample<String>
            = CovarianceSample<Any>()
    // :( Несоместимость типов
}
```

Ковариантность (диаграмма)



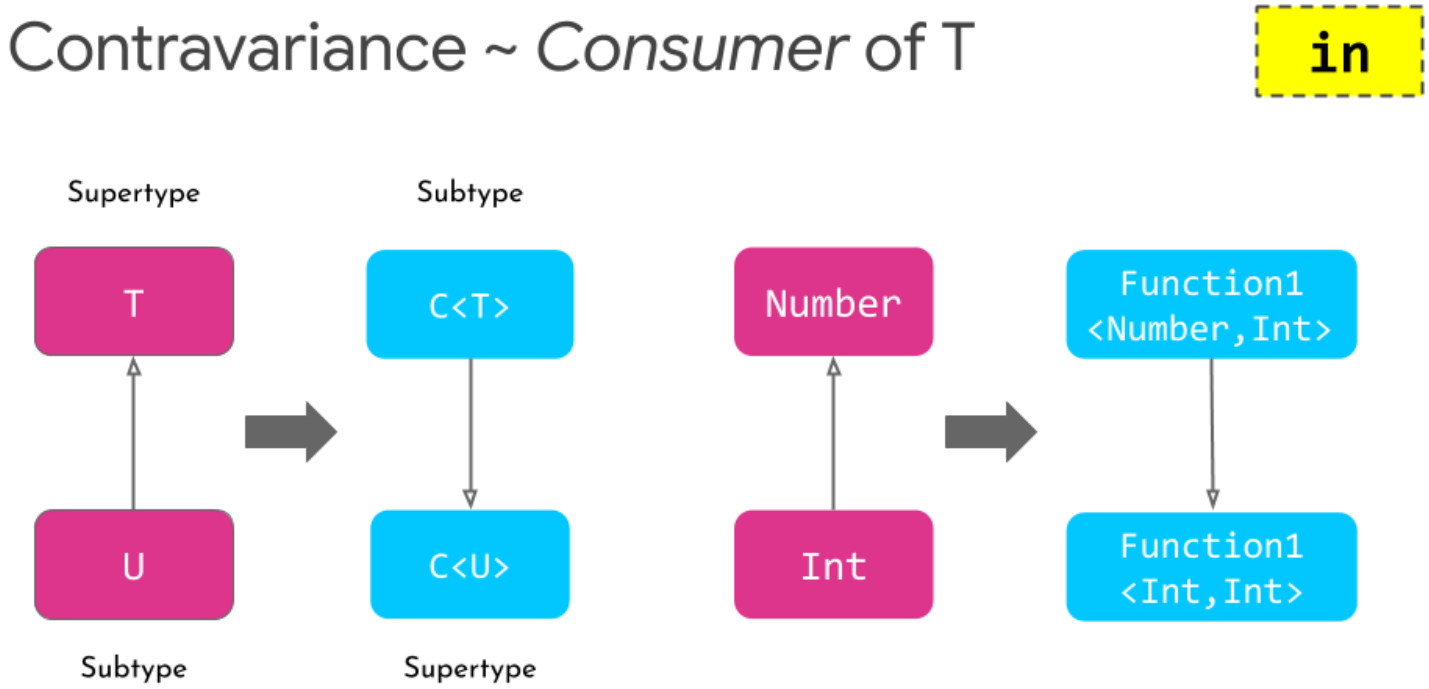
ClassName<out LowerBound>

Контрвариантность

- Если **C<T>** является универсальным типом с параметром типа **T**,
  а **U** - подтипом **T**, то **C<T>** является подтипом **C<U>**

- Если **U <: T** то **C<U> :> C<T>**
- *Направление отношения тип - подтип меняется*
- Пример: **Function1<Number, Int>** является подтипом **Function1<Int, Int>,**
  поскольку **Int** является подтипом **Number.**
- Применяется к типам, которые являются ***потребителями*** типа **T**
- **T** появляется только в позиции ***in***,
  то есть в типе аргумента функции.

Контрвариантность (диаграмма)



ClassName<in UpperBound>

Пример контрвариантности

- В интерфейсе **Comparator** оба параметра функции **compare** находятся в позиции ***in***

```
interface Comparator<in T> {
  fun compare(e1: T, e2: T) : Int
}
```

- Для сравнения объектов *конкретного типа* можно использовать реализацию **Comparator** для этого *конкретного типа* или для любого *супертипа конкретного типа*

```
val anyComparator = Comparator<Any> {
    e1, e2 -> e1.hashCode() - e2.hashCode()
}
val strings: List<String> = listOf("A", "B")
strings.sortedWith(anyComparator)
```

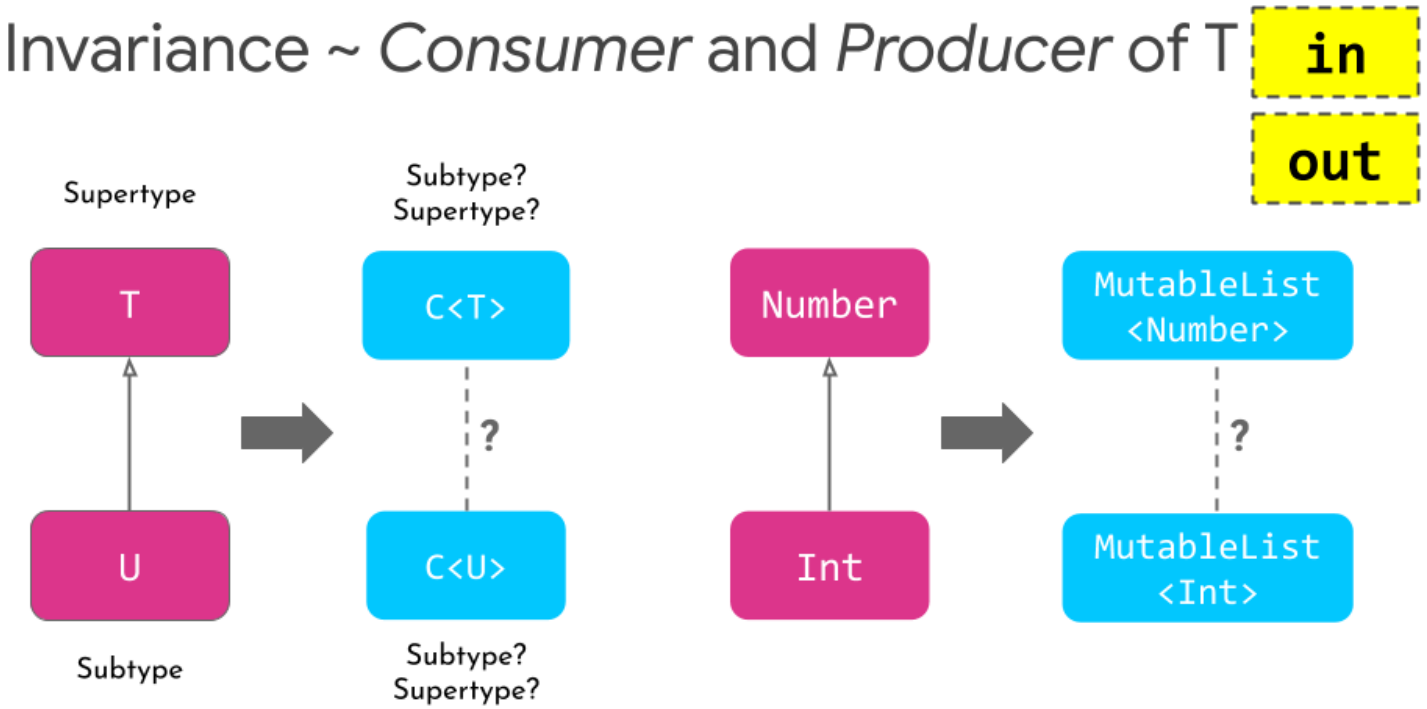
Инвариантность

- Если **C<T>** является подтипом **C<U>**, то **T** = **U**
- Пример: **Array<T>** инвариантен в **T**
- **T** появляется как в позиции ***in***, так и в позиции ***out***.
- Тип является производителем и потребителем **T**
- *Лямбда-выражения контравариантны по типам аргументов и ковариантны по типу возвращаемого значения*.

Ковариантные, контравариантные и инвариантные классы

Ковариантные	Контравариантные	Инвариантные
Producer<out T>	**Consumer<in T>**	**MutableList<T>**
T только в *исходящих* позициях ***out***	**T** только во *входящих* позициях ***in***	**T** в любых позициях
Направление отношения тип-подтип *сохраняется*	Направление отношения тип-подтип *меняется на обратное*	Нет отношения тип-подтип
Int :> Number	**Int :> Number**	_
Producer<Int> :> **Producer<Number>**	**Consumer<Int>** <: **Consumer<Number>**	_

Инвариантность (диаграмма)



pic

Определение вариантности в месте использования

- Если модификаторы вариантности указаны в объявлениях типов то указанные модификаторы применяются везде, где используется тип.
- Это определение вариантности *в месте объявления*.
- Возможно определение вариантности *в месте использования* для конкретного вхождения типового параметра.
- Такое определение типа в месте использования называется *проекцией типа (type projection)*
- Объявление **MutableList<out T>** в **Kotlin** означает то же самое, что **MutableList<? extends T>** в **Java**
- Объявление **MutableList<in T>** в **Kotlin** означает то же самое, что **MutableList<? super T>** в **Java**

Пример ковариантности в месте использования

```
class CovarianceSample<T>

fun main() {
  val firstSample: CovarianceSample<Any>
      = CovarianceSample<Int>()
      // :( Type mismatch

  val secondSample: CovarianceSample<out Any>
      = CovarianceSample<String>()
      // :) String is a subtype of Any

  val thirdSample: CovarianceSample<out String>
      = CovarianceSample<Any>()
      // :( Type mismatch
}
```

Проекция со звездочкой : использование * вместо типового аргумента

- Список элементов *неизвестного типа*: **MutableList<*>**
- Тип **MutableList<Any?>** определяет список, содержащий элементы *любого типа*.
- Тип **MutableList<*>** определяет список, содержащий элементы *конкретного неизвестного типа*
- Синтаксис проекций со звездочкой используется, когда информация о типовом аргументе не имеет никакого значения, то есть когда ваш код не использует методов, ссылающихся на типовои параметр в сигнатуре, или только читает данные без учета их типа

```
fun printFirst(list: List<*>) {
  if (list.isNotEmpty()) {
      println(list.first())
  }
}
```

ДЕЛЕГИРОВАНИЕ В ЯЗЫКЕ KOTLIN

Делегирование класса

- ***Шаблон делегирования*** - альтернатива наследованию реализации
- Класс **Derived** может реализовать интерфейс **Base**, делегируя все свои публичные члены указанному в параметре объекту
- Предложение **by** в списке супертипов для **Derived** указывает, что **b** будет храниться внутри в объектах **Derived**,
- Компилятор сгенерирует все методы **Base**, которые направляют на **b**.

```
interface Base {
    fun print()
}
class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}
class Derived(b: Base) : Base by b
fun main(args: Array<String>) {
    val b = BaseImpl(10)
    Derived(b).print() // prints 10
}
```

Переопределение члена интерфейса, реализованного делегированием

```
interface Base {
    fun printMessage()
    fun printMessageLine()
}
class BaseImpl(val x: Int) : Base {
    override fun printMessage() { print(x) }
    override fun printMessageLine() { println(x) }
}
class Derived(b: Base) : Base by b {
    override fun printMessage() { print("abc") }
}
fun main() {
    val b = BaseImpl(10)
    Derived(b).printMessageLine() // 10
    Derived(b).printMessage() // abc
}
```

Доступ к реализациям членов интерфейса

```
interface Base {
    val message: String
    fun print()
}
class BaseImpl(val x: Int) : Base {
    override val message = "BaseImpl: x = $x"
    override fun print() { println(message) }
}
class Derived(b: Base) : Base by b {
// Это свойство недоступно из реализации b `print`
    override val message = "Message of Derived"
}
fun main() {
    val b = BaseImpl(10)
    val derived = Derived(b)
    derived.print() // BaseImpl: x = 10
    println(derived.message) // Message of Derived
}
```

Делегирование свойств

```
class Example {
  var p: String by Delegate1()
}

fun main() {
    var e = Example()
    e.p = "Hello"
    println(e.p);
}
```

Синтаксис делегируемых свойств:

val/var <имя свойства>: <Тип> by <выражение>

- Выражение после **by** — делегат:
- обращения **get()**, **set()** к свойству будут обрабатываться этим выражением.
- Делегат не обязан реализовывать какой-то интерфейс. Достаточно, чтобы у него были методы **getValue()** и **setValue()** с определённой сигнатурой.

Сигнатура делегата

```
class Delegate1 {
operator
fun getValue(thisRef: Any?,
             property: KProperty<*>): String
{
  return "$thisRef, спасибо за делегирование мне"
   + " '${property.name}'!"
}
operator
fun setValue(thisRef: Any?,
             property: KProperty<*>,
             value: String)
{
  println("$value было присвоено значению " +
  "'${property.name} в $thisRef.'" )
}
}
```

Действия делегата

```
class Delegate1 {
operator fun getValue(thisRef: Any?,
            property: KProperty<*>): String {
 return "$thisRef, спасибо за делегирование мне"+
        " '${property.name}'!"
}
operator fun setValue(thisRef: Any?,
            property: KProperty<*>,
            value: String) {
  println("$value было присвоено значению " +
         "'${property.name} в $thisRef.'" )
}
}
// Hello было присвоено значению 'p в
delegates_prop.Example@433c675d.'
// delegates_prop.Example@433c675d, спасибо за делегирование
мне 'p'!
```

Ленивые свойства (lazy properties)

- **lazy** функция, которая принимает лямбду и возвращает экземпляр класса **Lazy<T>**
- экземпляр - делегат для реализации ленивого свойства
- первый вызов **get()** запускает лямбда-выражение, переданное **lazy()** в качестве аргумента, и запоминает полученное значение
- последующие вызовы просто возвращают вычисленное значение

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}
fun main(args: Array<String>) {
    println(lazyValue)
    println(lazyValue)
}
//computed!
//Hello
//Hello
```

Ленивые свойства. Функция lazy

```
public actual
fun <T> lazy(initializer: () -> T): Lazy<T>
    = SynchronizedLazyImpl(initializer)

public interface Lazy<out T> {
    public val value: T

    public fun isInitialized(): Boolean
}
```

Обозреваемые свойства. Observable (1)

- Функция **Delegates.observable()** принимает два аргумента:
- начальное значение свойства
- обработчик (лямбда), который вызывается *после* изменения свойства.
- У обработчика три параметра: описание свойства, которое изменяется, старое значение и новое значение.

```
class User {
var name: String
  by Delegates.observable("<no name>") {
  prop, old, new -> println("$prop: $old -> $new")
}}
fun main(args: Array<String>) {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

Обозреваемые свойства. Observable (2)

```
class User {
var name: String
  by Delegates.observable("<no name>") {
  prop, old, new -> println("$prop: $old -> $new")
}}
fun main(args: Array<String>) {
    val user = User()
    user.name = "first"
    user.name = "second"
}
// var delegates_prop.observable.User.name: kotlin.String: <no
name> -> first
// var delegates_prop.observable.User.name: kotlin.String: first ->
second
```

Хранение свойств в ассоциативном списке

- Хранение свойств в ассоциативном списке.
- В этом примере конструктор принимает ассоциативный список
- Делегированные свойства берут значения из этого ассоциативного списка (по строковым ключам)

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))
println(user.name) // Prints "John Doe"
println(user.age)  // Prints 25
```