

Тема 2. Знакомство с объектно-ориентированными возможностями языка Kotlin

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

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**
- множеств 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

Особенности 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 метода 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* становятся 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 = listOf(1, 2, 3)
```

// Kotlin

```
val i: Int = 1
```

```
val list: List<Int> = listOf(1, 2, 3)
```

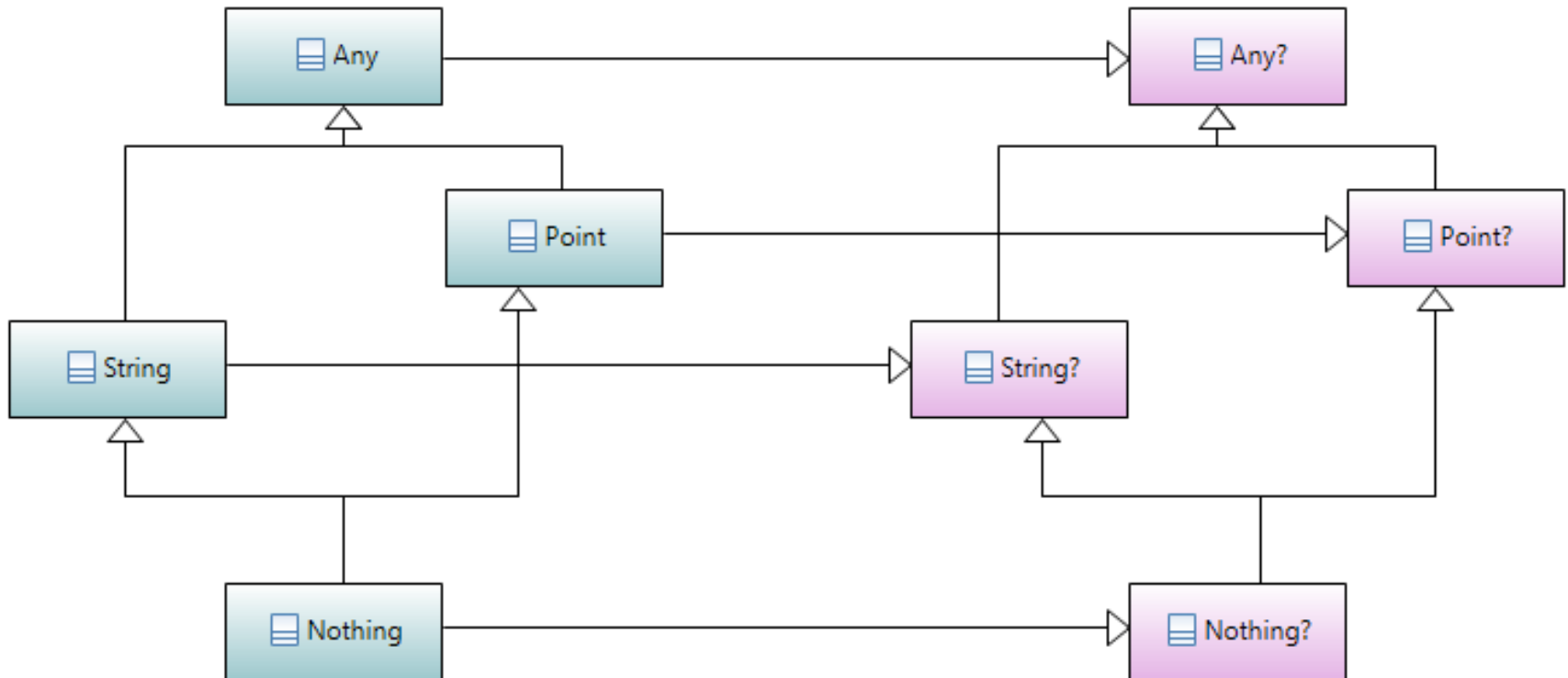
Примитивные типы с поддержкой null

- 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 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?> = listOf(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 значения 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 nullable (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 возвращаемого 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.takelf(predicate: (T) -> Boolean): T? {  
    return if (predicate(this)) this else null  
}
```

```
var k = 1
```

```
val t = k.takelf{ 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()
```

```
listOf(1, 2, 3).sum()
```

```
listOf(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 присваивании 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**.

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.

Особенности 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

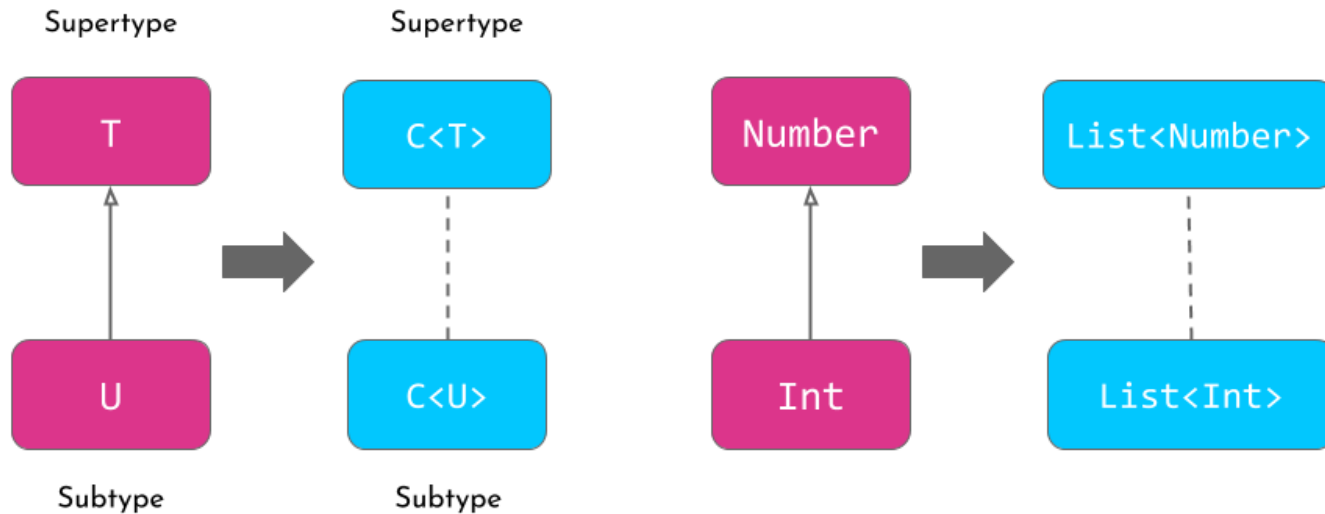
```
class CovarianceSample<out T>
```

```
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>()  
    // :( Несовместимость типов  
}
```


Ковариантность (диаграмма)

Covariance ~ *Producer* of T

out



ClassName<out LowerBound>

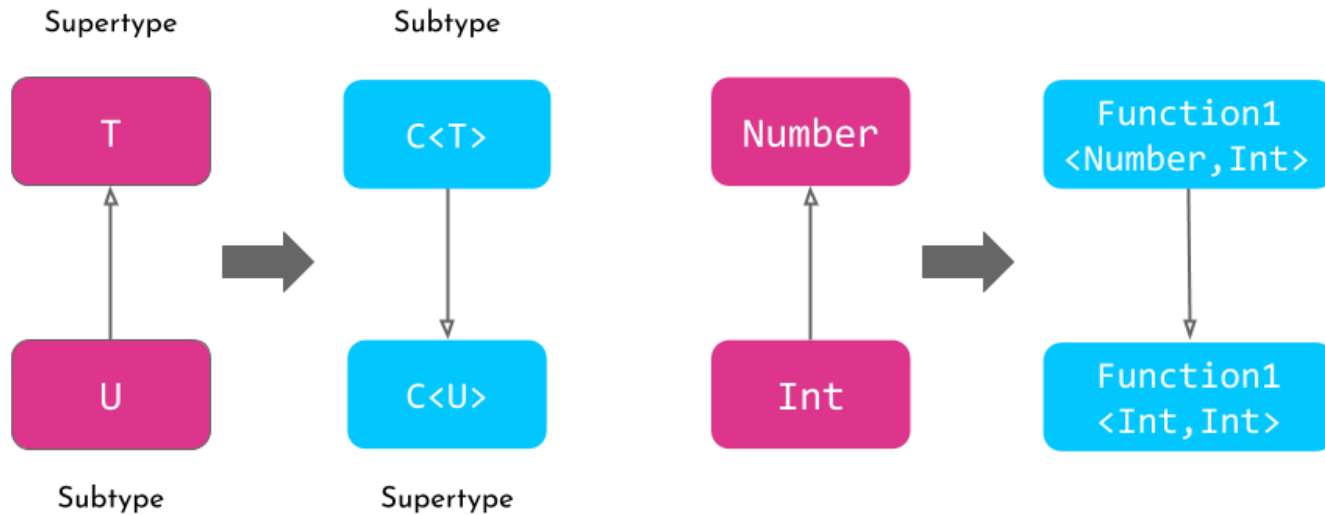
Контрвариантность

- 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)

Contravariance ~ *Consumer* of T

in



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 *контрвариантны по типам аргументов и ковариантны по типу возвращаемого значения.*

Ковариантные, контравариантные и инвариантные классы

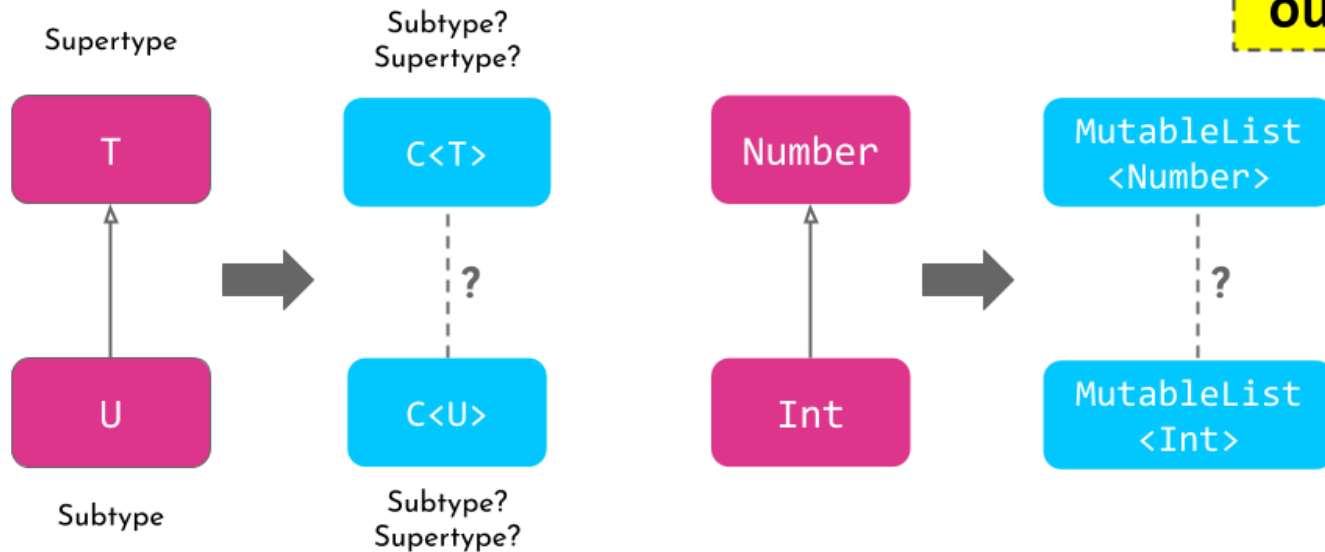
Ковариантные	Контвариантные	Инвариантные
Producer<out T>	Consumer<in T>	MutableList<T>
T только в <i>исходящих</i> позициях out	T только во <i>входящих</i> позициях in	T в любых позициях
Направление отношения тип-подтип <i>сохраняется</i>	Направление отношения тип-подтип <i>меняется на обратное</i>	Нет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

Инвариантность (диаграмма)

Invariance ~ *Consumer* and *Producer* of T

in

out



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 типовой параметр в сигнатуре, или только читает данные без учета их типа

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