# Программирование на языке Kotlin
### для любопытных

Kotlin объектно-ориентированный со статической типизацией компилируемый язык программирования. Имеется также возможность программирования в функциональном стиле. Язык кроссплатформенный. Имеются три версии Kotlin: Kotlin/JVM, Kotlin/JS и Kotlin/Native (обычный Kotlin), включающие язык, основные библиотеки и базовые инструменты. Пока мы ограничимся компилятором kotlinc-native, позволяющим создавать рабочий exe-файл без использования виртуальной машины Java.

## 1.Базовый синтаксис

В этой главе рассмотрим основные конструкции языка кратко, только для общего знакомства для того, чтобы облегчить дальнейшую работу. Возможно, что неискушённому читателю здесь кое-что будет непонятно; надо пока это просто пропустить, дальше мы всё разберём детально.

По традиции начнём с программы приветствия, которая на Kotlin предельно простая:

**fun main() { println("Привет, Мир! ") }** → *Привет, Мир!*

Идентификатор скрипта на Kotlin должен иметь расширение **.kt**. Поместим этот код, например, в файл **prog.kt**, расположенный там, где нам удобно. При работе из командной строки надо перейти в ту папку, где расположен файл, и выполнить команду:

**kotllinc-native prog.kt -o prog.exe** (расширение .exe здесь можно опустить).

В той же папке будет создан файл **prog.exe** (если нравится, название рабочего файла можно задать другим). К сожалению, компиляция на Kotlin выполняется очень медленно, поскольку язык реализован на Java, поэтому будьте готовы к бесполезной трате времени при тестировании примеров. Команда **prog.exe** или просто **prog** запустит файл на выполнение и мы получим результат *Привет, Мир!*.

Выводимые на терминал результаты я дальше буду записывать в тексте программы после стрелки (→), заменяющей слова «получим»,

«будет выведено» и тому подобное. При копировании кода для выполнения на своей машине эти стрелки и текст после них надо убрать. Отмечу сразу, что в текст программы можно вставлять комментарии с двумя прямыми слешами (//) вначале строк. Сам я нигде не буду применять эти комментарии, так как считаю их лишними в коротких учебных программах. В документации Kotlin имеется подробное описание рекомендаций по форматированию кода программ, но фактически эти рекомендации необязательны и компилятором форматирование не учитывается. Я лично считаю, что главное достоинство кода заключается в его краткости и к этому надо стремиться всегда и везде. Например, если тело функции или класса можно разместить на одной строке, то это надо делать всегда. Короткие выражения лучше располагать на одной строке, разделяя их точкой с запятой. Вместо применения «верблюжьих» идентификаторов делать их максимально краткими и так далее.

Пример показывает, что:

- в программе на Kotlin должна быть объявлена функция **main**, определяющая точку входа.
- функции объявляются со служебным словом **fun**.
- тело функции заключается в фигурные скобки.
- в конце строк не требуется (хотя и допустимо) ставить точку с запятой.
- для вывода на терминал применяется стандартная функция **println**, которая выполняет перевод строки. Есть также и функция **print** не выполняющая перехода на новую строку.

Рассмотрим ещё несколько примеров для того, чтобы освоить некоторые приёмы, которые нам понадобятся в дальнейшем. Несколько изменим программу приветствия:

**fun main(m: Array<String>) { println("Привет, " + m[0] + "!")** }

Здесь функция **main** имеет аргумент, представляющий массив **m** (обычно этот массив называют **args**, но это несущественно) с элементами типа **String**; этот тип представляет строки (обычно текст в кавычках). Добавляемую к названию типа информацию в угловых скобках называют параметром типа, но о них позже. Элементы массива получат значения, которые можно задать при вызове рабочей программы. Например, мы можем сделать такой вызов:

**prog Маша Катя**

Кавычек здесь не надо применять, эти значения всегда имеют тип String по умолчанию. Получим результат: *Привет, Маша!* Мы использовали только первый элемент массива (индексация начинается с нуля). Если **m[0]** заменить на **m[1]** получим *Привет, Катя!*. Можем также видеть, что в Kotlin в качестве знака конкатенации принят знак суммирования (+).

Программы всех примеров я буду записывать в один и тот же файл **prog.kt**, стирая предыдущие примеры. Советую также поступать и читателям, так как хранить на диске эти маленькие программы не имеет смысла, а для выполнения их на своей машине код легко скопировать из этого руководства.

Аргумент оператора **println** должен представлять строку (тип **String**), но сам массив **m** имеет тип не **String**, а **Array<String>** и не может быть передан оператору **println** непосредственно. Автоматическая конвертация типов здесь не выполняется. Если мы хотим вывести на экран весь список имён, то для приведения типа **Array<String>** к типу **String** надо применить стандартную функцию (метод) **contentToString()**:

```
fun main(m: Array<String>) {
   println(m.contentToString())
}
```

Здесь использован вызов метода в так называемой точечной нотации: аргумент — точка — функция. Запустим программу:

**prog Виктор Борис** → *[Виктор, Борис]*

Отмечу попутно, что названия типов записываются через двоеточие после идентификатора и всегда с заглавной буквы.

Запрограммируем теперь диалог:

```
fun main() {
   print("Как вас зовут? ")
   val x = readln();
   println("Привет, " + x + "!")
}
```

Стандартная функция **readln** позволяет вводить текст с клавиатуры. В программе объявлена переменная **x**, а для её объявления использовано служебное слово **val**, которое указывает, что переменная неизменяемая (*immutable*), дальше в программе ей нельзя присвоить новое значение. Изменяемые (*mutable*) переменные объявляются со словом **var**. Переменная **x** инициируется тем текстом,

который мы напечатаем на клавиатуре. Здесь мы не объявляли тип переменной **x**, и это значит, что он будет выведен (*derive*) компилятором из контекста. Поскольку функция **readln** всегда возвращает текст, то переменная **x** получит тип **String**.

После запуска программы на экране появится вопрос *Как вас зовут?* Если мы ответим *Victor Borisov*, то получим результат: *Привет, Victor Borisov!*

При вводе имени я использовал латиницу, поскольку с кириллицей при вводе с клавиатуры тут может иметь место проблема. Пока не будем с этим разбираться.

Использованную в программе переменную я обозначил одной буквой **x**. Эта переменная предназначена для хранения имени и обычно в руководствах рекомендуют применять в таких случаях содержательные идентификаторы, в данном случае лучше всего подходит слово **name** (Kotlin позволяет везде применять кириллицу и вообще-то здесь можно использовать и русское слово **имя**). Но в коротких учебных программах это несущественно и содержательные имена только загромождают код, не принося никакой пользы. Считаю, что при изучении языка надо применять максимально краткий текст программ, в котором легче всего понять суть рассматриваемой задачи и не отвлекаться на несущественные детали. Да и в реальных программах содержательные имена ничего не дают в тех случаях, когда ими называются объекты, жизнь которых ограничена всего несколькими строками текста. Этого правила я дальше и буду всегда придерживаться, так как очень не люблю длинных идентификаторов.

### Функции

Рассмотрим теперь создание и использование функций более подробно. Начнём с такого примера:

```
fun f(x: Int, y: Int): Int { return x + y }
fun main() { print(f(3, 4)) }
```
→ *7*

Здесь объявлена функция **f**, принимающая два аргумента **x** и **y** типа **Int** и возвращающая результат также типа **Int**. Указывать тип аргументов в Kotlin требуется всегда. Можно не указывать тип результата, но тогда он будет принят **Unit** по умолчанию. Этот тип означает отсутствие результата, в других языках обычно это тип **Void**. Функция **main**, с которой начинается работа программы, вызывает

функцию **f**, передаёт ей значения аргументов **x = 3**, **y = 4** и выводит результат на экран. Оператор **return** возвращает результат вычисления суммы. Отметим ещё, что порядок объявления функций в Kotlin не имеет значения, то-есть, функцию **main** можно было объявить первой.

Есть возможность написать эту же программу в другом виде:

```
fun f(x: Int, y: Int): Int = x + y
fun main() { println("Сумма 9 и 5 равна ${f(9, 5)}") } →
```

*Сумма 9 и 5 равна 14*

Если тело функции представлено только одним выражением, то допустима форма с использованием знака равенства. При этом выражение не надо заключать в фигурные скобки. Здесь не обязательно объявлять тип результата явно, так как в этой форме Kotlin способен вывести (derive) тип из контекста: если **x** и **y** представлены целыми числами, то и их сумма будет целым числом (тип **Int**).

При выводе не экран использована интерполяция: внутри текста, заключённого в двойные кавычки, блок после знака **$** вычисляется, результат приводится к типу **String** и вставляется в текст.

Некоторые функции не возвращают никакого результата, тогда указывается тип **Unit**:

```
fun f(x: Int, y: Int): Unit { println("Сумма x и y равна ${x + y}") }
fun main() { f(-1, 8) } → Сумма x и y равна 7
```

Здесь функция **f** только выводит текст на экран и ничего не возвращает. Явно этот тип можно не указывать, он всегда будет **Unit** по умолчанию.

Аргументы функций (а также и классов) в Kotlin всегда именованные. Это означает, что при передаче значений можно (но всегда необязательно) указывать идентификаторы аргументов:

```
fun f(x: Int, y: Int): Int { return x + y }
fun main() { print(f(y = 4, x = 3)) } → 7
```

При таком способе передачи данных аргументы можно располагать в любом порядке, что очень удобно при большом количестве данных.

Аргументы функций (и классов) всегда можно сделать «по умолчанию» , то-есть задать их значения в списке аргументов:

```
fun f(x: Int, y: Int = 4): Int { return x + y }
fun main() {
    println(f(3)) → 7
```

```
    print(f(3, 7)) → 10
}
```

Если надо, значение по умолчанию всегда можно изменить.

## Переменные

Покажем на примере разные способы объявления и инициализации переменных:

```
fun main() {
    val x: Int = 1; val y = 2; val z: Int; z = 3
    println("x = $x, y = $y, z = $z") → x = 1, y = 2, z = 3
}
```

Неизменяемые переменные объявляются со словом **val**. На самом деле это даже и не переменные, а константы. В примере переменная **x** объявлена с назначением типа **Int** и одновременно инициализирована значением **1**. Переменная **y** инициализирована без объявления типа (тип выводится из контекста). Переменная **z** сначала объявлена с назначением типа без инициализации. В этом случае **z** не будет иметь никакого значения. Инициировать такую переменную можно в другом месте простым присвоением ей значения (**z = 3**). При выводе в примере также применяется интерполяция, если вместо блока только идентификатор переменной, заключать его в фигурные скобки не обязательно, хотя и допустимо.

Теперь пример с изменяемой (mutable) переменной:

```
fun main() { var x = 5; x += 1; println("x = $x") } → x = 6
```

Изменяемые переменные объявляются со словом **var**, в программе им можно присваивать новые значения много раз, но всегда они должны иметь один и тот же тип. Выражение **x += 1** краткая форма для **x = x + 1** (инкремент).

Переменные, объявленные за пределами функций, доступны в теле функций. Такие переменные можно считать глобальными по отношению к функциям.

```
var x = 0; val PI = 3.14;
fun f() { x += 3 }
fun main() {
    println("x = $x, PI = $PI") → x = 0, PI = 3.14
    f(); print("${f()}, "); println("x = $x") → kotlin.Unit, x = 6
}
```

Как видим, функция **f** не возвращает значения (при выводе получили kotlin.Unit), хотя значение **x** изменилось. Здесь надо было бы применить **return**.

Только по традиции идентификаторы переменных записывают с маленькой буквы, хотя в Kotlin это требование не обязательное.

## Классы и экземпляры

Для объявления класса применяется ключевое слово **class** и название класса с большой буквы. Класс может иметь аргументы (часто их называют атрибутами):

```
class Rectangle(var h: Double, var l: Double) {
    var p = (h + l) * 2
}
fun main() {
    val r = Rectangle(5.0, 2.0)
    println("Периметр равен ${r.p}") → Периметр равен 14.0
}
```

Класс **Rectangle** (прямоугольник) имеет два аргумента: высоту **h** и длину **l**, оба имеют тип **Double** (числа с плавающей запятой). В классе объявлена переменная **p** (периметр прямоугольника), тип её не указан и значит будет выведен как **Double**. Такие переменные называют переменными экземпляра, часто также полями или свойствами. Переменная **r** в теле функции **main** называется экземпляром класса (или объектом класса). Для получения экземпляра достаточно классу передать значения его аргументов. Экземпляры создаются так называемым конструктором класса, в данном случае он вызывается неявно. Позже мы рассмотрим конструкторы подробно. Переменная экземпляра **p** доступна из внешней области и для получения её значения применяется точечная нотация: идентификатор экземпляра класса, точка и идентификатор переменной (**r.p**).

Классы являются основой объектно-ориентированного программирования, тема эта очень обширная и мы впоследствии займёмся ею капитально.

## Форматированный вывод

Форматированный вывод в Kotlin имеет большие возможности. Например он позволяет использовать так называемые образцы. Посмотрим на примере:

```
fun main() {
        var a = 1
        val s1 = "a стала $a"
        a = 2
        val s2 = "${s1.replace("стала", "была")}, а теперь стала $a"
                println(s2) → а была 1, а теперь стала 2
}
```

Здесь **s1** и **s2** оразцы. В образце **s1** применена интерполяция для **a**, а в образце **s2** использован образец **s1**, в котором слово «стала» заменено на «была» с помощью стандартной функции **replace**. Для вывода на экран достаточно образец передать функции **println**.

Метод **also** (также, к тому же) позволяет выполнить вывод одновременно с инициализацией. Применяется такой синтаксис:

```
fun f(s: String) {
        val x = "x: $s".also(::println) → x: Привет!
        print(x) → x: Привет!
}
fun main() { f("Привет!") }
```

При этом значение инициализируемой переменной (**x**) равно тому, что выводится (образец). Методу **also** можно передавать и другие операторы, а также и свои собственные функции:

```
fun f(s: Int) {
        fun g(t: Int) = println(t * t)
        val x = (s+1) .also(::g) → 36
        print(x) → 6
}
fun main() { f(5) }
```

Значит, функции **g** передан тот результат, которым инициирована переменная **x**. Позже мы ещё встретимся с применением этой техники.

## Ветвления

В Kotlin используется обычный оператор **if – else**:

```
fun f(a: Int, b: Int): Int { if (a > b) {return a} else {return b} }
fun main() { println("максимум из 7 и 12 равен ${f(7, 12)}") } →
                                        максимум из 7 и 12 равен 12
```

Для возвращения результата первым и вторым блоком здесь всегда требуется оператор **return**. Конструкция **if – else** в Kotlin является выражением и значит для функции можно применить вариант со знаком равенства. Тогда получим:

```
fun f(a: Int, b: Int): Int = if (a > b) a else b
fun main() { println("максимум из 7 и 12 равен ${f(7, 12)}") }
```

Значит в этом варианте не требуются блоки в обоих ветвях **if – else**. Отметим ещё, что поскольку **if** это выражение, то обе ветви должны возвращать результат одного и того же типа.

Иногда ветвь **else** может отсутствовать, например:

```
fun f(a: Int): Unit { if (a > 6) { println("a>6") } }
fun main() {
    f(9) → a>6
    f(2) → ничего не выводится
}
```

## Циклы

Kotlin имеет обычный цикл for:

```
fun main() {
    val m: List<String> = listOf("мы ", "не ", "рабы")
    for (x in m) { print(x) } → мы не рабы
}
```

В цикле использован список **m** (тип **List<String>**). Для создания списка использована стандартная функция **listOf**. Как обычно, тип можно было не указывать, он был бы выведен компилятором. Цикл **for** использует встроенный оператор **in** последовательно присваивая переменной **x** значения всех элементов списка и выводит эти значения на печать.

Для перебора элементов списка можно также использовать индекс:

```
fun main() {
    val m = listOf("мы ", "не ", "рабы")
    for (i in m.indices) { println("элемент с индексом $i: ${m[i]}") }
} → элемент с индексом 0: мы
```

*элемент с индексом 1: не*
*элемент с индексом 2: рабы*

Обращаю внимание на то, что при интерполяции идентификатор списка с индексом надо заключать в фигурные скобки: **${m[i]}**.
Стандартный метод **indices** создаёт ранг, представляющий индексы.
**println(m.indices)** → *0..2*
Ранг это разновидность списка с целочисленными элементами. Синтаксически ранг записывается как **n1..n2** и включает все целые числа из диапазона от **n1** до **n2** включая крайние значения. В примере также использован оператор **in**.

Можно использовать циклы **while**:

```
fun main() {
        val m = listOf("лебедь ", "рак ", "щука")
        var i = 0
        while (i < m.size) { print(m[i]); i++ } → лебедь рак щука
}
```

Стандартный метод **size** определяет размер списка (количество элементов). Оператор **i++** это инкремент (тоже, что и **i = i + 1**). Оператор **while** повторяет свой блок до тех пор, пока условное выражение **i < m.size** даёт **true**.

## Оператор when - else

Кроме оператора **if – else** есть ещё оператор **when - else** у которого в первой ветви может быть целый список альтернативных условий. Посмотрим на примере:

```
fun f(x: Any): String =
        when (x) {
        1 -> "Один"
        "Привет" -> "Приветствие"
        is Long -> "Длинное целое"
        !is String -> "Не строка"
        else -> "Что-то другое"
}
fun main() {
        println(f(1)); println(f("Привет"));println(f(1000L))
        println(f(2)); println(f("нечто"))
} →
```

*Один*
*Приветствие*
*Длинное целое*
*Не строка*
*Что-то другое*

Здесь аргументу функции **f** задан тип **Any** (всякий, любой). При таком неопределённом типе переменная **x** может принимать любое значение, с любым конкретным типом. Результату функции **f** задан тип **String**. Оператор **when** анализирует приданную ему переменную (**x**) с помощью списка условий заданных с помощью стрелки (->). Если значение **x** совпадает с одним из тех, что заданы слева от стрелки, возвращается значение указанное справа и остальные условия не анализируются. Если ни одно условие из списка не выполняется, возвращается значение из ветви **else**. Поскольку **x** имеет тип **Any**, то слева от стрелки могут быть значения разных типов, но справа должны быть только значения типа **String**.

Условие **is Long** проверяет, имеет ли **x** тип **Long** (длинное целое число), а **!is String** даёт **true** когда **x** имеет тип **не String**. Здесь используется встроенный оператор **is**. Восклицательный знак меняет условие **is** на противоположное. Длинные целые числа записываются с буквой **L** на конце.

## Примеры с рангами

Оператор **in** позволяет проверять принадлежность переменной списку или рангу, например:

```
fun f(x: Int) {
    if (x in 2..5) { println("да x в ранге")}
    else { println("нет")}
}
fun main() {
    f(3) → да x в ранге
    f(9) → нет
}
```

Оператор **in** также может применяться с восклицательным знаком впереди (**!in**) и тогда это будет противоположное условие «не входит» или «не принадлежит».

Можно создавать ранги с заданным шагом вместо единицы:

```
fun main() {
        for (x in 1..10 step 2) { print(x) } → 13579
        println()
        for (x in 9 downTo 0 step 3) { print(x)} → 9630
}
```

Шаг указывается после ранга с оператором **step**. Если точки заменить оператором **downTo**, то можно создавать ранги с убывающими членами.

### Некоторые операции со списками

В следующем примере показано применение нескольких стандартных методов работы со списками:

```
fun main() {
        val m = listOf("кура", "собака", "кошка", "овца")
        m.filter { it.startsWith("к") }
          .sortedBy { it }
          .map { it.uppercase() }
          .forEach { print("$it ") } → КОШКА КУРА
}
```

Метод **filter** позволяет отобрать элементы из списка по признаку, который задаётся в блоке. В данном случае в блоке использован метод **startsWith**, который даёт **true**, если слово начинается с заданной буквы. Переменная **it** встроенная, метод **filter** в цикле передаёт ей значения элементов списка. Метод **filter** создаёт новый список из отобранных элементов, и этому списку можно было бы дать новое имя. В Kotlin имеется синтаксический сахар, позволяющий работать с новым списком без использования имени. Для этого надо вызвать следующий метод (у нас **sortedBy**) без указания имени списка. Метод **sortedBy** сортирует список. Дальше аналогично применяем итератор **map**, который выполняет над элементами операцию, заданную в блоке. В нашем случае это метод **uppercase**, переводящий все буквы в слове в верхний регистр. Передаваемые таким способом методы называют лямбда-функциями (иногда их называют замыканиями closure). Итератору **forEach** тоже передана лямбда-функция **print** для вывода списка на терминал. Итератор **map** создаёт новый список, а **forEach** нет. В остальном эти итераторы похожи друг на друга.

Для случаев, когда используется встроенная переменная **it**, есть синтаксический сахар. Покажу на таком примере:

```
import kotlin.math.*
fun main() {
        val m = listOf(0.5, 1.2, 2.7)
        val r = m.map { sin(it) }
        println(r) → [0.4794255386, 0.93203908596, 0.42737988023]
}
```

Как обычно, здесь итератору **map** передаётся лямбда-функция с переменной **it**. Иначе этот код можно написать в таком виде:

```
import kotlin.math.*
fun main() {
        val m = listOf(0.5, 1.2, 2.7)
        val r = m.map (::sin)
        println(r) → [0.4794255386, 0.93203908596, 0.42737988023]
}
```

Если лямбда-функция представлена одним выражением, как в нашем примере, то вместо неё можно прямо передать это выражение с оператором :: впереди. В документации этот оператор называется *method reference* (что-то вроде ссылочного метода) . Теперь переменная **it** передаётся выражению неявно. Обращаю внимание на то, что здесь должны быть круглые скобки, так как это не лямбда-выражение. В таком стиле можно использовать и функции собственного приготовления:

```
fun main() {
        fun f(x: Double): Double { return x + 3.0 }
        val m = listOf(0.5, 1.2, 2.7)
        val r = m.map (::f)
        println(r) → [3.5, 4.2, 5.7]
}
```

Впоследствии мы рассмотрим работу со списками и другими коллекциями более детально.

## Тип Nullable

Иногда в программе возникает ситуация, при которой объявленная переменная не получает никакого значения. Попытка использования такой переменной приведёт к ошибке компиляции, например:

**val x: Int**
**print(x)** → *error: variable 'x' must be initialized* (получим при компиляции)

Для того, чтобы избежать подобных ситуаций вводится так называемый тип nullable (способный принимать значение **null**). В языке Swift такой тип называется optional (необязательный, в смысле не обязательно должен иметь значение). Любой тип можно превратить в nullable, для этого достаточно к названию типа добавить вопросительный знак. Любая переменная, объявленная таким образом, может получать специальное значение **null**, которое дальше можно использовать. При этом такие переменные могут использоваться и обычным образом:

```
fun main() {
        var x: Int?
        x = null; println(x) → null
        x = 77; print(x) → 77
}
```

Посмотрим теперь, как это можно использовать:

```
fun f(a: String, b: String) {
        val x: Int? = a.toIntOrNull(); val y: Int? = b.toIntOrNull()
        if (x != null && y != null) { println(x * y) }
        else { println("'$a' или '$b' не число") }
}
fun main() {
        f("6", "7") → 42
        f("a", "7") → 'a' или '7' не число
        f("a", "b") → 'a' или 'b' не число
}
```

Функция **f** принимает два целых числа, представленных в строковом выражении. Обычно для перевода строки в целое число применяется стандартный метод **toInt**. Для получения типа nullable (**Int?**) имеется другой метод **toIntOrNull**. Если строка представляет число, например**"6"**, этот метод вернёт это целое число, а если строка не число, например**"apple"**, то получим значение **null**. Появляется возможность контроля над входными данными, чтобы избежать непредвиденной ошибки и аварийного останова программы.

Ещё один характерный пример:

```
fun f(x: Any): Int? {
```

```
    if (x is String) { return x.length }
    return null
}
fun main() {
    fun g(s: Any) {
        println("размер '$s': ${f(s) ?: "s не строка"} ")
    }
    g("У лукоморья дуб зелёный") →размер 'У лукоморья дуб зелёный': 23
    g(1000) → размер '1000': s не строка
}
```

У функции **f** аргумент **x** имеет тип **Any**. Такие переменные могут принимать значения любого типа. Если функции **f** передать строку, то она вернёт длину строки (число символов в строке), полученную с помощью стандартного метода **length**. Если же аргумент будет какого-то другого типа (не String), то **f** вернёт значение **null**. Функция **g** просто вызывает функцию **f**, передавая ей свой аргумент, а при выводе на терминал проверяет, не имеет ли результат значение **null** . Если это так, то выводится сообщение, что функция получила значение другого типа (не String). Здесь использована краткая форма условного оператора **if – else** (синтаксический сахар). Эта форма имеет вид: **expression1 ?: expression2**. Если **expression1** имеет значение не **null**, то условное выражение возвращает это значение, в противном случае возвращается значение **expression2**. При этом в примере использована интерполяция результата. Покажу использование оператора **?:** на ещё одном, более простом примере:

```
fun g(x: String?): String? {
    val y = x ?: "x = null"
    return y
}
fun main() {
    println(g("Привет")) → Привет
    print(g(null)) → x = null
}
```

## Выходы (returns) и прыжки (jumps)

Для выхода из цикла по условию имеются обычные операторы **break** (для прерывания цикла) и **continue** (для возврата на начало цикла). Кроме того, в Kotlin любое выражение может иметь метку, представляющую собой идентификатор со знаком **@** на конце (**name@**). Для ссылки на метку надо использовать этот же идентификатор с тем же знаком **@** вначале (**@name**). С помощью метки можно, например, сделать переход по условию в любую точку цикла:

```
fun main() {
        метка@ for (i in 1..5) {
                for (j in 1..5) {
                        if ((i * j) % 3 == 0) continue@метка
                        print("$i, $j | ")
                }
        }
}
```

Результат:

*1, 1 | 1, 2 | 2, 1 | 2, 2 | 4, 1 | 4, 2 | 5, 1 | 5, 2 |*

Точно также можно применять и оператор **break**. Ели в примере заменить **continue@метка** на **break@метка**, получим:

1, 1 | 1, 2 |

Оператор **return** обеспечивает выход из вложенной функции во внешнюю функцию:

```
fun f() {
        val m = listOf(1, 2, 3, 4, 5)
        m.forEach { if (it == 3) return; print("$it, ") } → 1, 2,
        print("Эта точка недостижима") → Ничего не выводится
}
fun main() { f() }
```

Здесь выход выполнен из лямбда-функции (из closure) во внешнюю функцию **f**, а цикл итератора **forEach** на этом заканчивается. Заканчивается и работа самой функции **f** и код, расположенный после точки выхода, никогда не будет выполнен. С применением метки выход можно выполнить в любую точку:

```
fun f() {
        val m = listOf(1, 2, 3, 4, 5)
        m.forEach метка@{ if (it == 3) return@метка; print("$it, ") }
                                                        →1, 2, 4, 5,
```

```
    print("Эта точка достижима") → Эта точка достижима
}
fun main() { f() }
```

В этом варианте выполняется переход из *closure* в итератор **forEach** и цикл продолжается. Без перехода по метке это же можно сделать так:

```
fun f() {
    val m = listOf(1, 2, 3, 4, 5)
    m.forEach(fun(t: Int) { if (t == 3) return; print("$t, ")}) →
                                                    1, 2, 4, 5,
    print("Оператор выполняется") → Оператор выполняется
}
fun main() { f() }
```

Здесь в блоке итератора **forEach** использована безымянная функция и оператор **return** выполняет выход из этой функции, при этом цикл итератора продолжается.

Ещё один пример:

```
fun f() {
    val m = listOf(1, 2, 3, 4, 5)
    run метка@ { m.forEach { if (it == 3) return@метка; print("$it, ") }} → 1, 2,
    print(" Выполняется") → Выполняется
}
fun main() { f() }
```

Здесь использована встроенная функция **run**, которая просто выполняет приданный ей блок. В данном случае переход по метке прерывает цикл итератора. Можно применять ещё и неявные метки:

```
fun f() {
    val m = listOf(1, 2, 3, 4, 5)
    m.forEach { if (it == 3) return@forEach; print("$it, ") } →
                                                    1, 2, 4, 5,
    print(" С неявной меткой") → С неявной меткой
}
fun main() { f() }
```

Здесь неявная метка обеспечивает выход на указанный итератор и цикл продолжается. Можно придумать ещё и другие комбинации, но пожалуй достаточно и этих.

На этом закончим ознакомительную часть руководства и перейдём к более систематическому изучению языка Kotlin.

# 2.Основные (стандартные) типы

В Kotlin всё является объектом и стандартные типы можно применять, как обычные классы.

### Числа

Для целых чисел имеется четыре типа с разными максимальными значениями:
**Byte** (8 бит) — 127
**Short** (16 бит) — 32767
**Int** (32 бит) - 2,147,483,647
**Long** (64 бит) - 9,223,372,036,854,775,807

При инициализации переменной без указания типа числом не более 2,147,483,647 будет выведен автоматически тип **Int**, а при больших числах получим тип **Long**. Если требуется небольшому числу задать тип **Long**, надо объявить тип явно, или воспользоваться суффиксом **L**:
**val x: Long = 25**
**val x = 25L** (не допускается буква l)

Оба этих объявления равнозначны.

Числа с плавающей точкой (или по старой традиции с плавающей запятой) представлены двумя типами:
**Float** (32 бит) — 6-7 значащих цифр
**Double** (64 бит) — 15-18 значащих цифр

При инициализации переменной числом с плавающей точкой всегда будет выведен тип **Double**. Если нужен тип **Float** надо указать тип явно или применить суффикс **f** ( допустимо **F**):
**val x: Float = 0.0123e5** (допустима научная нотация)
**val x = 12.34f**

Нет автоматической конверсии целых чисел в числа с плавающей точкой:
**val x: Double = 123** - здесь будет ошибка, надо 123.0

Двоичные числа записываются с двумя знаками **0b** впереди:

**val x = 0b110100**
шестнадцатеричные — с **0x**:
**val x = 0xFC50** (цифра **0**, а не буква **o**)

При записи длинных чисел можно применять знак подчёркивания для лучшей читабельности:
**val x = 12_251_003_105**

Для конверсии чисел из одного типа в другой имеются следующие стандартные методы: **toByte()**: Byte, **toShort()**: Short, **toInt()**: Int, **toLong()**: Long, **toFloat()**: Float, **toDouble()**: Double, **toChar()**: Char. Метод **toChar()** конвертирует число в знак:
**print(74.toChar())** → *J*

Во многих случаях конверсия выполняется неявно без применения вышеназванных методов:
**val x = 3.0 / 2** → *1.5*
Здесь целое число **2** автоматически приводится к типу Double.

Типы беззнаковых целых чисел обозначаются с добавлением буквы U: **UByte**, **Ushort**, **Uint**, **ULOng**. Беззнаковые числа поддерживают большинство операций для знаковых типов. Можно использовать суффикс **u**:
**val x = 42u** - получим тип UInt

## Арифметические операции

Арифметические операторы в Kotlin имеют обычный вид: (+), (-), (*), (/), (%). Оператор **%** возвращает остаток от деления:
**5 % 2** → *1*
Этот оператор применим и для чисел с плавающей точкой:
**5.3 % 2.0** → *1.3*

Операции с целыми числами всегда возвращают целое число:
**9 / 4** → *2*
Если в арифметическом выражении есть хотя бы одно число с плавающей точкой, то и результат будет число с плавающей точкой:
**(8 + 3.0) / 3** → *3.867*

Операторы для поразрядных (побитовых) операций:
**shl()** – сдвиг влево с учётом знака, **shr()** – сдвиг вправо с учётом знака, **ushr()** - беззнаковый сдвиг вправо, **and()** - побайтовое «и», **or()**

- побайтовое «или», **xor()** - побайтовое «исключающее или», **inv()** - побайтовая инверсия.

Операторы проверки на равенство: **a == b** и **a != b**

Операторы сравнения: **a < b, a > b, a <= b, a >= b**

Для проверки принадлежности числа заданному рангу применяются операторы **in** (принадлежит) и **!in** (не принадлежит): **x in a..b, x !in a..b**.

Все математические функции расположены в модуле **math**, который надо предварительно импортировать:

**import kotlin.math.***

Звёздочка означает, что импортируются все функции. Можно импортировать какую-нибудь одну:

**import kotlin.math.cos**

## Тип Boolean

Тип **Boolean** представлен двумя значениями: **true** и **false**. **Boolean** имеет **nullable** эквивалент **Boolean?**, который может принимать значение **null**. Встроенные операции с типом Boolean включают:

**||** – дизъюнкция (логическое **OR**)

**&&** – конъюнкция (логическое **AND**)

**!** - отрицание (логическое **NOT**)

Операции || и **&&** являются ленивыми (lazy). Это значит, что если в выражении **x || y** *x = true*, а в выражении **x && y** *x = false*, то второй операнд y не вычисляется.

```
fun main() {
        val x: Boolean = true
        val y: Boolean = false
        val z: Boolean? = null
        println(x || y) → true
        println(x && y) → false
        println(!x) →false
        print(z) → null
}
```

В Kotlin применяются обычные операции сравнения: (**>, >=, <, <=**) и проверки на равенство: (==) - равенство по значению, (===) - равенство по ссылке (даёт **true** для одного и того же экземпляра) и операция, противоположная (==): (!=) - не равно по значению.

## Символы (знаки, буквы) (characters)

Символы представлены типом **Char** со значениями в виде знаков в одиночных кавычках: **'a', 'b', 'c'**. Несколько специальных символов (так называемые управляющие последовательности) записываются с обратным слешем впереди: **\t, \b, \n, \r, \', \", \\, \$**. Если символ представляет цифру, то её можно конвертировать в число с помощью стандартного метода **digitToInt**:

```
fun main() {
        val x: Char = 'a'
        val y = '7'
        println(x) → a
        println(2 * y.digitToInt()) →14
}
```

## Строки (String)

Тип **String** представляет текст в двойных кавычках (одинарные кавычки здесь недопустимы). Строки почти то же самое, что списки: отдельные символы можно извлекать по индексу и обрабатывать строки можно в цикле **for**:

```
fun main() {
        val x = "Лариса"; println(x[2]) → p
        val y = "Москва"; for (i in y) { print(i + "_") } → М_о_с_к_в_а
}
```

Применение разных методов трансформации строк всегда создаёт новую строку, а исходная строка не изменяется:

```
fun main() {
        var s = "abcd"
        println(s.uppercase()) →ABCD
        println(s) →abcd
}
```

Для конкатенации строк применяется знак суммирования (+), добавлять к строке можно и значения других типов, которые автоматически конвертируются в тип String:

```
fun main() { val s = "abc" + 1; println(s + "def") } → abc1def
```

Можно использовать текст, состоящий из нескольких строчек, которые надо поместить между тройками из двойных кавычек:

```
val s = """ for (c in "foo")
print(c) """
fun main() { for (i in s) { print(i + " ") }} →
    for (c in "foo")
    print(c)
```

Здесь переменная **s** тоже имеет тип **String** и может использоваться, как обычно. Внутри такого текста могут быть кавычки, управляющие и другие специальные символы, которые не обрабатываются и выводятся на терминал «как есть». Не требуется, чтобы тройные кавычки располагались на отдельных строчках, как в некоторых других языках.

## Списки и массивы

Списки и массивы в Kotlin похожи, но между ними есть разница, иногда существенная. Ранее уже упоминалось, что для создания списка применяется метод **listOf()**. Соответственно, для массива есть метод **arrayOf()**:

```
var s = listOf(1,2,3,4,5)
var m = arrayOf(1,2,3,4,5)
fun main() {
    println(s); → [1, 2, 3, 4, 5]
    println(m.contentToString()) → [1, 2, 3, 4, 5]
    println(s[2]) → 3
    println(m[3]) → 4
    m[4] = 88
    println(m.contentToString()) → [1, 2, 3, 4, 88]
}
```

Список можно передать оператору **println** непосредственно, он автоматически приводится к типу **Strin**g. Для массива конвертацию типа надо делать принудительно с помощью метода **contentToString()**. Для извлечения элементов по индексу в обоих случаях применяется одинаковый синтаксис. Элементы массива можно изменять по индексу, для списка это невозможно (существует разновидность изменяемых списков, рассмотрим позже). Списки и массивы одинаково можно использовать в цикле **for** и в итераторах.

Если список или массив объявить со словом **var**, то ему можно будет присваивать другие значения всё с теми же методами **listOf** или **arrayOf**.

Элементы списков и массивов могут быть любых типов, общее обозначение типа для них будет **Any**:

```
var s: List<Any> = listOf(1,3.05,"abc",true,5)
var m: Array<Any> = arrayOf(1,3.05,"abc",true,5)
```

Пустой список или массив создаётся так:

```
 s: List<String> = listOf(); println(s); → []
var m: Array<Any> = arrayOf(); println(m.contentToString()) → []
```

Тип в данном случае надо указывать обязательно.

Массивы и списки имеют так называемые конструкторы **Array** и **List** соответственно. Эти конструкторы принимают размер массива или списка и им придаётся функция, вычисляющая элементы:

```
fun main() {
        val m = Array(5) { i -> (i * i).toString() }
        m.forEach { print(it + " ") } → 0 1 4 9 16
        val s = List(5) { i -> (i * i).toString() }
        println(); println(s) → [0, 1, 4, 9, 16]
}
```

Можно, конечно, и так:

```
fun main() {
        fun f(i: Int): String { return (i * i).toString() }
        val m = Array(5) { i -> f(i) }
        m.forEach { print(it + " ") } → 0 1 4 9 16
        val s = List(5) { x -> f(x) }
        println(); println(s)→ [0, 1, 4, 9, 16]
}
```

А так можно создавать списки и массивы с повторяющимися элементами:

```
val s = List(5) { 9 }; println(s) → [9, 9, 9, 9, 9]
```

Имеются кроме того так называемые примитивные типы массивов: **ByteArray, ShortArray, IntArray, DoubleArray**, и так далее, и соответствующие функции вроде **intArrayOf()**:

```
fun main() {
        val m: DoubleArray = doubleArrayOf(3.0, 2.0, 7.0)
        m[0] = m[1] + m[2]
        print(m.contentToString()) →[9.0, 2.0, 7.0]
```

**}**

Массив **m** был объявлен со словом **val**, но, как видим, изменения его элементов допустимы и в этом случае. Для списков подобных примитивных типов нет. Кроме списков и массивов в Kotlin существуют и другие коллекции, мы их потом рассмотрим отдельно.

# 3.Классы

## Конструкторы

В своей основе класс в Kotlin подобен классам в любом современном языке программирования. Но есть и существенные особенности; мы будем отмечать их по ходу дела. Начнём с простейшего примера:

```
class Demo(x: Double, y: Double) {
        val z = (x + y) * 3.2
        val t = x; val s = y
        fun f(v: Double): Double { return (t + s) / (v - 2.0) }
}
fun main() {
        val p = Demo(2.1, 3.75)
        println(p.z)  →18.72
        println(p.f(5.0))  →1.95
}
```

Строка **class Demo(x: Double, y: Double)** называется заголовком класса, а в фигурных скобках — тело класса. Слово **class** в заголовке служебное (ключевое), **Demo** — название (идентификатор) класса, всегда с заглавной буквы, так как это новый тип данных. За названием следует список аргументов класса. Если таковых нет, то пустые скобки ставить не обязательно, хотя и допустимо. При создании экземпляра класса (или объекта) аргументы передаются некоторой функции, которая называется конструктором. Обычно конструктор вызывается неявно, но в Kotlin есть возможность указать его явно. Для этого надо в заголовке вставить слово **constructor** после названия класса:

```
class Demo constructor(x: Double, y: Double) {..}
```

Если конструктору придаётся дополнительная информация, то слово **constructor** в заголовке становится обязательным Например, конструктор может иметь модификатор доступа **private**:
**class Demo private constructor(x: Double, y: Double) {..}**
По умолчанию конструктор всегда **publick**.

Объявленная в теле класса переменная **z** называется переменной экземпляра класса (полем или свойством класса). Как видим, при вычислении свойства **z** доступны аргументы класса (в нашем случае это **x** и **y**). Название *переменная экземпляра* хорошо подходит потому, что эта переменная доступна по чтению из экземпляра класса. А если её объявить со словом **var**, то она будет доступна и по записи.

Функция **f** соответственно называется методом (или функцией) экземпляра класса. (В документации Kotlin функции, объявленные внутри класса или объекта, называются member function. Для краткости чаще будем использовать термин метод класса.) В теле этой функции аргументы класса не доступны, то-есть, мы не можем написать:
**fun f(v: Double): Double { return (x + y) / (v - 2.0) }**
Приходится вводить локальные переменные **t** и **s**, вычисляемые через аргументы класса. В принципе здесь можно не вводить новых идентификаторов, а использовать те же **x** и **y**, то-есть, написать:
**val x = x; val y = y**
**fun f(v: Double): Double { return (x + y) / (v - 2.0) }**
Конфликта имён не происходит, так как это разные переменные. Обычно так всегда и поступают. Можно также применить служебное слово **this** в таком виде:
**val t = x; val s = y**
**fun f(v: Double): Double { return (this.t + this.s) / (v - 2.0) }**

Здесь **this** ссылается на экземпляр (объект) класса.

Строка **val p = Demo(2.1, 3.75)** создаёт экземпляр класса (или объект) с идентификатором **p**, передавая ему аргументы. На самом деле по имени класса вызывается функция-конструктор, которая в некоторых языках вызывается словом **new** перед названием класса. Создавая класс мы создаём новый тип данных, который называется по имени класса. Переменная **p** имеет тип **Demo**, который можно было бы указать явно:
**val p: Demo = Demo(2.1, 3.75)**

Теперь можно вызывать свойства и методы экземпляра класса **p**, применяя точечную нотацию: **p.z** и **p.f(5.0)**.

Аргументами класса могут быть любые объекты. Поскольку функции в Kotlin тоже объекты, то они также могут быть аргументами класса, например:

```
class A(g: (Int, Int) -> Int) { val f = g }
fun main() {
        val p = A({x, y -> x + y})
        println(p.f(2, 5)) → 7
}
```

Выражение **val f = g** создаёт переменную экземпляра (свойство), а на самом деле это функция. Здесь при передаче значения классу использовано лямбда-выражение (или closure) **{x, y -> x + y}**, которые рассмотрим позже. Инициируя переменную **f** этим лямбда-выражением, получаем функцию (или метод) экземпляра.

Попробуем теперь придумать какую-нибудь хотя и примитивную, но похожую на реальную задачу. Пусть например на заводе работают токари, вытачивающие на старом токарном станке какую-то деталь (теперь это делают автоматы). Пусть их зарплата определяется некоторой платой (назовём её словом тариф) за каждую деталь (такая зарплата называется сдельной). Составим программу по расчёту этой зарплаты. Поскольку Kotlin позволяет использовать кириллицу, напишем всё на русском языке. Хотя многие современные языки имеют такую возможность, на практике всё пишется только на латинице. Позволим себе нарушить эту традицию:

```
class Токарь(имя: String, детали: Array<Int>) {
        val всего = детали.sum()
        val среднее = всего.toDouble()/детали.size
        val имя = имя
        fun производительность() {
                println("$имя производит $среднее деталей за день")
        }
        fun зарплата(тариф: Double) {
                print("Зарплата $имя равна ${тариф * всего} рублей")
        }
}
fun main() {
        val детали = arrayOf(7, 12, 5, 10)
```

```
    val Борисов = Токарь("Виктор Борисов", детали)
    Борисов.производительность()
    Борисов.зарплата(25.7)
}
```

Получим такой результат:

*Виктор Борисов производит 8.5 деталей за день*
*Зарплата Виктор Борисов равна 873.8 рублей*

Итак, мы создали класс **Токарь** с аргументами: **имя** — имя конкретного токаря, и массива **детали**, элементы которого равны количеству фактически выточенных деталей за каждый день. В классе **Токарь** имеется два свойства **всего** и **среднее** и два метода **производительность** и **зарплата**. Для конкретного токаря Борисов создаём объект **Борисов**, передавая ему полное имя токаря**"Виктор Борисов"** и массив **детали** с поденной выработкой (ограничились четырьмя днями). Теперь можно вызывать методы объекта для получения результатов. Для другого токаря надо создавать новый объект.

Нетрудно было бы запрограммировать диалог для ввода исходных данных и создать цикл для перебора всех токарей в цеху. Все объекты можно поместить в массив, например с названием **рабочие**:

```
рабочие: Array<Токарь>
```

Вывод результатов можно было бы оформить в виде таблицы и так далее. Ясно, что в реальности всё было бы несколько сложнее, но, как мне кажется, основная идея понятна: класс это общий образ всех токарей, а объекты (или экземпляры) представляют каждого токаря в отдельности. Можно сказать, что в объекте мы конкретизируем контент класса.

В Kotlin конструктор может иметь так называемый блок инициализации, у него впереди ставится служебное слово **init**. В этом блоке может быть некоторый программный код, который выполняется в момент создания объекта. Посмотрим, как такой блок можно применить в первом примере:

```
class Demo(x: Double, y: Double, v: Double) {
    val z = "Свойство z: ${(x + y) * 3.2}".also(::println)
    init{ println("Результат равен: ${(x + y)/(v - 2.0)}") }
}
fun main() { Demo(2.1, 3.75, 5.0) }
```

Получаем такой результат:

*Свойство z: 18.72*
*Результат равен: 1.95*

Здесь кроме блока инициализации мы ещё применили знакомый нам уже метод **also** для вывода свойства **z**. Выгода тут заключается в том, что нам не нужно теперь в функции **main** вызывать для вывода свойство **z** и метод **f**, вывод результатов выполняется автоматически. Кроме того, в блоке инициализации доступны аргументы класса и отпадает необходимость в локальных переменных. Экземпляр класса здесь мы создаём, но отпала необходимость вводить переменную **p** для использования экземпляра в дальнейшем. Пришлось, правда, переменную **v** передавать через аргументы класса, иногда это может быть невозможным по логике самой задачи. Тогда придётся создавать соответствующее свойство (поле) класса.

В списке аргументов класса можно применять значения по умолчанию точно также, как это делается для функций. Например, заголовок нашего класса мог бы быть таким:

**class Demo(x: Double, y: Double, v: Double = 5.0)**

Тогда при создании объекта не надо было бы передавать значение переменной **v**:

**fun main() { Demo(2.1, 3.75) }**

Но при этом у нас остаётся возможность задать новое значение для **v**, если потребуется, например:

**fun main() { Demo(2.1, 3.75, 9.05) }**

Отметим ещё, что в классе может быть несколько блоков инициализации, все они будут выполнены при создании объекта.

В Kotlin конструктор, указанный (явно или неявно) в заголовке, называется первичный конструктор, потому что кроме него могут быть один или несколько вторичных конструкторов. Вторичный конструктор объявляется со словом **constructor**, после которого располагается список аргументов, а затем блок конструктора в фигурных скобках:

```
class C(x: Int) {
        init { println("Первичный конструктор: $x") }
        val v :Int = 2
        val x: Int = x
        fun f(t: Int) = t * x
        constructor(x: Int, y: Int) : this(x) {
                println("Функция f: ${f(3)}")
```

```
            val z = (x + y) * v
            println("Вторичный конструктор: $z")
        }
}
fun main() {
        C(5, 7) → Первичный конструктор: 5
                  Функция f: 15
                   Вторичный конструктор: 24
        C(5) → Первичный конструктор: 5
}
```

Если во вторичном конструкторе используются аргументы первичного конструктора, то эти аргументы надо вставить в список аргументов вторичного конструктора и связать их с первичным конструктором с помощью метода **this**. Анализируя данный пример мы можем отметить:

- с помощью вторичного конструктора можно добавить в список аргументов класса дополнительные аргументы (мы добавили **y**).
- во вторичном конструкторе доступны свойства и методы первичного конструктора (у нас свойство **v** и метод **f**).
- при создании экземпляра класса блок вторичного конструктора немедленно выполняется.
- если при создании экземпляра задать только аргументы первичного конструктора (без добавленных во вторичном конструкторе), то блок вторичного конструктора не будет выполнен.

Строка **val x: Int = x** потребовалась для метода **f** (не для вторичного конструктора).

В целом, вторичный конструктор похож на блок инициализации. В примере от применения вторичного конструктора мы получили мало пользы, подробнее разбираться со всем этим надо на реальных задачах.

## Наследование (Inheritance)

Все классы в Kotlin имеют общий родительский класс (суперкласс) **Any**. По умолчанию все классы **final**, а это значит, что они не могут иметь дочерние классы. Для того, чтобы класс мог иметь наследников, он должен быть объявлен с модификатором **open**:

```
open class Abc() {...}
```

Среди аргументов дочернего класса должны содержаться все аргументы суперкласса. После списка аргументов через двоеточие надо указать имя суперкласса со своими аргументами:

```
open class A(x: Int, y: Int) {
        val z = 2 * (x + y)
        val x = x; val y = y
        fun f(t: Int): Int = t + x + y
        init { println("Конструктор класса A") }→
                                                Конструктор класса A
}
class B(x: Int, y: Int, q: Int) : A(x, y) {
        val v: String = "class B: ${x + y}".also(::println) → class B: 5
        val g = 7 * q
}
fun main() {
        val p = B(2, 3, 4)
        println(p.z) → 10
        println(p.f(7)) → 12
        println(p.g) → 28
}
```

Итак, можем видеть, что в дочернем классе **B** доступны все свойства и методы суперкласса **A** (у нас свойство **z** и метод **f**). В дочернем классе к аргументам суперкласса можно добавлять свои аргументы (аргумент **q**). При создании экземпляра дочернего класса выполняются блоки инициализации родительского класса.

Если в родительском классе есть вторичный конструктор, то в дочернем классе надо также создать вторичный конструктор. После слова **constructor** указать аргументы вторичного конструктора родительского класса и после двоеточия вставить слово **super** с аргументами в скобках, например:

```
constructor(x: Int, y: Int) : super(x, y)
```

При создании объекта дочернего класса вторичный конструктор суперкласса будет выполняться также, как если бы он был в дочернем классе.

## Подмена (overriding) методов и свойств

Часто эту операцию называют ещё перегрузкой и заключается она в замене методов и свойств суперкласса на другие значения этих методов и свойств в дочернем классе. Kotlin требует явно указывать, какие методы и свойства могут перегружаться. Для этого применяется тот же модификатор **open**. Пример перегрузки метода:

```
 open class A() {
        open fun f(x: Int, y: Int) = x + y
}
class B() : A() {
        override fun f(x: Int, y: Int) = x * y
}
fun main() {
        val p = B(); print(p.f(2, 3)) → 6
        val r = A(); print(r.f(2, 3)) → 5
}
```

Значит, достаточно в дочернем классе объявить перегружаемый метод заново после служебного слова **override**. А теперь пример перегрузки свойства:

```
open class A() { open val x: Int = 0 }
class B() : A() { override val x = 5 }
fun main() {
        val p = B(); println(p.x) → 5
}
```

В общем-то всё также, как и для метода. Теперь более сложный пример:

```
open class A(val x: String) {
        init { println("Это класс A") }
        open val s: Int = x.length.also { println("s в классе A: $it") }
}
class B( x: String, val y: String) : A(x.replaceFirstChar { it.uppercase() }) {
        init { println("Это класс B") }
        override val s: Int = (super.s + y.length).also { println("s в классе B: $it") }
        }
fun main() {
        B("hello", "world")
```

```
}
```

Получим такой результат:

*Это класс A*
*s в классе A: 5*
*Это класс B*
*s в классе B: 10*

Здесь при связывании аргумента **x** дочернего класса **B** с аргументом **x** родительского класса **A** одновременно выполняются некоторые действия над этим аргументом:

```
A(x.replaceFirstChar { it.uppercase() })
```

С помощью функции **replaceFirstChar** первая буква заменяется на то, что вычисляется в придаваемом блоке, там эта буква переводится в верхний регистр. Функция **replaceFirstChar** передаёт свой аргумент в блок с идентификатором **it**.

При вычислении свойства **s** в классе **A** одновременно выполнен вывод на терминал с использованием метода **also**. В данном случае методу передан блок (двойное двоеточие не требуется). Метод **also** также передаёт аргумент в блок с идентификатором **it**. В классе **B** свойство **s** перегружается тоже с одновременным выводом результата на терминал.

В обоих классах **A** и **B** имеются блоки инициализации, которые выполняются при создании экземпляра класса **B**. При этом блок инициализации родительского класса выполняется первым.

### Вызов методов и свойств суперкласса

В дочернем классе можно вызвать любой метод или свойство суперкласса с помощью ключевого слова super:

```
open class A {
        open fun f() { println("в классе A") }
        val x: String = "Привет!"
}
class B : A() {
        override fun f() { super.f(); println("в классе B") }
        val y: String = super.x
}
fun main() { val p = B(); p.f(); println(p.y) }
```

Результат:

*в классе A*
*в классе B*
*Привет!*

Здесь перегруженная функция **f** в классе **B** вызывает саму перегружаемую функцию из класса **A**.

Kotlin позволяет создавать внутренние классы (а ещё и вложенные, позже мы их рассмотрим). Для этого применяется модификатор **inner**. Во внешнем классе можно создавать экземпляры (объекты) внутреннего класса. Рассмотрим пример, когда внутренний класс создаётся в дочернем классе:

```
open class A { fun f(x: Int) = x * x }
class B: A() {
        inner class C { fun g(t: Int) = super@B.f(t) }
        val r = C()
}
fun main() {
        val p = B()
        print(p.r.g(3)) → 9
}
```

В примере показано, как из внутреннего класса **C** можно вызывать метод суперкласса **A**. Для этого используется слово **super** и специальный синтаксис со знаком **@**:

```
fun g(t: Int) = super@B.f(t)
```

Метод **g** класса **C** вызывает метод **f** суперкласса **A**. Обратите внимание на то, что после знака **@** стоит имя дочернего класса **B**, а не родительского **A**. В классе **B** создан объект внутреннего класса **C** (**val r = C()**). При вызове метода **g** из функции **main** указано **p.r.g(3)**, где **p** – объект класса **B**. Это надо понимать так, что вызывается метод **g**, принадлежащий объекту **r**, который в свою очередь принадлежит объекту **p**. Аналогичным образом из внутреннего класса вызывается свойство суперкласса:

```
open class A { val x: String = "Привет!" }
class B: A() {
        inner class C { val s = super@B.x }
        val r = C()
}
fun main() {
        val p = B()
```

```
        println(p.r.s) → Привет!
}
```

Классы в Kotlin могут использовать так называемые интерфейсы, о которых будем говорить позже. А пока рассмотрим случай, когда дочерний класс подключает ещё и интерфейс, содержащий метод с тем же именем, что и метод, принадлежащий суперклассу:

```
open class A { open fun f() { println("класс A") } }
interface C { fun f() { println("интерфейс C") }}
class B() : A(), C {
        override fun f() {
                super<A>.f()
                super<C>.f()
        }
}
fun main() { val p = B(); p.f() }
```

Результат:

*класс A*
*интерфейс C*

Интерфейс C (они объявляются аналогично классам) содержит метод **f**, и суперкласс **A** также содержит метод **f**. Класс **B** наследует классу **A** и подключает интерфейс **C** (для этого надо указать имя интерфейса после имени суперкласса через запятую). Для вызова методов с одинаковыми именами надо функцию объявить с модификатором **override** и вызывать со словом **super**, указав имя класса или интерфейса (тип) в угловых скобках.

## Вложенные (nested) и внутренние (inner) классы

Классы могут быть вложенными в другие классы:

```
class A {
        val x: Int = 5
        class B {
                fun f(y: Int) = 2 * y
        }
}
fun main() {
        val r = A()
        println(r.x) → 5
```

```
    val p = A.B()
    println(p.f(3)) → 6
}
```

Нельзя создать экземпляр вложенного класса: **val v = B()**. Из вложенного класса нет доступа к свойствам и методам наружного класса. В нашем примере нельзя написать:

```
fun f(y: Int) = y * x
```

Переменная **x** здесь недоступна. Доступ к свойствам и методам внешнего класса возможен из вложенного класса с модификатором **inner**:

```
class A {
    val x: Int = 5
    inner class B {
        fun f(y: Int) = x * y
    }
}
fun main() {
    val p = A().B()
    println(p.f(3)) → 15
}
```

Надо иметь ввиду особенность связанную с доступом к свойствам наружного класса: для вложенного класса надо писать: **val p = A.B()** (без пустых скобок у класса A), а для inner-класса: **val p = A().B()** (скобки у класса **A** обязательны). Это справедливо и для случая, когда у класса **A** есть аргументы.

Внутри классов можно создавать так называемые компаньоны (**companion object)**. Методы и свойства таких компаньонов могут быть вызваны по имени самого класса, без создания экземпляра класса:

```
class A {
    companion object B {
        fun f(x: Int, y: Int) = x + y
    }
}
fun main() {
    println(A.f(2, 3)) → 5
}
```

Название компаньона **B** при этом не используется и его можно вообще не указывать. Впрочем, при желании экземпляр можно и создать:

```
fun main() {
        val p = A.B
        println(p.f(2, 3)) → 5
}
```

Компаньон не может иметь аргументов и пустые скобки тоже не требуются. Если имя компаньона не указано, по умолчанию это имя будет **Companion**:

```
fun main() {
        val p = A.Companion
        println(p.f(2, 3))
}
```

## Классы Data (данные, информация)

Это необычный вид классов, предназначенных для хранения данных. В этих классах некоторые стандартные и вспомогательные функции механически выводятся из типа данных. При объявлении таких классов применяется модификатор **data**:

```
data class User(val name: String, val age: Int)
fun main() {
        val p = User("Иван", 25)
        println(p.toString()) → User(name=Иван, age=25)
}
```

В данном случае к экземплярам класса **User** оказалась применимой функция **toString()**, которая возвращает результат в определенном формате. Кроме **toString()** можно также применять методы **equals()**, **hashCode()**, **componentN()** и **copy()**.

Для data-классов должны выполнятся следующие требования:

- первичный конструктор должен иметь хотя бы один аргумент.
- все аргументы первичного конструктора должны быть с модификатором **val** или **var**.
- data-классы не могут быть **abstract**, **open**, **sealed**, или **inner**.

При автоматически генерируемых функциях используются только свойства, объявленные в конструкторе (в заголовке класса). Для

исключения свойств из генерируемой реализации их надо объявить в теле класса:

```
data class A(val x: String) { var y: Int = 25 }
fun main() {
        val p = A("Москва")
        println(p.toString()) →A(x=Москва)
}
```

Метод **equals()** позволяет проверять экземпляры data-класса на равенство:

```
data class A(var x: String) { var y: Int = 25 }
fun main() {
        val p1 = A("Москва"); val p2 = A("Москва")
        p1.y = 10; p2.y = 20
        println("p1 == p2: ${p1 == p2}") →p1 == p2: true
        println("p1 y ${p1.y}: ${p1}") → p1 y 10: A(x=Москва)
        println("p2 y ${p2.y}: ${p2}") → p2 y 20: A(x=Москва)
}
```

В этом примере использованы методы **toString()** и **equals()** неявно. Хотя свойство **y** имеет разные значения, экземпляры **p1** и **p2** считаются равными.

Метод **copy()** позволяет создавать копии экземпляров data-классов:

```
data class A(val x: String, val y: Int) {}
fun main() {
        val p = A(y = 25, x = "Виктор")
        val r = p.copy(y = 30)
        println(p.toString()) → A(x=Виктор, y=25)
        println(r.toString()) → A(x=Виктор, y=30)
}
```

При копировании можно изменять некоторые свойства. В примере показано, что значения аргументов можно передавать с указанием идентификаторов, то-есть, аргументы в Kotlin всегда именованные. Это позволяет вводить их в произвольном порядке:

```
val p = A(y = 25, x = "Виктор")
```

С помощью data-классов можно выполнять групповое присваивание (в документации это названо destructuring declarations):

```
data class A(val x: String, val y: Int)
fun main() {
        val p = A("Пушкин", 25)
```

```
    val(a, b) = p
    print("$a, $b") → Пушкин, 25
}
```

Типы тут выводятся автоматически. Kotlin имеет два стандартных класса **Pair** и **Triple**, которые позволяют групповое присваивание для двоек и троек переменных:

```
fun main() {
    val p = Triple("Пушкин", 25, true)
    val(a, b, c) = p
    print("$a, $b, $c") → Пушкин, 25, true
}
```

## Абстрактные классы

Абстрактные классы содержат объявления абстрактных свойств и методов (с модификатором **abstract**) без контента, их конкретная реализация выполняется в дочерних классах с помощью оператора **override.** Кроме абстрактных допустимы методы и свойства с реализацией и тогда он могут использоваться без перегрузки:

```
abstract class A {
    abstract fun f(x: Int): Int
    abstract val y: Int
    fun g(x: Double) { println(x) }
}
class B : A() {
    override fun f(x: Int) = x * x
    override val y = 77
}
fun main() {
    val p = B()
    println(p.f(5)) → 25
    println(p.y) → 77
    p.g(7.25) → 7.25
}
```

Как видим, для перегрузки родительский абстрактный класс и его члены не требуется снабжать модификатором **open**. Допустимо и обратное: абстрактный класс может наследовать обычному классу. При этом в дочернем классе можно перегружать члены родительского

класса, делая их абстрактными или с реализацией. Затем дочерний класс можно использовать как абстрактный.

```
open class A {
        open fun f() { println(77) }
}
abstract class B : A() {
        abstract override fun f()
}
class C : B() {
        override fun f() { println("Hello") }
}
fun main() { val p = C(); p.f() } → Hello
```

## This

В теле класса **this** означает ссылку на объект класса:

```
class A() {
        fun f() { println("aaa") }
        val a = this
        val r = this.f()
}
fun main() {
        val p = A()
        p.a.f() → aaa
        p.r → aaa
}
```

Применение **this** в такой ситуации ничего не даёт, пользу можно получить в более сложных ситуациях, например при наличии внутреннего класса:

```
class A() {
        fun f() { println("class A") }
        inner class B {
                fun f() { println("class B") }
                val a = this@A
                val b = this@B
                val c = this
```

```
    }
}
fun main() {
    val p = A().B()
    p.a.f() → class A
    p.b.f() → class B
    p.c.f() → class B
}
```

Знак @ позволяет указать, на объект какого класса ссылается слово **this**. Без применения этой метки **this** ссылается на объект внутреннего класса **B**.

# 5.Интерфейсы (interface)

Интерфейсы внешне похожи на классы, только объявляются со словом **interface**. Интерфейсы могут содержать методы и свойства, как абстрактные, так и с реализацией. Абстрактные методы не имеют тела, а абстрактные свойства не инициализированы. Классы могут подключать интерфейсы, один или более. Подключение интерфейсов синтаксически аналогично наследованию классов:

```
interface A {
    fun f(): Int →  абстрактный метод
    fun g() { println("Привет!") }
    val a: String → абстрактное свойство
}
class B(x: Int) : A {
    val x = x
    override fun f(): Int { return x * x }
    override val a = "Лариса"
}
fun main() {
    val p = B(3)
    println(p.f()) → 9
    p.g() → Привет!
    println(p.a) → Лариса
}
```

Здесь не требуются модификаторы **open** ни для подключаемых интерфейсов, ни для перегружаемых методов.

В интерфейсах могут объявляться свойства, абстрактные или с реализацией. Свойства могут быть инициализированы только с использованием функции доступа **get()** (обычно её называют *accessor*):

```
interface A {
        val x: Int
        val y: String get() = "Одесса"
        fun f() { println(x) }
}
class B : A {
        override val x: Int = 29
}
fun main() {
        val p = B()
        println(p.y) → Одесса
        p.f() → 29
}
```

Как видим, в интерфейсе объявленное абстрактное свойство **x** доступно в методе **f**.

Класс может подключать несколько интерфейсов:

```
interface A { val x: String }
interface B : A {
        val a: String; val b: String
        override val x: String get() = "$a $b"
}
data class C (override val a: String, override val b: String, val c:Int):B
fun main() {
        val p = C("Виктор", "Борисов", 77)
        println(p.x) → Виктор Борисов
        println(p.toString()) → C(a=Виктор, b=Борисов, c=77)
}
```

Сначала интерфейс **B** подключает интерфейс **A**, а затем класс **C** подключает интерфейс **B**. Кстати, строку **println(p.toString())** можно заменить на **println("$p")**, так как функция **toString()** при интерполяции вызывается автоматически.

Kotlin имеет разные способы применения интерфейсов. Рассмотрим их на простом примере функции, проверяющей чётность числа. Сначала обычный способ:

```
interface A { fun f(x: Int): Boolean }
class B : A {
    override fun f(x: Int): Boolean { return x % 2 == 0}
}
fun main() {
    val p = B(); println(p.f(8)) → true
}
```

Можно создать объект с подключённым интерфейсом без объявления нового класса. Для этого надо применить оператор **object**:

```
interface A { fun f(x: Int): Boolean }
val p = object : A {
    override fun f(x: Int): Boolean { return x % 2 == 0}
}
fun main() {
    println(p.f(8)) → true
}
```

Код можно сделать ещё более кратким и выразительным, применив так называемый функциональный интерфейс. Такой интерфейс можно применять только в том случае, если он имеет всего только один абстрактный метод . В документации такие интерфейсы кратко называют SAM (Single Abstract Method) интерфейсы. Для объявления SAM-интерфейса применяется модификатор **fun**. Объект в этом варианте создаётся просто передачей интерфейсу лямбда-функции (closure):

```
fun interface A { fun f(x: Int): Boolean }
val p = A { it % 2 == 0 }
fun main() {
    println(p.f(8)) → true
}
```

Аргумент абстрактного метода передаётся в closure, применяется стандартный идентификатор **it**. Функциональный интерфейс создаёт новый тип. В примере переменная **p** имеет тип **A** и его можно было указать явно:

```
val p: A = A { it % 2 == 0 }
```

Функциональный интерфейс можно заменить так называемым псевдонимом типа (*type alias*):

```
typealias G = (i: Int) -> Boolean
val f: G = { it % 2 == 0 }
fun main() {
    println("Число 7 чётное? - ${f(7)}") → Число 7 чётное? - false
}
```

В отличие от интерфейса псевдоним не создаёт объект, то-есть, не создаёт новый тип. Кроме того, псевдоним может иметь только один абстрактный метод, тогда как функциональный интерфейс кроме одного абстрактного метода может иметь другие не абстрактные методы.

В примере псевдонимом заменяется сигнатура функции. Для функций сигнатура и является типом. Псевдонимами можно также заменять любые другие типы. Это бывает удобно, когда название типа длинное и псевдоним позволяет избежать повторения этих длинных названий. Например тип хеша может иметь вид:

```
Map<String, Double>
```

Если такой тип требуется повторять, то лучше его заменить на псевдоним, например одной буквой:

```
typealias H = Map<String, Double>
```

# 6.Расширения (extensions)

Kotlin представляет возможность расширять функциональность классов или интерфейсов без наследования. В частности можно добавлять новые методы для классов и интерфейсов из сторонних библиотек. Этот механизм называется расширением (extension) классов или интерфейсов. Начнём с простейшего примера, в котором выполним расширение нашего собственного класса:

```
class A { val x: String = "Москва" }
fun main() {
    fun A.f() { println(this.x) }
    val p = A(); p.f() → Москва
}
```

Класс **A** имеет только одно свойство **x**. Мы добавили в класс функцию **f** для вывода значения этого свойства на терминал. При этом применяется такой синтаксис: перед названием класса надо вставить

слово **fun**, а затем через точку написать объявление добавляемой функции. Слово **this** в теле функции представляет экземпляр класса, а **this.x** это вызов значения свойства **x** соответственно. После этого функцию **f** можно применять на правах метода класса. Посмотрим ещё на вариант, когда класс имеет аргумент:

```
class A( x: Int) { val x = x }
fun A.f(a: Int, b: Int): Int { return (this.x * (a + b)) }
fun main() {
        val p = A(5)
        println(p.f(2, 3)) → 25
}
```

Выражение **val x = x** нужно для создания свойства класса, обычно для него применяют тот же идентификатор, что и для аргумента, мы это уже встречали.

Теперь рассмотрим расширение в библиотечном классе. Добавим функцию для перестановки элементов в списке в классе **MutableList**, представляющем изменяемые списки:

```
fun MutableList<Int>.f(i1: Int, i2: Int) {
        val x = this[i1]
        this[i1] = this[i2]
        this[i2] = x
}
fun main() {
        val m = mutableListOf(1,2,3,4,5)
        m.f(1, 3)
        print(m) → [1, 4, 3, 2, 5]
}
```

В данном случае для класса требуется указать параметр типа (у нас **<Int>**). Здесь слово **this** представляет объект класса - список. Аргументы функции представляют индексы двух элементов которые надо поменять местами. Интересно, что mutable-списки, объявленные со словом **val**, позволяют переставлять элементы, хотя, конечно, было бы логичнее применить здесь модификатор **var**.

Отметим несколько деталей, которые следуют иметь ввиду при расширениях. Посмотрим на такой пример:

```
fun main() {
        class A {
```

```
        fun f() { println("Москва") }
    }
    fun A.f() { println("Париж") }
    val p = A(); p.f()→ Москва
}
```

Здесь добавляемой функции мы дали то же название **f**, что и у метода расширяемого класса **A**. В таком варианте **p.f()** вызывает метод класса, а не добавленную функцию. Посмотрим ещё на пример, в котором функции с одинаковым названием добавляются в родительский и дочерний классы:

```
fun main() {
    open class A
    class B: A()
    fun A.f() = "Москва"
    fun B.f() = "Париж"
    fun g(s: A) { println(s.f()) }
    g(B()) → Москва
}
```

Функция **g** принимает экземпляр класса **A** (аргумент типа **A**) и вызывает метод **f** этого экземпляра. Такой функции можно передавать и экземпляры дочернего класса **B**, однако при этом вызываться будет всё равно метод родительского класса **A** в соответствии с типом аргумента **s**.

Многие стандартные функции не яляются nullable и при передаче им значения **null** генерируют исключение. Например в классе **Any**, который является суперклассом для всех классов Kotlin, есть метод **toString()**, не способный принимать значение null. Расширим класс **Any**, сделав его метод **toString()** nullable:

```
fun Any?.toString(): String {
    if (this == null) return "null"
    return toString()
}
fun main() {
    fun f(x: Any?): String { return x.toString() }
    println(f(77)) → 77
    println(f(null)) → null
}
```

Свойства тоже можно добавлять в класс аналогично методам:

```
class A {}
val A.x: String get() = "Москва"
fun main() {
        val p = A()
        print(p.x) → Москва
}
```

Здесь переменная **x** имеет доступ только по чтению (**val**). Несколько сложнее добавить переменную с доступом по чтению и по записи (**var**):

```
class A {}
var A.x: String
        get()  = "Москва"
        set(x) { println(x) }  →  Париж

fun main() {
        val p = A()
        println(p.x)  →  Москва
        p.x = "Париж"
}
```

Немного позже мы рассмотрим эти функции **get** и **set** (обычно их называют getter и setter).

Если класс имеет companion object, то его тоже можно расширять добавляя методы и свойства:

```
class A {
        companion object B { }
}
fun A.B.f() { println("companion B") }
fun main() {
        A.f() →companion B
}
```

Добавленный метод **f** вызывается по имени класса без создания экземпляра.

Можно создавать расширение класса внутри другого класса. Для этого надо передать экземпляр класса этому другому классу в качестве аргумента:

```
class A(val x: String) { fun f() { print(x) } }
class B(val y: A, val z: Int) {
        fun g() { print(z) }
```

```
    fun A.u() { f(); print(":"); g() }
    fun v() { y.u() }
}
fun main() {
    val p = B(A("Маша"), 25)
    p.v() → Маша:25
}
```

Аргумент **y** класса **B** имеет тип **A**, то-есть это экземпляр класса **A**. Строка

**fun A.u() { f(); print(":"); g() }**

создаёт функцию **u**, как расширение класса **A**. Далее эта функция доступна в классе **B**, как метод объекта **y** класса **A** и мы можем её вызвать в теле метода **v** класса **B**:

**fun v() { y.u() }**

Вне класса **B** эта функция недоступна. Например мы не можем её вызывать в теле функции **main**:

**A("Маша").u()** - здесь будет ошибка.

Расширения, объявленные как члены класса, могут иметь модификатор **open** и перегружаться в дочерних классах. В следующем примере показана сложная иеархия расширений и перегрузок функций в классах **A, B, C** и **D**.

```
open class A { }
class B : A() { }
open class C {
    open fun A.f() { println("A расширение в C") }
    open fun B.f() { println("B расширение в C") }
    fun g(b: A) { b.f() }
}
class D: C() {
    override fun A.f() { println("A расширение в D") }
    override fun B.f() { println("B расширение в D") }
}
fun main() {
    C().g(A()) → A расширение в C
    D().g(A()) → A расширение в D
    D().g(B()) → A расширение в D
}
```

# 7.Свойства

Свойства можно объявлять как изменяемые (mutable) со словом **var**, так и неизменяемые (только для чтения) со словом **val**:

```
class A {
        var x: String = "Людмила Сенчина"
        var y: String = "London"
        val z: String? = null
        val v: String = "123456"
}
fun f(a: A): A {
        a.x = "Анна Герман"
        a.y = "Киев"
        return a
}
fun main() {
        val r = f(A())
        println("${r.x}, ${r.y}, ${r.z}, ${r.v}") →
                                        Анна Герман, Киев, null, 123456
}
```

Функция **f** принимает экземпляр класса **A**, присваивает изменяемым свойствам новые значения и возвращает этот же изменённый объект. Доступ к свойствам объекта извне осуществляется с помощью методов доступа **get()** и **set()**. Обычно эти методы называют **getter** (получатель) и **setter** (установщик). Чаще всего эти методы вызываются неявно, как в нашем примере, но иногда их приходится вызывать явно. В общем виде синтаксис объявления изменяемого свойства имеет такой вид:
*var имя [: тип] [= значение]*
*[get()] [set()]*

Значение, getter и setter необязательны (optional). Тип необязательный только в том случае, когда он может быть выведен из контекста или когда getter возвращает тип. Приведу примеры:

```
var x = "мама"
```

Переменная **x** получает тип **String**, getter и setter вызываются неявно (по умолчанию, optional). Объявление в таком виде:

**var x**

приведёт к ошибке, так как тип не может быть выведен из контекста.

Неизменяемые (read-only) свойства отличаются от изменяемых (mutable) свойств тем, что применяется модификатор **val** вместо **va**r. При этом метод **set()** не используется.

**val x: Double?**

Здесь свойство **x** имеет тип **Double** (и **null-able**), метод **get()** вызывается неявно (optional).

**val x = 2**

Свойство **x** имет тип **Int**, getter optional. Посмотрим на пример явного использования **get()**:

```
class A(val x: Int, val y: Int) {
        val z: Int get() = this.x * this.y
}
fun main() {
        val p = A(3, 4)
        println("x=${p.x}, y=${p.y}, z=${p.z}") → x=3, y=4, z=12
}
```

Служебное слово **this** ссылается на аргумент класса. Явный вызов **get()** здесь можно опустить и тогда getter будет optional. Можно не использовать и **this**, как мы делали это везде раньше. Но при этом придётся ввести локальные переменные и тогда наш класс будет иметь вид:

```
class A(x: Int, y: Int) {
        val x = x; val y = y
        val z: Int = x * y
}
```

Обратите внимание на то, что модификатор **val** для аргументов класса при этом надо убрать, или применить новые идентификаторы для локальных переменных:

```
class A(val x: Int, val y: Int) {
        val t = x; val s = y
        val z: Int = t * s
}
```

Для изменяемых переменных применяется модификатор **var** и оба метода **get()** и **set()** явно или неявно:

```
var x: String get() = "hello"
set(y) { println(y) } → world
fun main() {
        x = "world"
        println(x) → hello
}
```

При инициализации переменной **x** вызывается setter **set(y)**, аргументу которого передаётся задаваемое значение. Затем выполняется блок метода **set**, в котором используется этот аргумент. Сама переменная **x** при этом не изменяется. Для изменения переменной надо применить метод **set** без аргумента:

```
var x: String = "hello"
        set
fun main() {
        x = "world"
        println(x) → world
}
```

# 8.Параметры типов (генерики)

Ранее мы уже применяли значение типа **List<Int>**, где **Int** в угловых скобках указывает, что элементы списка имеют тип **Int**. Kotlin позволяет здесь в угловых скобках вместо конкретного значения **Int** указать переменную, например **T**, которой в дальнейшем можно присваивать конкретные значения типа элементов. Такие переменные называются параметрами типа или генериками (generic – родовой). Применение параметров типа позволяет делать код более универсальным. Приведём пример, где класс имеет параметр типа :

```
class A<T>(t: T) {
        var x = t
}
fun main() {
        val p = A<Int>(5)
        println(p.x) → 5
        val r = A<String>("мама")
```

```
    println(r.x) → мама
    val m: List<Int> = listOf(1,2,3)
    val v = A<List<Int>>(m)
    println(v.x) → [1, 2, 3]
}
```

Введя параметр типа **T**, мы получили возможность задавать аргументу **t** класса **A** значения разных типов. В примере использовались типы **Int**, **String** и **List<Int>**. В последнем случае параметр типа сам имел свой параметр типа. Посмотрим на пример, в котором параметр типа принимает значение типа (класса), созданного пользователем:

```
class A(x: String, y: Int) {
    val x = x; val y = y
    fun f() { println("$x $y") }
}
class B(x:Double, y: Boolean) {
    val x = x; val y = y
    fun f() { println("$x $y") }
}
class C<T>(p: T) { val p: T = p }
fun main() {
    val r: A = A("Катя", 25)
    val s: C<A> = C<A>(r)
    s.p.f() → Катя 25
    val r1: B = B(1.234, true)
    val s1: C<B> = C<B>(r1)
    s1.p.f() →1.234 true
}
```

Как обычно, если тип может быть выведен из контекста, то явное указание типа можно опустить. В примере для объявления объектов **s** и **s1** можно было написать:

```
val s = C(r)
val s1 = C(r1)
```

Функции также могут иметь параметр типа, который располагается перед идентификатором функции:

```
fun <T> f(x: T): List<T> { return listOf(x) }
fun main() {
    val m = f<Int>(5)
```

```
        println(m) → [5]
        println(f<String>("aa")) → [aa]
}
```

При вызове функции значение параметра типа указывается после названия функции (**f<Int>**). Впрочем его тоже можно не указывать, если он может быть выведен автоматически:

```
val m = f(5)
println(f("aa"))
```

При программировании расширения функций с параметрами типа применяется особый синтаксис:

```
fun <S> f(x: S) {println(x)}
fun <T> T.f(x: T) {
        if (x is String) {println(x)} else {println("не String")}
}
fun main() {
        String.f("роза") → роза
        Int.f(77) → не String
}
```

При вызове расширенной функции тип надо указывать впереди названия функции через точку. В примере функцию **f**, выводящую свой аргумент на терминал, мы расширили, включив анализ аргумента.

Значения параметра типа можно ограничить, например:

```
fun <T : Comparable<T>> f(x: T, y: T) { if (x > y) {println("Yes")}
        else {println("No")} }
fun main() {
        f(7, 5) → Yes
        f(false, true) → No
//      val m1: List<String> = listOf("aa", "bb")
//      val m2: List<String> = listOf("cc", "dd")
//      f(m1, m2)→ error: type mismatch
}
```

Ключевое слово **Comparable** (сопоставимый) означает, что аргументы функции **f** должны быть сопоставимы (сравнимы). Например списки не удовлетворяют этому условию и если раскомментировать строки, то получим ошибку на этапе компиляции. Ограничений может быть больше одного и тогда они должны быть перечислены с применением слова **where**:

```
fun <T> f(m: List<T>, x: T): List<String>
        where T : CharSequence,
                T : Comparable<T> {
        return m.filter { it > x }.map { it.toString() }
}
fun main() {
        val m1: List<String> = listOf("d", "t", "y", "a", "m", "g", "i")
//      val m2: List<Int> = listOf(3, 1, 5, 4, 8)
        println(f(m1, "k")) → [t, y, m]
//      println(f(m2, 4)) → error: type mismatch
}
```

Ограничение **CharSequence** требует, чтобы элементы списка были представлены последовательностью знаков (Char). Тип аргументов должен удовлетворять обоим ограничениям одновременно. Тип **String** удовлетворяет обоим условиям, а тип **Int** удовлетворят условию **Comparable**, но не удовлетворят условию **CharSequence** и если раскомментировать строки, получим ошибку на этапе компиляции.

# 9.Объекты, делегаты и прочее

Ранее мы уже знакомились с применением в классах объектов — компаньонов. Кроме них имеются ещё некие разновидности объектов в классах. В частности, Kotlin позволяет применять так называемые объекты - выражения, представляющие собой некую модификацию экземпляра класса без создания самого класса. Например:

```
fun main() {
        val p = object {
                val x = "Привет"
                val y = "Мир"
                override fun toString() = "$x, $y!"
        }
        print(p) → Привет, Мир!
}
```

Для создания объекта (**p)** используется слово **object** с приданным ему блоком. В блоке объявлены с инициализацией две переменные **x** и **y** и перегружена стандартная функция **toString()** так, чтобы она

возвращала значения этих переменных, что позволяет передавать объект оператору вывода **print** непосредственно. В целом объект **p** действительно похож на экземпляр класса.

Объекты - выражения могут наследовать классам и интерфейсам, названия которых надо указать после слова **object** через двоеточие:

```
open class A(x: Int) {
        open val y: Int = x
}
interface B {
        fun f(x: Int): Int { return x * x }
}
}
val p: A = object : A(3) { override val y = 15 }
val r: B = object : B {
        override fun f(x: Int): Int {return x + x}
}
fun main() {
        println(p.y) → 15
        println(r.f(3)) → 6
}
```

Объекты могут возвращаться функцией, как результат. Такие объекты называют анонимными:

```
class A {
        private fun f() = object { val x: String = "Hello" }
        fun g() { val r = f(); println(r.x) }
}
fun main() {
        val p = A(); p.g() → Hello
}
```

Функции, возвращающие объект, должны быть с модификатором **private**. В следующем примере применим интерфейс и уже знакомый нам companion – объект:

```
interface A<T> {
        fun f(): T
        fun q(x: Int) = x + x
}
class B {
        companion object : A<B> {
```

```
            override fun f(): B = B()
            fun g(x: Int) = x * x
        }
        fun u(x: Int) = 3 * x
}
fun main() {
        val p: A<B> = B
        println(p.q(5)) → 10
        val r = B.f()
        println(r.u(7)) → 21
        println(B.g(5)) → 25
}
```

Интерфейс **A** мы снабдили параметром типа <**T**>, что позволяет создавать с использованием класса что-то вроде экземпляра интерфейса (**p**), хотя сами интерфейсы конструктора не имеют. Таким образом мы получаем непосредственный доступ к свойствам и методам интерфейса **(p.q(5))**. Компаньон-объект обеспечивает доступ к свойствам и методам компаньона по имени самого класса без создания экземпляра **(B.g(5))**. Наконец, перегруженная в компаньоне функция **f** создаёт экземпляры класса (**r**), обеспечивающие доступ к свойствам и методам самого класса **(r.u(7))**.

Делегаты позволяют создавать ещё более сложные связи между классами и интерфейсами. Для реализации делегата применяется служебное слово **by**. Посмотрим на примере:

```
interface A { fun f() }
class B(val x: Int) : A {
        override fun f() { println(x) }
}
class C(b: A) : A by b { val y: String = "Маша" }
fun main() {
        val p = B(10)
        val r = C(p)
        r.f() → 10
        println(r.y) → Маша
}
```

Здесь классу **C** передаётся в качестве аргумента экземпляр класса **B**, к которому подключён интерфейс **A**. Создавая с помощью конструктора класса **C** объект **r**, мы получаем доступ, благодаря

делегату (с помощью оператора **by**), к свойствам и методам интерфейса **A**, перегруженным в классе **B**. Делегатом называют блок после оператора **by.** Естественно, что объект **r** обеспечивает также доступ к собственным свойствам и методам класса **C**.

В делегате можно перегружать свойства и методы подключённого к классу интерфейса:

```
interface A { fun f(); fun g(); }
class B(val x: Int) : A {
        override fun f() { println(x) }
        override fun g() { println(x) }
}
class C(b: A) : A by b { override fun f() { println("Люда") } }
fun main() {
        val b = B(77)
        val p = C(b)
        p.f() → Люда
        p.g() → 77
}
```

Ещё один пример перегрузки свойств и методов интерфейса в делегате:

```
interface A { fun f(); fun g(); val y: String }
class B(val x: Int) : A {
        override fun f() { println(x) }
        override fun g() { println(x) }
        override val y = "B: Наташа"
}
class C(b: A) : A by b {
        override fun f() { println("Люда") }
        override val y = "C: Катя"
}
fun main() {
        val b = B(77)
        println(b.y) →B: Наташа
        val p = C(b)
        p.f() → Люда
        println(p.y) → C: Катя
        p.g() →77
}
```

В примере мы передали классу **C** экземпляр **b** класса **B**, но в экземпляре **p** класса **C** вызывается перегруженные в делегате метод **f** и свойство **y**. Значит, перегруженные в делегате методы и свойства имеют приоритет при их вызове.

В стандартной библиотеке Kotlin имеется несколько полезных делегатов, например:

```
val x: String by lazy { println("вычислили!"); "Привет!" }
fun main() {
    println(x)
    println(x)
}
```

Получаем такой результат:

*вычислили!*
*Привет!*
*Привет!*

Данный делегат вызывает функцию **lazy**, принимающую closure (в примере это **println("вычислили!")**) и возвращающую экземпляр **Lazy<T>**. Делегат позволяет получить ленивое свойство (**x**), при первом вызове которого выполняется closure и запоминается возвращаемый результат (у нас это строка **"Привет!"**). При последующих вызовах свойства **x** вычисление не выполняется, а просто возвращается сохранённый результат.

```
import kotlin.properties.Delegates
class A {
    var x: String by Delegates.observable("начало") {
        _, a, b ->
        println("$a -> $b")
    }
}
fun main() {
    val p = A()
    p.x = "первый" → начало -> первый
    p.x = "второй" → первый -> второй
    p.x = "третий" → второй -> третий
}

class A {
    var x: Int = 0
```

```
        @Deprecated("Use 'x' instead", ReplaceWith("x"))
        var y: Int by this::x
        @Deprecated("Use 'x' instead", ReplaceWith("x"))
        var z: Int by this::x
}
fun main() {
        val p = A()
        p.y = 42
        println(p.x) →42
        p.z = 77
        println(p.x) → 77
}
```

# 10.Коллекции

Kotlin поддерживает следующие типы коллекций:
- списки (**List**)
- массивы (**Array**) (Со списками и массивами ранее мы уже познакомились достаточно подробно).
- множества (**Set**). Множества содержат уникальные (без повторений) элементы. В отличие от списков и массивов порядок элементов в множествах не фиксируется и их выборка по индексу невозможна.
- ассоциированные списки (**Map**). Иначе их ещё называют словарями (dictionary) или хешами (hash). Ассоциированные списки состоят из множества пар ключ — значение. Ключи всегда уникальны, так как повторяющиеся ключи не имеют смысла. Значения могут дублироваться.

Kotlin позволяет манипулировать элементами коллекций независимо от типа элементов. Стандартная библиотека предлагает различные интерфейсы, классы и функции для создания коллекций и работы с их элементами любых типов. Интерфейс **Collection<T>** является корневым в иерархии коллекций и задаёт общее поведение неизменяемых коллекций: определение их размеров, проверку принадлежности и так далее. Интерфейс **Iterable<T>**, определяет итераторы для элементов коллекций.

Переменные с типом **Collection<T>** могут быть аргументами функций, например:

```
fun f(m: Collection<Any>) {
        for(s in m) print("$s ")
        println()
}
fun main() {
        val m1 = listOf(0.25, "Маша", false)
        f(m1) → 0.25 Маша false
        val m2 = setOf("один", "два", "три", "два")
        f(m2) → один два три
}
```

Как видим, аргумент **m: Collection<Any>** принимает коллекции разных видов, в примере это список **m1** и множество **m2**. Значение параметра типа **Any** позволяет иметь элементы разных типов. Для создания множества применяется функция **setOf**, аналогичная знакомой нам функции **listOf** для списков. Подобные функции есть и для других видов коллекций. Поскольку в множествах недопустимы повторяющиеся элементы, все дубли функцией **setOf** игнорируются.

Для изменяемых (mutable) коллекций есть интерфейс **MutableCollection<T>** с такими функциями, как **add**, **remove** и так двлее. Приведём пример с изменяемым списком:

```
fun main() {
        val x = "A long time ago in a galaxy far far away".split(" ")
        println(x) →[A, long, time, ago, in, a, galaxy, far, far, away]
        val m = mutableListOf<String>()
        x.filterTo(m) { it.length <= 3 }
        println(m) → [A, ago, in, a, far, far]
        val s = setOf("a", "A", "an", "An", "the", "The")
        m -= s
        println(m) → [ago, in, far, far]
}
```

Функция **split(" ")** разбивает строку на слова по пробелам, функция **mutableListOf<String>()** создаёт пустой изменяемый список, итератор **filterTo** выбирает из списка **x** элементы, в которых не более трёх букв и помещает результат в список **m**. Множество **s** содержит слова (в данном случае английские артикли), которые надо удалить из списка, если они там есть. Удаление выполняет оператор вычитания (краткая его форма **m -= s**). Рассмотрим далее все типы коллекций по порядку (в основном на примерах).

## Списки (List)

**List<T>** содержат упорядоченный набор элементов с индексами начиная от нуля и заканчивая значением **m.syze – 1**, где **m.syze** – размер списка **m**. В следующем примере показаны способы определения размера списка, выборки элементов и определения индекса элемента:

```
fun main() {
    val m = listOf("один", "два", "три", "четыре")
    println("Размер списка: ${m.size}") → Размер списка: 4
    println("Третий элемент: ${m.get(2)}") →
                                        Третий элемент: три
    println("Четвёртый элемент: ${m[3]}") →
                                    Четвёртый элемент: четыре
    println("Индекс элемента \"два\": ${m.indexOf("два")}") →
                                    Индекс элемента "два": 1
}
```

Как видим, есть два способа выборки элемента по индексу: обычный с указанием индекса в квадратных скобках (**m[3]**) и с применением функции get (**m.get(2)**).

Элементы списка могут дублироваться. Два списка равны друг другу, если имеют равные размеры и равные элементы на одних и тех же позициях:

```
data class A(var x: String, var y: Int)
fun main() {
    val p = A("Маша", 31)
    val r = listOf(A("Катя", 20), p, p)
    val s = listOf(A("Катя", 20), A("Маша", 31), p)
    println(r == s) → true
    p.y = 32
    println(r == s) → false
}
```

Следующий пример иллюстрирует применение функций, добавляющих, удаляющих и изменяющих элементы списка:

```
fun main() {
    val m = mutableListOf(1, 2, 3, 4)
    m.add(5); m.removeAt(1); m[0] = 0; m.shuffle()
```

```
    println(m) → [4, 3, 5, 0]
}
```

Функция **shuffle()** тасует элементы в случайном порядке.

### Множества (Set)

Пример использования множеств:

```
fun main() {
    val s = setOf(1, "Люда", true, 4)
    val s1  = setOf(4, true, "Люда", 1)
    println(s.first() == s1.first()) → false
    println(s.first() == s1.last()) → true
    println(s == s) → true
}
```

Независимо от расположения элементов в множествах, они считаются равными при одинаковом составе элементов. Хотя порядок элементов в множествах не фиксирован, функции **first()** и **last()** позволяют извлекать первый и последний элементы соответственно. Множества могут содержать элементы разных типов, тип таких множеств **Set<Any>**.

Имеется также и разновидность множеств **mutableSet**.

### Ассоциированные списки (Map, Hash)

Тип неизменяемого хеша имеет вид **Map<T,S>**, где **T** – тип ключей (key), **S** – тип значений (value) Ключи и значения в хешах разделяются словом **to** (в других языках применяется двоеточие или стрелка). На самом деле пары создаются инфиксной функцией **to**.

```
fun main() {
    val h = mapOf("Толя" to 12, "Борис" to 9, "Коля" to 15, "Петя"
to 11)
    println(h.keys) → [Толя, Борис, Коля, Петя]
    println(h.values) →[12, 9, 15, 11]
    if ("Толя" in h) println(h["Толя"]) → 12
    if (12 in h.values) println("12 есть в h") → 12 есть в h
    if (h.containsValue(9)) println("9 есть в h") → 9 есть в h
}
```

Методы **keys** и **values** возвращают в форме списков все ключи и все значения соответственно. Присутствие заданного значения в хеше проверяется с помощью служебного слова **in** или функции **containsValue()**. Извлечение значений из хеша выполняется также, как и для списков, только роль индексов здесь выполняют ключи. Тип хеша **h** можно было объявить явно:

**val h: Map<String, Int> = (…)**

Если тип хеша задать **Map<Any, Any>**, то значения ключей и значений могут иметь любые типы:

**val h: Map<Any, Any> = mapOf("Толя" to 12, 0.25 to "aaa", "Коля" to false, "Петя" to 7.03)**

Два хеша считаются равными, если они содержат одинаковые пары, независимо от их расположения:

```
fun main() {
    val h = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
    val h1 = mapOf("key2" to 2, "key1" to 1, "key4" to 1, "key3" to 3)
    println(h == h1) → true
}
```

Изменяемые хеши имеют тип **MutableMap<T,S>**. Для таких хешей можно добавлять новые пары или изменять значения для существующих ключей:

```
fun main() {
        val h = mutableMapOf("one" to 1, "two" to 2)
        h.put("three", 3)
        h["one"] = 11
        println(h) → {one=11, two=2, three=3}
}
```

Значит, метод **put** позволяет добавлять новые пары. Отметим ещё, что при выводе хеша на печать список пар выводится в фигурных скобках, а служебное слово **to** заменяется знаком равенства.

Хеши можно создавать с применением функции **apply**, что позволяет существенно экономить память. Применяется такой синтаксис:

**val h = mutableMapOf<String, Int>().apply { this["one"] = 1; this["two"] = 2 }**

Значит здесь используется оператор **this**. Такой способ применим только для изменяемых (mutableMap) хешей.

## Функции для работы с коллекциями

В Kotlin имеются функции **buildLisi, buildSet** и **buildMap**, позволяющие создавать соответствующие изменяемые коллекции:

```
fun main() {
   val h = buildMap {
            put("a", 1)
            put("b", 0)
            put("c", 4)
       }
       println(h) → {a=1, b=0, c=4}
}
```

Для хешей здесь применяется функция **put**. Тип элементов можно указывать или он может быть выведен из контекста. Для списков и множеств применяется функция **add**:

```
fun main() {
   val m: List<String> = buildList {
            add("a")
            add("b")
            add("c")
       }
       println(m) → [a, b, c]
}
```

Функции **emptyList()**, **emptySet()**, и **emptyMap()** позволяют создавать пустые коллекции:

```
val h = emptyMap<String, Int>()
println(h) → {}
```

Указывать тип элементов здесь необходимо.

Для списков можно использовать конструктор List в роли своеобразного итератора для инициализации:

```
fun main() {
       val m = List(3, { it * 2 })
       println(m) →[0, 2, 4]
}
```

Здесь **3** указывает размер списка, а блок **{ it * 2 }** вычисляет значения элементов. Стандартной переменной **it** конструктор передаёт значения индексов, образуя цикл. Вместо **List** можно применить **MutableList**.

Стандартные функции **toList(), toMutabltList(), toSet()** и подобные им для других видов коллекций позволяют создавать копии коллекций. Эти копии являются новыми коллекциями, а не ссылками на один и тот же экземпляр. Покажем на примере:

```
class Person(var name: String)
fun main() {
        val p = Person("Маша")
        val m = mutableListOf(p, Person("Коля"))
        val m1 = m.toList()
        m.add(Person("Катя"))
        p.name = "Люда"
        m.forEach {print("${it.name} ")}; println() → Люда Коля Катя
        m1.forEach {print("${it.name} ")} → Люда Коля
}
```

Как видим, добавление элемента **Person("Катя")** к списку **m** не изменило копию **m1**. Изменение свойства **name** в экземпляре класса **p** (**Люда** вместо **Маша**) конечно отразилось в обоих списках. Если заменить строку

```
val m1 = m.toList()
```

на строку

```
val m1 = m
```

то все изменения отразятся на обоих списках, так как в этом случае **m** и **m1**ссылаются на один и тот же объект. При выводе мы в обоих случаях получим: *Люда Коля Катя*. Есть возможность объявить список **m1** неизменяемым:

```
val m1: List<Int> = m
```

Значит, изменяемым списком можно инициировать неизменяемый список.

Эти же функции применяются для конвертирования одних видов коллекций в другие. Например можем трансформировать список **m** в множество **s** с помощью функции **toMutableSet()**:

```
fun main() {
        val m = mutableListOf(1, 2, 3)
        val s = m.toMutableSet()
        s.add(3); s.add(4)
        println(s) →[1, 2, 3, 4]
}
```

Ранее мы уже много раз применяли итераторы для работы с элементами коллекций, например такие, как **forEach**, **filter**, **map**. Все они базируются на функции **iterator()** из интерфейса **Iterable<T>**. Прямое использование функции **iterator()** довольно замысловато, как это можно видеть на следующем примере:

```
fun main() {
    val m = listOf("one", "two", "three", "four")
    val i = m.iterator()
    while (i.hasNext()) { println(i.next()) } → one two three four
}
```

Применив функцию **iterator()** к коллекции, получим некий объект (в примере он обозначен **i**), для которого можно вызвать функцию **hashNext().** Эта функция вернёт **true**, если есть первый элемент коллекции, то-есть коллекция не пустая. Применив затем к объекту **i** функцию **next()**, получим значение первого элемента и будет сделан переход к следующему элементу. С помощью оператора цикла **while** можно последовательно извлечь все элементы. Когда будет вызван последний элемент, функция **hashNext()** выдаст **false** и цикл закончится.

Мы уже знаем, что для коллекций можно использовать цикл **for**:

```
for (x in m) {…}
```

На самом деле здесь также использована функция **iterator()** неявно.

Для списков есть также специальный итератор **listIterator()**, который отличается от **iterator()** только тем, что позволяет перебирать элементы в обратном порядке:

```
fun main() {
    val m = listOf("one", "two", "three", "four")
    val i = m.listIterator()
    while (i.hasNext()) i.next(); → Прямая итерация
    println("Обратная итерация:")
    while (i.hasPrevious()) {
        print("Индекс: ${i.previousIndex()}")
        println(", Значение: ${i.previous()}")
    }
}
```

Получим результат:

*Обратная итерация:*

*Индекс: 3, Значение: four*
*Индекс: 2, Значение: three*
*Индекс: 1, Значение: two*
*Индекс: 0, Значение: one*

Значит, для обратной итерации применяются функции **hasPrevious()** и **previous()**. Функция **previousIndex()** позволяет получать индексы элементов. При прямой итерации индексы позволяет получить функция **nextIndex()**.

Для изменяемых коллекций функция **remove()** позволяет удалять элементы:

```
fun main() {
        val m = mutableListOf("one", "two", "three", "four")
        val i = m.iterator()
        i.next()
        i.remove()
        println(m) →[two, three, four]
}
```

Функция **add()** позволяет добавлять, а функция **set()** - заменять элементы списка по индексу, устанавливаемому функцией **next()**:

```
fun main() {
        val m = mutableListOf("one", "four", "four")
        val i = m.listIterator()
        i.next(); i.add("two")
        i.next(); i.set("three")
        println(m) → [one, two, three, four]
}
```

Ранее мы уже применяли ранги. Ранги могут иметь типы **Int, Long, Char**:

```
fun main() {
        val r = 1..5
        for(i in r) { print("$i ") }; println() → 1 2 3 4 5
        val r1 = 4L..7L
        for(i in r1) { print("$i ") }; println() →4 5 6 7
        val r2 = 'c'..'f'
        for(i in r2) { print("$i ") }; println() → c d e f
}
```

Ранги позволяют извлекать элементы с заданным шагом, применяя ключевое слово **step**. Если вместо точек поставить слово **downTo**,

порядок элементов меняется на обратный. Если вместо точек поставить слово **until**, ранг не будет содержать верхнюю границу:

```
fun main() {
        for (i in 3..9 step 2) print("$i "); println() → 3 5 7 9
        for (i in 'ю' downTo 'ш') print("$i ") →ю э ь ы ъ щ ш
        val r = 3 until 8; print(r) → 3..7
}
```

Обратный поядок можно получить также с помощью функции **reversed()**:

```
fun main() {
        for (i in (1..4).reversed()) print(i) → 4321
}
```

Есть также метод **rangeTo**, создающий ранг. Он применяется к начальному значению, а конечное значение задано аргументом:

```
fun main() {
        val r = 3.rangeTo(7)
        print(r) →3..7
}
```

Как и другие виды коллекций, ранги позволяют применять итераторы, например **filter**:

```
fun main() {
        println((1..10).filter { it % 2 == 0 }) → [2, 4, 6, 8, 10]
}
```

Результат получили в форме списка.

## Последовательности

Кроме коллекций Kotlin позволяет использовать ещё одну разновидность структурных объектов (контейнерных типов по документации). Это так называемые последовательности (sequences), представляющие тип **Sequence<T>**. Для работы с последовательностями применимы те же функции (в частности итераторы), что и для коллекций, но их реализация основана на другом подходе. На каждом шаге итерации возвращается как результат некая промежуточная коллекция, которая затем используется на следующем шаге. При этом принят ленивый подход — результат вычисляется только при наличии запроса.

Для создания последовательности применяется функция **sequenceOf()**:

```
val s = sequenceOf("four", "three", "two", "one")
```

Последовательность можно также создавать путём трансформации коллекций, например списков или множеств, с использованием метода **asSequence()**:

```
fun main() {
        val m = listOf("one", "two", "three", "four")
        val s = m.asSequence()
        s.forEach {print("$it ")} → one two three four
        val m1 = s.toList()
        println(m1) →[one, two, three, four]
}
```

Последовательность нельзя передать оператору **print** непосредственно, приходится применять итератор. Метод **toList()** (или **toSet()**) позволяет выполнять обратное преобразование.

Функция **generateSequence()** принимает closure и позволяет вычислять элементы последовательности (подобно итератору **map** для коллекций):

```
fun main() {
        val s = generateSequence(1) { it + 2 }
        println(s.take(5).toList()) → [1, 3, 5, 7, 9]
//      println(s.count()) → s бесконечна, раскомментировать нельзя
}
```

Аргумент **1** — первый элемент. Количество вычисленных элементов определяет функция **take()**, как раз она и делает запрос. А вообще последовательность **s** бесконечна. Генерация элементов обрывается, если очередному элементу задать значение **null**:

```
fun main() {
        val s = generateSequence(1) { if (it < 8) it + 2 else null }
        println(s.count()) → 5
        println(s.toList())→ [1, 3, 5, 7, 9]
}
```

Функция **sequence** принимает блок, содержащий функции **yield()** или **yieldAll()**, с помощью которых генерирует последовательность.

```
fun main() {
        val s = sequence {
                yield(1)
```

```
            yieldAll(listOf(3, 5))
            yieldAll(generateSequence(7) { it + 2 })
        }
        println(s.take(5).toList()) →[1, 3, 5, 7, 9]
}
```

Функция **yield()** добавляет один элемент в последовательность, а функция **yieldAll()** добавляет заданную последовательность или бесконечную последовательность, генерируемую функцией **generateSequence**.

Следующие два примера демонстрируют различие в применении ленивого подхода при работе со списками и с последовательностям. Сначала пример со списком:

```
fun main() {
  val s = "У лукоморья дуб зелёный златая цепь на дубе
том".split(" ")
  val m = s.filter { print("$it "); it.length > 3}; println()
  val m1 = m.map { print("${it.length} "); it.length }; println()
  val m2 = m1.take(4)
  print("Длина первых 4 слов длиннее чем 3 знака: ")
  println(m2)
}
```

Получаем такой результат:

*У лукоморья дуб зелёный златая цепь на дубе том*
*9 7 6 4 4*
*Длина первых 4 слов длиннее чем 3 знака: [9, 7, 6, 4]*

Трансформируем список в последовательность с помощью функции **asSequence()** и выполним с ней те же операции, что и со списком в предыдущем примере:

```
fun main() {
  val s = "У лукоморья дуб зелёный златая цепь на дубе
том".split(" ")
  val s1 = s.asSequence()
  val m = s1.filter { print("$it "); it.length > 3 }; println()
  val m1 = m.map { print("${it.length} "); it.length }; println()
  val m2 = m1.take(4)
  println("Длина первых 4 слов длиннее чем 3 знака: ")
  println(m2.toList())
}
```

Получим другой результат:
*Длина первых 4 слов длиннее чем 3 знака:*
*У лукоморья 9 дуб зелёный 7 златая 6 цепь 4 [9, 7, 6, 4]*

Вследствие ленивого подхода итераторы выполняют свои действия только тогда, когда последовательность **m2** трансформируется в список с помощью функции **toList()**.

# 11.Функции

Не повторяя уже известных нам сведений о функциях, рассмотрим здесь дополнительные возможности и ограничения. Начнём с примера:

```
var t = 0
fun f(x:Int, y: Int): Int {  t = x - y; return x + y}
fun main() {
      println(f(7, 2)) → 9
      println(t) → 5
}
```

Для получения результата всегда должен применяться оператор **return**; он же выполняет и выход из функции и потому должен стоять на последнем месте. Выражение после **return** можно не заключать в круглые скобки. В Kotlin блок автоматически не возвращает последнего вычисленного значения, как это практикуется во многих современных языках. Например, если мы напишем:

```
val a = 7; val b = 3
val t = { a + b; a - b }; println(t)
```

то получим: *$main$lambda…* Значит, блок в такой ситуации компилятор считает лямбда-функцией (о них позже). Тут можно применить специальную функцию **run**, которая вычисляет стоящий за нею блок:

```
val t = run { a + b; a - b }; println(t) → 4
```

Теперь получим последнее вычисленное значение в блоке. Подробнее функцию **run** рассмотрим позже.

Список аргументов функции (также и класса) можно располагать на нескольких строках:

```
fun f(x: Int, y: Double,
      z: String,
```

```
    ): String  { return "$x $y " + z}
fun main() {
    println(f(5, 2.07, "Маша")) → 5 2.07 Маша
}
```

Допустимо ставить запятую после последнего аргумента в списке, компилятор её игнорирует.

Всегда можно применять аргументы по умолчанию. При задании их значений можно употреблять выражения, а в них допустимо даже использовать переменные из этого же списка аргументов. Эти переменные должны располагаться в списке впереди выражения:

```
fun f(x: Int, y: Int = 5, z: Int = x + y): Int {
    return x + y + z
}
fun main() {
    println(f(3)) → 16
}
```

Обычно аргументы по умолчанию ставят в конце списка, а если это правило не соблюдается, то при вызове функции надо применять именованные переменные:

```
fun f(y: Int = 5, x: Int, z: Int = x + y): Int {
    return x + y + z
}
fun main() {
    println(f(x = 3)) → 16
}
```

Переменным по умолчанию всегда можно задать новые значения:

```
println(f(x = 3, z = 2)) → 10
```

Если метод с аргументами по умолчанию перегружается в дочернем классе, то там повторять значение по умолчанию не требуется:

```
open class A { open fun f(x: Int = 5): Int { return 2 * x } }
class B : A() { override fun f(x: Int): Int { return 3 * x } }
fun main() {
    val p = B()
    println(p.f()) → 15
}
```

В перегруженном методе нельзя задать и новое значение по умолчанию.

**class B : A() { override fun f(x: Int = 7): Int { return 3 * x } }**

В таком варианте получим ошибку.

Можно маркировать аргумент функции модификатором **vararg**. Это предоставляет дополнительную свободу при вводе значений аргументов:

```
fun <T> f(vararg ts: T): List<T> {
        val r = mutableListOf<T>()
        for (t in ts) r.add(t); return r
}
fun main() {
        val m = f(1, 2, 3); println(m) → [1, 2, 3]
}
```

Функция **f** создаёт список, в её теле переменная **ts** трактуется, как массив. Аргументами функции **f** может быть всё, что угодно. Например, это могут быть лямбда-выражения:

```
fun <T> f(vararg ts: T): List<T> {
        val r = mutableListOf<T>()
        for (t in ts) r.add(t); return r
}
fun main() {
        val g = f({ x:Int, y: Double -> x * y }, { x:Int, y: Double -> x + y })
        println(g[0](3, 7)) → 21.0
        println(g[1](4, 9)) → 13.0
}
```

Здесь тело лямбда-выражений может быть любым, но число их аргументов должно быть одинаковым одного и того же типа.

Только один аргумент в списке может быть vararg. Если он находится не в конце списка можно использовать синтаксис именованного аргумента. Если среди аргументов есть массив, надо использовать spread-оператор (звёздочка перед идентификатором массива):

```
val a = arrayOf(1, 2, 3)
val m = f(-1, 0, *a, 4)
```

Таким образом, применение vararg-аргументов в сочетании с параметрами типа делает функцию более универсальной.

## Рекурсия

Kotlin позволяет программировать в функциональном стиле. В частности, можно применять рекурсивные функции. Запрограммируем решение уравнения: *cos(x) = x* методом последовательных приближений:

```
import kotlin.math.*
val eps = 1E-10
fun f(x: Double = 1.0): Double =
        if (abs(x - cos(x)) < eps) x else f(cos(x))
fun main() {
        println(f()) → 0.7390851331706995
}
```

Здесь константа **eps** задаёт допустимую погрешность. Без рекурсии это было бы примерно так:

```
import kotlin.math.*
val eps = 1E-10
fun f(): Double {
        var x = 1.0
        while (true) {
                val y = cos(x)
                if (abs(x - y) < eps) return x
                x = cos(x)
        }
}
fun main() {
        println(f())→ 0.7390851331706995
}
```

Кстати, уравнение *sin(x) = x* имеет такое решение: *0.0008434325396267086*

Наверное было бы неправильно не привести самый популярный пример рекурсивной функции — вычисление факториала:

```
var r = 1
fun f(n: Int): Int {
        if (n <= 1) {return r} else {r = r * n; return f(n - 1)}
}
fun main() {
        println(f(5)) → 120
}
```

Имеется возможность объявлять рекурсивную функцию с модификатором tailrec:

**tailrec fun f(n: Int): Int { … }**

Это вариант так называемой хвостовой рекурсии. В этом случае рекурсивный вызов (функция вызывает саму себя) должен быть последней операцией в теле функции, после неё не должно быть никакого кода. Оба приведённых выше примера соответствуют этому требованию.

## Анонимные функции и лямбда-выражения

Анонимные функции объявляются также, как и именованные, надо только опустить идентификатор функции:

```
fun (x: Int, y: Int): Int {return x + y}
```

Анонимной функцией можно инициировать какую-нибудь переменную, которая сама становится функцией:

```
val f = fun (x: Int, y: Int): Int {return x + y}
fun main() {
    println(f(2, 3)) → 5
}
```

Можно, конечно, использовать и **var**, тогда переменная **f** будет изменяемой (mutable) и ей можно присваивать новые значения:

```
var f = fun (x: Int, y: Int): Int {return x + y}
fun main() {
    println(f(2, 3)) → 5
    f = fun (a: Int, b: Int): Int {return a * b }
    println(f(2, 3)) → 6
}
```

Когда тело функции представлено одним выражением, как в нашем примере, можно применить вариант со знаком равенства, а тип результата тогда не обязателен:

```
var f = fun (x: Int, y: Int) = x + y
```

Поскольку в Kotlin всякая сущность (term) является объектом, то и функции тоже представляют объекты и имеют тип. Тип функции обычно называют сигнатурой и для функции **f** в предыдущем примере она имеет вид:

```
(Int, Int) -> Int
```

В сигнатуре в круглых скобках указываются типы всех аргументов функции, а после стрелки — тип возвращаемого результата. При объявлении функции тип всегда можно указать явно:

```
var f: (Int, Int) -> Int = fun (x: Int, y: Int): Int {return x + y}
```

Часто тип не указывают, если компилятор может вывести его из контекста. Иногда удобно заменять сигнатуру кратким псевдонимом, который создаётся с помощью директивы **typealias**:

```
typealias T = (Int) -> Int
val f: T = { x ->  7 + x }
fun main() {
    println(f(3)) → 10
}
```

Поскольку функция — объект, то она может быть аргументом для другой функции:

```
fun f(x: Int, y: Int, g: (Int, Int) -> Int): Int { return g(x, y) }
val r = fun (a: Int, b: Int) =  a + b
fun main() {
    println(f(2, 3, r)) → 5
}
```

Как и для всех других аргументов, для функции здесь должен быть указан тип - сигнатура. Аргументу-функции можно задавать значение по умолчанию:

```
fun f(x: Int, y: Int, g: (Int, Int) -> Int =
    fun (a: Int, b: Int) = a + b ): Int { return g(x, y) }
fun main() {
    println(f(2, 3)) → 5
}
```

Или задавать значение при вызове:

```
fun f(x: Int, y: Int, g: (Int, Int) -> Int): Int { return g(x, y) }
fun main() {
    println(f(2, 3, fun (a: Int, b: Int): Int = a * b)) → 6
}
```

Функцию, как результат может возвращать другая функция (higher-order function):

```
fun f(): (Int, Int) -> Int  { return fun (x: Int, y: Int) = (x + y) * 2 }
fun main() {
    println(f()(2, 3)) → 10
}
```

Здесь функция **f** без аргументов возвращает функцию, которую можно вызвать, передав ей значения её аргументов. Поскольку сигнатура результата задана, то типы аргументов анонимной функции можно не указывать:

```
fun f(): (Int, Int) -> Int  { return fun (x, y) = (x + y) * 2 }
fun main() {
    println(f()(2, 3)) → 10
}
```

Можно делать и так:

```
val r = fun (x: Int, y: Int): Int = (x + y) * 3
fun f(w: (Int, Int) -> Int): (Int, Int) -> Int  { return w }
fun main() {
    println(f(r)(2, 3)) → 15
}
```

Как видим, в этих примерах необходима анонимная функция; обычную именованную функцию так использовать невозможно.

Кроме анонимных функций можно применять лямбда-выражения (термин из теории функций). Их синтаксис покажем на примере:

```
{ x, y -> x + y }
```

Лямбда-выражения всегда в фигурных скобках. Фактически это синтаксический сахар для анонимной функции:

```
fun (x, y) { return x + y }
```

В действительности требуется указывать типы. В самом общем виде объявленная с помощью лямбда-выражения функция имеет вид:

```
val f: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

На самом деле здесь можно опустить сигнатуру или типы у аргументов **x** и **y**. Лямбда-выражение можно передавать функции как аргумент:

```
fun f(x: Int, y: Int, g: (Int, Int) -> Int ): Int { return g(x, y) }
fun main() {
    println(f(2, 3, { x: Int, y: Int -> x + y })) → 5
}
```

Здесь можно было сделать и так:

```
fun f(x: Int, y: Int, g: (Int, Int) -> Int) = g(x, y)
```

Иногда удобнее сначала инициировать переменную лямбда-выражением:

```
val r = { x: Int, y: Int -> x + y }
fun f(x: Int, y: Int, g: (Int, Int) -> Int): Int { return g(x, y) }
```

```
fun main() {
        println(f(2, 3, r)) → 5
}
```

Если аргумент-анонимная функция (или лямбда-выражение) в списке аргументов стоят после других аргументов, то допустимы разные способы задания значений аргументов. Покажем возможные варианты на примере:

```
fun f(x: Int = 3, y: Int = 4, g: (Int, Int) -> Int) = g(x, y)
fun main() {
        println(f(g = { x, y -> x + y })) → 7
        println(f { x, y -> x + y }) → 7
        println(f(5) { x, y -> x + y })  → 9
}
```

Варианты:

**g = { x, y -> x + y }** - обычный именованный аргумент

**f { x, y -> x + y }** - компилятор сам определяет, что здесь аргумент-функция, а круглые скобки не требуются. Круглые скобки не нужны также тогда, когда аргумент-функция единственный в списке.

**{ x, y -> x + y }** - аргумент-функцию можно выносить из списка за круглые скобки.

Если лямбда-функция имеет один аргумент, то можно использовать такой синтаксический сахар:

```
fun f(x: Int, g: (Int) -> Int): Int { return g(x) }
fun main() {
        val r = f(5) { it * it }
        println(r) → 25
}
```

Здесь аргумент передаётся блоку со стандартным идентификатором **it**, а стрелка -> не требуется. Такая форму удобно применять для стандартных методов, например:

```
val s = listOf(7, 3, 1, 9, 8)
fun main() {
        val r = s.filter { it > 3 }
        println(r) → [7, 9, 8]
}
```

Без применения такого стиля пришлось бы написать:

```
val r = s.filter { x -> x > 3 }
```

Этот синтаксис позволяет создавать цепочки методов:

```
val s = listOf("собака", "гусь", "корова", "бегемот")
fun main() {
    val r = s.filter { it.length == 6 }.sortedBy { it }.map
{ it.uppercase() }
    println(r) → [КОРОВА, СОБАКА]
}
```

Такой код в документации называется LINQ-style code,

В Kotlin любой блок, заключённый в фигурные скобки, считается лямбда-выражением. Его можно выполнить с помощью специальной функции **run**. Результатом будет значение последнего выражения в блоке:

```
val x = 4; val y = 5; val r = { x + y; x * y }
fun main() {
    println(run(r)) → 20
}
```

В документации такой блок называют closure. Вообще-то так можно называть любые функции, которые могут служить аргументами других функций.

Не используемые переменные в лямбда-выражении можно заменять знаком подчёркивания (placeholder):

```
val h = mapOf("Россия" to "Москва", "Китай" to "Пекин",
    "Индия" to "Дели")
fun main() {
    h.forEach { (_, v) -> print("$v ") } → Москва Пекин Дели
}
```

### Функции с «приёмником»

Kotlin позволяет применять функции с так называемым приёмником (function type with receive, не очень ясно почему тут принят такой термин). Тип этого приёмника (или получателя) указывается в сигнатуре впереди типов аргументов и отделяется от них точкой. Посмотрим на примере:

```
fun main() {
    val f: Double.(Int) -> Double = { x -> plus(x) }
    println(f(2.9, 3)) → 5.9
    val y = 7.2
    println(y.f(3)) → 10.2
```

**}**

Здесь приёмник имеет тип **Double**, а аргумент — тип **In**t. В лямбда-выражении использована функция суммирования **plus**. Автоматически суммирование выполняется с приёмником. Функцию **f** можно вызывать, как функцию с двумя аргументами, или применять точечную нотацию, введя приёмник явно. Можно также применить указатель **this**, который будет указывать на приёмник. Теперь будет достаточно знака суммирования:

```
fun main() {
        val f: Double.(Int) -> Double =  { x -> x + this }
        println(f(2.5, 7)) → 9.5
        println(2.5.f(7)) → 9.5
}
```

Точечная нотация тут также допускается. Можно применить вариант с анонимной функцией:

```
fun main() {
        val f = fun Int.(x: Int, y: Int): Int = this + x * y
        println(f(3,4,5)) → 23
        println(3.f(4,5)) → 23
}
```

В таком варианте допускается тело функции и в блоке, когда это тело содержит более одного выражения:

```
fun main() {
        val f = fun Int.(x: Int, y: Int): Int { val r = x + y; return this * r }
        println(f(3,4,5)) → 27
        println(3.f(4,5)) → 27
}
```

Посмотрим на ещё один пример:

```
fun main() {
        val f: String.(Int) -> String = { x -> this.repeat(x) }                (1)
        val f1: (String, Int) -> String = f                                    (2)
        fun g(fi: (String, Int) -> String): String { return fi("hello", 3) }
(3)
        println(g(f)) → hellohellohello                                        (4)
        println(f1("World", 4)) → WorldWorldWorldWorld
}
```

Я поставил номера, чтобы прокомментировать строчки (не забывайте удалять посторонний текст при копировании для тестирования).

В строке (1) объявлена функция с приёмником. Строка (2) показывает, что можно получить обычную функцию (без приёмника), если указать её сигнатуру и инициировать функцией с приёмником. При этом аргумент, заменяющий приёмник всегда стоит первым в списке. В строке (3) объявлена функция **g** с аргументом-функцией **fi** не имеющей приёмника. В строке (4) показано, что допустимо этому аргументу передать значение в виде функции с приёмником.

### Функции let, run, with, apply, also

Стандартная библиотека Kotlin содержит пять указанных в заголовке функций, предназначенных для выполнения блока кода, который рассматривается, как отдельное пространство имён (scope). Поэтому эти функции называются scope function (затрудняюсь перевести). От обычных эти функции отличаются способом их использования. Иногда применение этих функций делает код более кратким (лаконичным).

### Let

Пусть требуется отфильтровать из списка длин элементов типа **String** числа больше заданного. Сначала обычное применение итераторов **map** и **filter**:

```
fun main() {
        val m = mutableListOf("one", "two", "three", "four", "five")
        val r = numbers.map { it.length }.filter { it > 3 }
        println(r) → [5, 4, 4]
}
```

Теперь с функцией **let:**

```
fun main() {
        val m = mutableListOf("one", "two", "three", "four", "five")
        m.map { it.length }.filter { it > 3 }.let { println(it) } →  [5, 4, 4]
}
```

Значит, если после итератора **filter** добавить через точку функцию **let** с приданным ей блоком, то переменной **it** в этом блоке будет

передан результат предыдущего действия, в данном случае отфильтрованный список, который, например, можно в блоке вывести на печать. Можно добавлять последовательно несколько функций **let**, например так:

```
fun main() {
        val m = mutableListOf("one", "two", "three", "four", "five")
        m.map { it.length }.filter { it > 3 }.let { it.sum()
         }.let {println(it)} → 13
}
```

Блок **{ it.sum()** } вычисляет сумму элементов отфильтрованного списка.

Если блок после **let** содержит только одно выражение, то можно применять method reference (::) вместо лямбда:

```
fun main() {
        val m = mutableListOf("one", "two", "three", "four", "five")
        val f: Int.(Int) -> Int = { x -> plus(x) }
        m.map { it.length }.filter { it > 3 }.let {it.sum()}.
        let {it.f(3)}.let (::println) → 16
}
```

При работе с nullable-переменными функция **let** имеет средство для безопасного кода. Для этого надо поставить знак вопроса перед обращением к функции **let**:

```
fun g(x: String) { println(x) }
fun main() {
        val y: String? = "Hello"
        y?.let {
                println(it) →  Hello
                g(it) →  Hello
        }
}
```

Здесь функции **g** передается  nullable-значение переменной **y** типа **String?**, хотя объявленный аргумент функции **g** имеет тип **String**. Если убрать вопросительный знак - **y.let {…}**, то получим ошибку: *g(it) - inferred type is String? but String was expected.* (Это значит, что функция g получила значение типа **String?**, а требовался тип **String**)

В блоке функции **let** вместо стандартной **it** можно использовать свою локальную переменную:

```
fun main() {
```

```
    val m = listOf("one", "two", "three", "four")
    val x = m.first().let { y -> println("Первый элемент:'$y'")
        if (y.length >= 5) y else "!" + y + "!"
    }.uppercase()
    println("После модификации: '$x'")
}
```

Получим такой результат:

*Первый элемент:'one'*
*После модификации: '!ONE!'*

Здесь локальная переменная **y** объявлена в лямбда-выражении и используется в блоке далее.

## With

Функция **with** имеет аргумент, который доступен в приданном ей блоке по ссылке **this**.

```
fun main() {
    val m = mutableListOf("one", "two", "three")
    with(m) {
        println(this) → [one, two, three]
        println(size) → 3
    }
}
```

При этом часто **this** можно использовать неявно (вместо **println(this.size)** в нашем примере **println(size)**).

В общем, функцию **with** можно трактовать так: с объектом полученным, функцией в качестве аргумента, мы делаем то-то и то-то:

```
fun main() {
    val m = listOf(3, 7, 2, 9)
    with(m) {
        val r = first() + last()
        println(r) → 12
    }
}
```

## Run

Функция **run** делает почти то же, что и **let**, только в приданном блоке доступны свойства и методы объекта непосредственно, без использования переменной **it**. В примере ниже показана работа обеих функций для сравнения:

```
class A(var name: String, var age: Int) {
    fun f(): String = "Маша"
    fun g(x: String): String = " '$x'"
}
fun main() {
    val p = A("Катя", 25)
    val r1 = p.run {
        age = 20
        g(f() + " $age лет")
    }
    val r2 = p.let {
        it.age = 20
        it.g(it.f() + " ${it.age} лет")
    }
    println(r1) → 'Маша 20 лет'
    println(r2) → 'Маша 20 лет'
}
```

Функцию **run** часто применяют просто для выполнения блока, содержащего одно или несколько выражений, например:

```
fun main() {
    val s = run {
        val d = "0-9"
        val h = "A-Fa-f"
        val s = "+-"
        Regex("[$s]?[$d$h]+")
    }
    for (match in s.findAll("+123 -FFFF !%*& 88 XYZ")) {
        println(match.value)
    }
}
```

В результате получим:

*+123*
*-FFFF*
*88*

Пример иллюстрирует применение регулярных выражений, которые рассмотрим позже.

### Apply

Функция **apply** вызывается для объекта и выполняет действия, определённые приданным блоком. В блоке доступны свойства и методы объекта:

```
data class Person(var name: String, var age: Int = 0, var city: String = "")
fun main() {
    val p = Person("Виктор").apply {
        age = 32
        city = "Москва"
    }
    println(p) → Person(name=Виктор, age=32, city=Москва)
}
```

Действия функции **apply** можно трактовать так: выполнить следующее для заданного объекта. Обращаться к свойствам можно через **this**:

```
this.age = 32
```

### Also

Функция **also** позволяет вставить в цепочку дополнительные действия, которые никак не влияют на дальнейшее:

```
fun main() {
    val m = mutableListOf("one", "two", "three")
    m.also { println(it) }.add("four")
}
```

Функция **also** вызывается для объекта, который доступен в блоке через **it**.

Все эти пять функций в общем-то взаимозаменяемые. В документации есть рекомендации по выбору подходящей функции. Указано также, что не стоит этими функциями особенно увлекаться, так как они могут ухудшать некоторые характеристики программы.

Имеется ещё две полезные функции подобного рода: **takeIf** и **takeUnless**. Этим функциям придаётся блок, возвращающий результат

типа **Boolean**. Если блок даёт **true**, то функция **takeIf** возвращает объект, для которого она вызвана, а если **false**, то возвращается **null**. Функция **takeUnless** обратная **takeIf**. Вот пример:

```
import kotlin.random.*
fun main() {
        val x = Random.nextInt(100)
        val a = x.takeIf { it % 2 == 0 }
        val b = x.takeUnless { it % 2 == 0 }
        println("even: $a, odd: $b") → even: null, odd: 75
}
```

Если с результатом, возвращаемым этими функциями, будут выполняться дальнейшие действия, то надо не забывать ставить знак **?**, поскольку этот результат **nullable**:

```
fun main() {
        val s = "Hello"
        val x = s.takeIf { it.isNotEmpty() }?.uppercase()
        println(x) → HELLO
}

fun main() {
        val s = ""
        val x = s.takeIf { it.isNotEmpty() }?.uppercase()
        println(x) →null
}
```

После этих функций могут вызываться scope function, образуя обычную цепочку. Например, это может быть функция **let**:

```
fun main() {
        fun f(x: String, y: String) {
                x.indexOf(y).takeIf { it >= 0 }?.let {
                        println("The substring $y is found in $x.")
                        println("Its start position is $it.")
                }
        }
        f("010000011", "11")
        f("010000011", "12")
}
```

В результате получим:

*The substring 11 is found in 010000011.*

*Its start position is 7.*

Когда **takeIf** возвращает **null**, функция **let** не вызывается.

### Композиция функций

Kotlin позволяет создавать композиции функций с использованием специального синтаксиса. Рассмотрим пример композиции двух функций. Пусть функции **f** передаётся функция **g** в качестве аргумента. Тогда можно создать композицию в виде функции **fi**, задав её сигнатуру:

```
fun <A, B, C> fi(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

Применение параметров типа позволяет создавать весьма универсальные композиции, способные работать с переменными разных типов. В целом это может выглядеть, например, так:

```
fun <A, B, C> fi(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
fun f1(x: Int) = x % 2 != 0
fun f2(s: String) = s.length
fun main() {
    val f3 = fi(::f1, ::f2)
    val m = listOf("abc", "ab", "a")
    println(m.filter(f3)) → [abc, a]
}
```

Здесь функция **f1** возвращает **true** для нечётных чисел и **false** – для чётных, а функция **f2** определяет число букв в слове. Композиция **fi** позволяет создать функцию **f3**, которой передаются функции **f1** и **f2** с использованием *method reference* (::). Итератор **filter** выбирает слова с нечётным количеством букв. Конечно, явно функцию **f3** можно было не создавать:

```
fun main() {
    val m = listOf("abc", "ab", "a")
    println(m.filter(fi(::f1, ::f2)))
}
```

# 12.Параллелизм, асинхронность и корутины (coroutine)

Параллельные вычисления позволяют выполнять несколько задач одновременно, а асинхронность даёт возможность не блокировать основной ход приложения. Это особенно важно, если параллельные вычисления занимают продолжительное время. Например, мы создаем графическое приложение и нам надо по нажатию на кнопку отправлять запрос к интернет-ресурсу. Этот запрос может занять продолжительное время. И чтобы приложение не зависало на период отправки запроса, его следует отправлять асинхронно. При асинхронных запросах пользователь не ждет пока придет ответ от интернет-ресурса, а продолжает работу с приложением, а при получении ответа получит соответствующее уведомление.

В стандартной библиотеке Kotlin отсутствуют средства для параллелизма и асинхронности, но эти возможности реализованы в виде так называемых корутин (*coroutine* – сопрограмма). По сути корутины это блоки кода, которые могут выполняться параллельно с основной программой, а базовая функциональность, связанная с корутинами, сосредоточена в библиотеке *kotlinx.coroutines*. Эту библиотеку, разработанную JetBrains необходимо подключить дополнительно.

Для подключения надо в разрабатываемом на Kotlin проекте добавить зависимости (*dependency*) так, как это описано в документации. Имеется несколько способов, но проще всего это сделать, если при разработке проекта используется редактор Intellij IDEA. После создания проекта надо выполнить следующие шаги:

1. В меню *File* выбрать *ProectStructure*
2. На вкладке *Proect Setting* перейти к *Libraries* и тогда в центральном поле редактора появится список уже добавленных в проект библиотек.
3. Нажать знак + в правом верхнем углу основного поля  и в контекстном меню выбрать *From Maven* (эта система позволяет управлять использованием стандартных библиотек).
4. В открывшемся окне в поле ввода вести название нужной нам

библиотеки *kotlinx-coroutines-core-jvm* и нажать кнопку поиска. Если библиотека будет найдена, отобразится выпадающий список с результатами.

5. Выбрать из списка нужную версию (у меня это 1.7.2, но в конкретном случае номер может быть другим). Потом надо поставить нужные флажки и нажать *OK*. После этого библиотеку можно будет найти в списке установленных библиотек.

Как увидим далее, корутины используют некие объекты под названиями **launch**, **runBlocking**, **coroutineScope**. Все они в документации называются одним словом **builder** (строитель, конструктор, компоновщик). Для краткости я буду дальше их называть словом оператор, хотя по существу это не совсем верно.

Теперь можно попробовать первую корутину. В созданном ранее на редакторе Intellij IDEA проекте заменим шаблонный код в файле **Main.kt** на следующий (Редактор Intellij IDEA позволяет создавать новые файлы с кодом на Kotlin, но я советую для каждого примера просто менять код в одном и том же файле **Main.kt**):

```
import kotlinx.coroutines.*
fun main() = runBlocking {
   launch { delay(1000L); println("Мир") }
   println("Привет")
}
```

Запустив программу, на терминале получим:

*Привет*
*Мир*

Оператор **launch** активирует корутину и выполняет приданный ему блок параллельно с основной программой, представленной здесь функцией **main()**. Сам этот оператор должен находится в блоке, приданном оператору **runBlocking**, отсутствие которого вызовет ошибку компиляции. В общем случае в этом блоке может быть несколько корутин. Функция **delay** вызывает остановку программы на время в миллисекундах, заданное целым числом типа **Long**. Пока выполнение блока оператора **launch** приостановлено, основная программа успевает выполнить строку **println("Привет")**. В результате текст *Мир!* выводится на терминал после текста *Привет*.

Чтобы убедиться, что параллельные потоки выполняются независимо друг от друга, вставим задержку времени также и в основной поток:

```
import kotlinx.coroutines.*
fun main() = runBlocking {
   launch { delay(1000L); println("Мир") }
   delay(2000L); println("Привет")
}
```

Поскольку теперь задержка в основном потоке больше (2000мск), параллельный поток будет выполнен раньше и в результате получим:
*Мир*
*Привет*

Модификатор **suspend** позволяет создавать так называемые suspend-функции, которые могут быть вызваны в блоке оператора **runBlocking**:

```
import kotlinx.coroutines.*
fun main() = runBlocking {
   launch { f() }; println("Hello")
}
suspend fun f() {
   delay(1000L); println("World")
}
```

Suspend-функции могут также быть вызваны в теле других suspend-функций. Оператор **launch** со своим блоком тоже можно считать suspend-функций.

Оператор **coroutineScope** позволяет создавать suspend-функци, содержащие в своём теле два или более конкурирующих потока:

```
import kotlinx.coroutines.*
fun main() = runBlocking { f() }
suspend fun f() = coroutineScope {
   launch { delay(1000L); print("Наташа") }
   println("Здравствуй, ")
}
```

Получим: *Здравствуй, Наташа*.

Теперь создадим suspend-функцию с тремя конкурирующими потоками:

```
Import kotlinx.coroutines.*
fun main() = runBlocking { f(); println("Конец") }
suspend fun f() = coroutineScope {
   launch { delay(2000L); println("Маша") }
```

```
    launch { delay(1000L); println("Катя") }
    println("Привет")
}
```

Получим:

*Привет*
*Катя*
*Маша*
*Конец*

Оператор **launch** создаёт объект (на самом деле функцию), которым можно инициировать переменную, а метод **join()** позволяет этот объект вызвать в нужном месте:

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    val x = launch { delay(1000L); println("год") }
    println("Новый")
    x.join()
    println("Конец")
}
```

Получим:

*Новый*
*год*
*Конец*

Метод **cancel()** позволяет прервать параллельный поток. Посмотрим на примере:

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    val x = launch {
        repeat(1000) { i -> println("x: Я сплю $i ..."); delay(500L) }
    }
    delay(1300L); println("main: Я устал ждать!")
    x.cancel(); x.join()
    println("main: Теперь я могу уйти.")
}
```

Получим:

*x: Я сплю 0 ...*
*x: Я сплю 1 ...*
*x: Я сплю 2 ...*
*main: Я устал ждать!*

*main: Теперь я могу уйти.*

Пока основной поток простаивает 1300 мсек **x.join()** запускает поток **x** в котором печатается «*Я сплю...»;* при этом в каждом цикле поток **x** задерживается на 500мсек. Когда 1300мсек проходят, выводится "*Я устал ждать!*" и метод **cancel()** прерывает поток **x**.

Отметим попутно, что оператор **repeat** в цикле увеличивает на **1** аргумент лямбда-выражения в приданом оператору блоке.

Функция **withTimeoutOrNull()** позволяет без использования **cancel()** прерывать поток если его время превышает заданное:

```
import kotlinx.coroutines.*
fun main() = runBlocking {
   val x = withTimeoutOrNull(1300L) {
      repeat(1000) { i -> println("Я сплю $i ..."); delay(500L) }
   }
   println("Результат: $x")
}
```

*Я сплю 0 ...*
*Я сплю 1 ...*
*Я сплю 2 ...*
*Результат: null*

Функция возвращает **null**, если всё нормально.

Функция **measureTimeMillis** позволяет определять суммарное время работы потоков, запущенных в теле этой функции:

```
import kotlinx.coroutines.*
import kotlin.system.*
fun main() = runBlocking<Unit> {
   val t = measureTimeMillis {
      val x = f1()
      val y = f2()
      println("Результат: ${x + y}")
   }
   println("Всего затрачено времени: $t ms")
}
suspend fun f1(): Int {
   delay(1000L) // Делаем тут что-нибудь полезное
   return 13
}
suspend fun f2(): Int {
```

```
    delay(1000L) //Делаем тут что-нибудь полезное
    return 29
}
```

*Результат: 42*
*Всего затрачено времени: 2056 ms*

В предыдущем примере потоки выполнялись последовательно. Для того, чтобы они работали параллельно, можно использовать оператор **async** и функцию **await()**:

```
import kotlinx.coroutines.*
import kotlin.system.*
fun main() = runBlocking<Unit> {
    val t = measureTimeMillis {
        val x = async { f1() }
        val y = async { f2() }
        println("Результат ${x.await() + y.await()}")
    }
    println("Выполнено за $t ms")
}
suspend fun f1(): Int {
    delay(1000L)
    return 13
}
suspend fun f2(): Int {
    delay(1000L)
    return 29
}
```

*Результат 42*
*Выполнено за 1322 ms*

Как видим, здесь мы имеем экономию времени за счёт параллельности.

# Контроль и конвертирование типов

## Операторы is и !is

Операторы **is** и **!is** позволяют проверить, принадлежит ли переменная заданному типу:

```
fun main() {
        val x: Any = "Ленинград"; val y: Any = 123.04
        if (x is String) { println(x.length) } → 9
        if (y !is String) { print("$y не строка") } else { print(y.length) }
                                        → 123.04 не строка
}
```

Здесь нельзя задавать конкретный тип:
**val x: String = "Ленинград"** и **val y: Double = 123.04** это приведёт к ошибке «несовместимый тип» при выполнении операторов **is** и **!is**.

Ранее мы уже видели, что операторы **is** и **!is** можно применять и в условном операторе **when – else**:

```
fun main() {
        fun f(x: Any) {
                when (x) {
                        is Int -> print(x + 1)
                        is String -> print(x.length + 1)
                        is IntArray -> print(x.sum())
                }
        }
        val m = intArrayOf(1,2,3,4,5)
        f(m) → 15
}
```

Здесь использована стандартная функция **sum()**, вычисляющая сумму элементов массива.

## Конвертирование типов

Часто нет необходимости конвертировать тип явно, компилятор может делать это автоматически - «разумное конвертирование » (Smart casts ). Приведём примеры:

```
fun f(x: Any) { if (x is String) { print(x.length) } }
fun main() {
    f("Ленинград") → 9
    f(123) → ничего не выводится
}
```

Здесь компилятор сам приводит тип для **x** к **String**. Аналогичное поведение будет и в этом случае:

```
if (x !is String) return
print(x.length)
```

В следующих двух примерах переменная **x** приводится к **String** в правой стороне от логических операторов || или **&&**:

```
if (x !is String || x.length == 0) return
if (x is String && x.length > 0) { print(x.length) }
```

Разумное конвертирование работает в выражениях **when** и в циклах **while**:

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

Иногда автоматическое конвертирование невозможно, а при попытке сделать это получаем исключение. В документации это называют «опасное конвертирование» (Unsafe cast). Например, такая ситуация может быть при использовании оператора **as**:

```
val x: String = y as String
```

Здесь получим исключение, если переменная **y** будет иметь значение **null**. Чтобы предотвратить это, надо применить тип optional (nullable):

```
val x: String? = y as String?
```

Кроме того есть также оператор **as?**, который делает то же самое:

**val x: String? = y as? String**

}