Основы языка Kotlin

Будем пользоваться нотацией REPL: после `=>` описывается тип результата, затем строковое представление результата. Если код что-то выводит, то это написано после результата.

Итак, Котлин. Давайте для начала разберёмся с местными примитивными конструкциями. У нас есть следующие типы данных:

- Логические значения

```
true
```

```
  => : Boolean = true
```

- Целые числа (32-битные в two's complement)

```
566
```

```
  => : Int = 566
```

- Длинные целые числа (64-битные)

```
566L
```

```
  => : Long = 566
```

- Строки

```
"Hello"
```

```
  => : String = "Hello"
```

- Символы

```
'a'
```

```
  => : Char = 'a'
```

- Вещественные числа (Стандарт `IEEE-754`, double precision floating point number)

```
3.14
```

```
  => : Double = 3.14
```

Все эти типы заимствованы из JVM. Собственно, из неё же заимствованы ещё и типы `Byte` (8-битное целое), `Short` (16-битное целое) и `Float` (single-precision вещественное), но их используют крайне редко (поскольку всё меньше остаётся даже 32-битных процессоров).

Зато, для желающих, есть *довольно быстрая* реализация `unsigned` целых чисел: `UInt`, `ULong` etc.

С этими типами можно делать довольно ожидаемые действия:

- Складывать:

```
0.1 + 0.2
```

```
  => : Double = 0.30000000000000004
```

- Умножать:

```
5 * -2
```

```
  => : Int = -10
```

- Делить:

```
1.0 / 0.0
```

```
  => : Double = Infinity
```

Без неожиданных спецэффектов. Автоприведения типов здесь нет, так что `"22" - "2" = 20`, как в JS, не получится. С символами, впрочем, операции производить можно:

```
'c' - 'a'
```

```
=> : Int = 2
```

Также в наличии логические операции

```
!false
```

```
=> : Boolean = true
```

```
true && false
```

```
=> : Boolean = false
```

Ну и так далее. Разберётесь по ходу ведения.

Ветвление:

```
if (1 + 2 < 4) {
    println("First")
} else {
    println("Second")
}
```

```
=> : Unit = Unit
```

```
First
```

О, здесь сразу несколько интересных вещей: `println` отвечает за вывод строки в `stdout` (стандартный поток вывода). А `Unit` — это тип, который возвращается, если не возвращается ничего. Есть в Котлине два таких интересных типа: `Unit` — когда единственное, что нам нужно знать — это завершилась функция или нет. Если завершилась, то возвращается `Unit` — единственное значение типа `Unit`. Ну, содержательного с ним ничего нельзя сделать — только сказать, что он есть. Второй — `Nothing` — означает, что функция никогда ничего не вернёт, потому что не завершится успешно. Если у предыдущего типа существует единственное значение, то у типа `Nothing` нет ни одного значения.

Так, погодите. Выходит, у нас `if...else` что-то возвращает? А можно, чтобы он возвращал содержательную информацию? Можно.

```
if (1 + 2 < 4) {
    // Do something
    566
} else {
    // Do some another thing
    239
}
```

```
=> : Int = 566
```

Последняя строчка интерпретируется как выражение — то есть, что-то-таки возвращает наружу. Собственно, привычный многим в Java и в C++ тернарный оператор `a ? b : c` здесь так и сделан: если выражение однострочное, то фигурные скобки можно и не ставить.

```
if (1 + 2 < 4) 566 else 239
```

```
=> : Int = 566
```

Теперь циклы. А для циклов сначала нужны переменные. Их есть:

```
var i = 0
while (i < 5) {
    print(i)
    i++
}
```

```
01234
```

Во-первых, `print` — это как `println`, только без перевода строки. Во-вторых, погодите, а результат? Мы думали, и в магазине можно стеночку приподнять?

А нет. Цикл не является выражением, он не возвращает даже `Unit`. Как, впрочем, и объявление переменной.

В-третьих, переменные в Котлине имеют тип. То есть, если у вас про переменную заявлено, что там лежат числа, вы не можете положить туда строку:

```
var a : Int = 0
a = 1
a = "2"
```

```
Kotlin: Type mismatch: inferred type is String but Int was expected
```

А если вы не заявляли тип явно, то компилятор сам выведет наиболее узкий, который сможет.

Кроме переменных, конечно, есть постоянные:

```
val a : Int = 0
a = 1
```

```
Kotlin: Val cannot be reassigned
```

Если вы можете использовать `val`, используйте `val`, а не `var`.

Есть также циклы `do-while`:

```
do {
    val line = readln()
    // Do something with it
} while (line != "exit")
```

Переменная, объявленная внутри цикла, в общем случае снаружи не видна, а вот в случае конкретно `do-while` видна в условии, что приятно. Кажется, так было не всегда.

… и циклы for:

```
for (i in 0 until 5) {
    print(i)
}
```

```
01234
```

until — это специальная инфиксная функция, которая возвращает арифметическую прогрессию, от первого аргумента включительно, до второго исключительно. Что означает “инфиксная”? Ну, функции в котлине бывают трёх различных вариантов: префиксная (`cos(1.0)`), условно-постфиксная (`566.toString()`) и инфиксная (`0 until 5`). *Условно*-постфиксные они потому, что в скобках могут быть ещё аргументы, например, `566.toString(3)` , чтобы преобразовать в троичную систему счисления. Все инфиксные функции можно вызывать также и в условно-постфиксной записи: `0.until(5)` , но не наоборот.

Помимо until есть ещё downTo и rangeTo:

until	`0 until 5`	`0.until(5)`	`[0, 1, 2, 3, 4]`
rangeTo	`0..5`	`0.rangeTo(5)`	`[0, 1, 2, 3, 4, 5]`
downTo	`5 downTo 0`	`5.downTo(0)`	`[5, 4, 3, 2, 1, 0]`

В экспериментальных на момент написания конспекта версиях Котлина есть ещё вариант `0..<5` для until (Точнее, строго говоря, будет вызван метод `rangeUntil`, но делает он то же самое).

Как это работает? Э-хе-хе, подождите, давайте сначала научимся префиксные функции писать.

```
fun max(a: Int, b: Int): Int
```

Что бы всё это значило? `fun` собственно объявляет функцию. `a` и `b` — это имена параметров, после них написаны их типы (`Int`), а после двоеточия после всего объявления тип того, что возвращает функция. В нашем случае функция принимает два аргумента типа `Int` и возвращает тоже `Int`. Теперь собственно, что делает эта функция?

```
fun max(a: Int, b: Int): Int {
    if (a < b) {
        return b
    } else {
        return a
    }
}
```

Ну, здесь всё понятно.

Проверим, что работает:

```
max(239, 566)
```

```
=> : Int = 566
```

Ещё не забыли, что `if` у нас также является тернарным оператором?

```
fun max(a: Int, b: Int): Int {
    return if (a < b) b else a
}
```

Если тело функции состоит только из `return`-statement, то фигурные скобки и `return` не нужны, вместо них пишем

```
fun max(a: Int, b: Int): Int = if (a < b) b else a
```

Здесь уже можно опустить возвращаемый тип, ибо он очевиден компилятору, но лучше всё же писать.

Так, кроме того, мы ввели ключевое слово `return`. Кроме него существуют ещё такие интересные штуки как `break`, `continue` и `throw`, которые делают понятно что. Но их объединяет следующая черта: после них точно будет выполняться уже не эта строка. В некотором смысле, это выражение никогда ничего не возвращает. О, где-то такое уже было. На самом деле на уровне языка оно возвращает тип `Nothing`. Например, если мы напишем `val x = return` , то понятно, что до присвоения значения постоянной `x` дело так и не дойдёт.

Для понимания следующего сначала рассмотрим обобщающие типы. Так, если мы пишем `if (condition) 1 else 2` , то понятно, какой тип выведет компилятор — `Int`.

А если типы возвращаемых данных разные? Ошибка?

```
if (1 < 2) 3 else "4"
```

```
=> : Comparable<*> = 3
```

А нет. Что-то мы всё же про это знаем, и числа, и строки можно сравнивать. А значит, мы на выходе получили что-то *сравниваемое* (`Comparable<*>`). Что означает звёздочка в треугольных скобках — мы потом разберём, сейчас это несущественно. А бывают типы данных, которые нельзя сравнивать? Бывают, например `Unit`. Ну, его сравнивать довольно бесполезно — он всегда окажется равен самому себе, поскольку единственное, что существует типа `Unit` — это `Unit` сам по себе.

```
if (1 < 2) 3 else Unit
```

```
=> : Any = 3
```

Вот `Any` — это почти самый общий тип. Он означает “ну, что-то там определённо лежит”. С ним можно проделать не так много вещей:

- потребовать строковое представление
- посчитать хэшкод
- сравнить на равенство с чем-то другим.

В общем-то, всё.

Есть ещё значение `null`, которое не является `Any`, а о его типе поговорим чуть позже.

Так вот, всё, что не `null` — является `Any`, и может быть приведён к типу `Any`. А тип `Nothing` — наоборот, может быть приведён *к* любому типу. Потому что на самом деле никогда не понадобится приводить, потому что объектов типа `Nothing` не существует.

Значение `null` в Котлине несёт тот же смысл, что `null` в Java, nullptr в C++, None в Python и т.д. Это заглушка, возможность сказать, что здесь могло бы быть значение, но пока его нет. Но хранить их можно только в специальных nullable типах, помечаемых знаком вопроса.

```
var s : String = "abc"
s = null
```

```
Null can not be a value of a non-null type String
```

```
var s : String? = "abc"
s = null // OK
```

В отличие от Java, nullable типами могут быть и примитивные (`Int`, `Long` etc.).

> Впрочем, это довольно неэффективно, так как, в то время как `Int`, `Char`, и так далее транслируются в использование примитивов (int, char), nullable типы `Int`?, `Char`? — в объектные Integer, Character и так далее, чтобы можно было присваивать им `null`. Это занимает больше места, больше времени и прочее. Аналогично происходит с `lateinit var`, которые под капотом nullable.

Поскольку NullPointerException — одна из самых частых ошибок времени исполнения в Java, в Kotlin добавили специальных операторов для работы с nullable значениями:

- Non-null assertion

> Если значение не `null`, то вернуть его, если `null` — бросить ошибку.
>
> ```
> val s : String? = "abacaba"
> s!!
> ```
>
> ```
> => : String = "abacaba"
> ```
>
> ```
> val s : String? = null
> s!!
> ```
>
> ```
> Exception in thread "main" java.lang.NullPointerException
>     at test.TestKts.main(Test.kts:2)
> ```

- Safe call

> Если значение не `null`, то вызвать метод, если `null` — вернуть `null`.
>
> ```
> val s : String? = "abacaba"
> s?.substring(2, 6)
> ```
>
> ```
> => : String? =  "acab"
> ```
>
> ```
> val s : String? = null
> s?.substring(2, 6)
> ```
>
> ```
> => : String? =  null
> ```

- Elvis operator

> Если значение не `null`, то вернуть его, если `null` — то вернуть другое.
>
> ```
> val s : String? = "abacaba"
> s ?: "by default"
> ```
>
> ```
> => : String? =  "abacaba"
> ```
>
> ```
> val s : String? = null
> s ?: "by default"
> ```
>
> ```
> => : String = "by default"
> ```

Какой тип имеет сам `null`?

```
null
```

```
=> : Nothing? =  null
```

Почему так? Логично, что если тип `T` приводится к типу `U`, то тип `T?` должен приводиться к типу `U?` (Если там было значение типа `T`, то его можно положить в `U`, а значит, в `U?` тем более. А если там был `null`, то его всё равно можно положить в `U?`). И логично, чтобы `null` приводился к любому nullable типу. `Nothing` приводится к любому `T`, значит, `Nothing?` приводится к любому `T?`.

Осталось поговорить про массивы. А их здесь несколько разных бывает.

```
arrayOf("abc", "def", "ghi")
```

```
=> : Array<String> = [Ljava.lang.String;@3b9a45b3
```

Будьте здоровы, Вы, кажется, чихнули.

Да, нормального встроенного строкового представления для массивов не подвезли, выводится некая системная информация. Для этого придётся вызвать специальную функцию `joinToString` (на самом деле она гораздо мощнее, но об этом мы поговорим позже).

```
arrayOf("abc", "def", "ghi").joinToString()
```

```
=> : String = "abc, def, ghi"
```

Тем не менее, с массивами можно работать, как в любом другом языке.

```
val arr = arrayOf("abc", "def", "ghi")
arr[1]
```

```
=> : String = "def"
```

(Да, индексация с нуля)

```
arr[0] = "magic"
arr.joinToString()
```

```
=> : String = "magic, def, ghi"
```

Массивы типизированы: за это, собственно, отвечает `String` в треугольных скобках после `Array<String>` (как это работает, опять-таки — позже, сейчас пользуемся как магией). То есть, нельзя просто так взять и положить `Int` в массив строк:

```
arr[0] = 1
```

```
The integer literal does not conform to the expected type String
```

Также, если попытаться обратиться за границу массива, получаем ошибку:

```
arr[3]
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of
bounds for length 3
    at TestsKts.main(Tests.kts:6)
```

Можно создавать и массивы более общих типов. Например,

```
arrayOf(1, "2", null)
```

```
=> : Array<Comparable<*>?> = [Ljava.lang.Object;@3b9a45b3
```

Как мы помним, у `Int` и `String` есть нечто общее — они оба `Comparable<*>`. А у `Comparable<*>` и `null` общим типом будет, очевидно, `Comparable<*>?`.

Можно также задать тип вручную:

```
arrayOf<Number>(1, 2, 3)
```

```
=> : Array<Number> = [Ljava.lang.Number;@3b9a45b3
```

`Number` — это обобщение `Int`, `Long`, `Double` и так далее. В `Array<Number>` мы можем положить любые числа. И когда мы что-то достаем из массива `Number`, мы не можем точно быть уверены, какое конкретно это число.

Мы можем выполнять общие операции, определённые для всех чисел, например, привести к типу `Double`:

```
val arr = arrayOf<Number>(1, 2, 3)
arr[1].toDouble()
```

```
=> : Double = 2.0
```

Или проверить, какого оно типа:

```
arr[1] is Long
```

```
=> : Boolean = false
```

```
arr[1] is Int
```

```
=> : Boolean = true
```

После этого, если мы уверены, привести явным образом

```
arr[1] as Int
```

```
=> : Int = 2
```

Стоит заметить, во-первых, что попытка каста к не тому типу обернётся ошибкой времени исполнения:

```
arr[1] as Double
```

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer
cannot be cast to class java.lang.Double (java.lang.Integer and java.lang.Double are
in module java.base of loader 'bootstrap')
    at TestsKts.main(Tests.kts:6)
```

А попытка проверить на `is` что-то совершенно точно бессмысленное — ошибкой компиляции:

```
arr[1] is String
```

```
Incompatible types: String and Number
```

...Так как компилятор считает правильным сообщить вам, что `Number` совершенно точно никогда строкой не окажется.

Так, давайте вернёмся к массивам. Можно ли создавать массивы массивов? Сколько угодно:

```
arrayOf(arrayOf("abc", "def"), arrayOf("ghi", "jkl"))
```

```
=> : Array<Array<String>> = [[Ljava.lang.String;@3b9a45b3
```

Будьте здоровы.

```
arrayOf(arrayOf("abc", "def"), arrayOf("ghi", "jkl")).joinToString()
```

```
=> : String = "[Ljava.lang.String;@6d03e736, [Ljava.lang.String;@568db2f2"
```

Будьте здоровы!

```
val matrix = arrayOf(arrayOf("abc", "def"), arrayOf("ghi", "jkl"))
matrix.joinToString(transform = Array<String>::joinToString)
```

```
=> : String = "abc, def, ghi, jkl"
```

Это ещё куда ни шло, но совершенно непонятно, где кончаются границы одного и другого массива внутри.

```
val matrix = arrayOf(arrayOf("abc", "def"), arrayOf("ghi", "jkl"))
matrix.joinToString(", ", "[", "]") { it.joinToString (", ", "[", "]") }
```

```
=> : String = "[[abc, def], [ghi, jkl]]"
```

Есть, конечно, возможность заставить это работать. Но… вам не надоело?

Есть несколько причин, почему в Котлине непосредственно массивы используются крайне редко. Во-первых, вышеупомянутые проблемы со строковым представлением. Во-вторых, и это более важно, проблемы с безопасностью. Допустим, вы передали куда-то в функцию массив.

```
fun blackBox(data: Array<String>) {
    data[2] = "flowers"
}

val array = arrayOf("Some", "important", "data", "here")
blackBox(array)
array.joinToString()
```

```
=> : String = "Some, important, flowers, here"
```

Полундра! Данные скомпрометированы!

Проблема в том, что массивы передаются исключительно по ссылке, копирования при передаче не происходит. А значит, отдавая кому-то на сторону данные, вы не можете быть уверены в их сохранности — фактически вы отдаёте ссылку на область в памяти, где эти данные лежат.

Чтобы этого избежать, существуют листы:

```
listOf("Some", "important", "data", "here")
```

```
=> : List<String> = [Some, important, data, here]
```

И строковое представление нормальное, и записать туда что-то нам не дадут:

```
val arr = listOf("Some", "important", "data", "here")
arr[2] = "flowers"
```

```
Unresolved reference. None of the following candidates is applicable because of
receiver type mismatch:
    public inline operator fun kotlin.text.StringBuilder /* = java.lang.StringBuilder
*/.set(index: Int, value: Char): Unit defined in kotlin.text
No set method providing array access
```

… очень много информации. Компилятор попытался найти метод с именем `set`, который отвечал бы за подобное присваивание, и не справился. Содержательная часть здесь `No set method providing array access` — записывать сюда нам не дадут.

Листы и массивы можно превращать друг в друга:

```
val arr = arrayOf("Some", "important", "data", "here")
arr.toList()
```

```
=> : List<String> = [Some, important, data, here]
```

```
val arr = listOf("Some", "important", "data", "here")
arr.toTypedArray()
```

```
=> : Array<String> = [Ljava.lang.String;@4fca772d
```

И в том, и в другом случае действительно происходит копирование.

Кстати, говорилось про три варианта? Да, есть ещё `MutableList`.

```
mutableListOf(1, 2, 3)
```

```
=> : MutableList<Int> = [1, 2, 3]
```

В него, как и в массив, можно записывать, из него можно читать, но, что особенно важно, можно его расширять (массивы имеют постоянную длину!):

```
val arr = mutableListOf(1, 2, 3)
arr.add(566)
arr
```

```
=> : MutableList<Int> = [1, 2, 3, 566]
```

Так вот, `MutableList<T>` в частности, является `List<T>` для любого типа `T`, так как `MutableList<T>` предоставляет те же методы: чтение, проверка длины, некоторые другие; но предоставляет и дополнительные — изменение, расширение. Так что можно передать `MutableList<T>` в функцию, которая требует `List<T>` и (почти) не бояться, что его изменят.

> Почти — потому что некоторая возможность всё же есть; если вдруг это данные для запуска ядерных ракет и вы передаёте их в функцию — лучше всё-таки сделайте копию.

> Ну, было бы желание сломать — сломать получится. Например, до Java версии 1.8 включительно можно было провернуть очень интересный фокус, следите за руками:
>
> ```
> 1 as Any
> ```
>
> ```
> => : Any = 1
> ```
>
> Всё, казалось бы, логично, от того, что мы привели к более общему типу, значение-то не поменялось. Да?

```
val rnd = Random(56630239)
val clazz = Class.forName("java.lang.Integer\$IntegerCache")
val field = clazz.getDeclaredField("cache")
field.isAccessible = true
val cache = field.get(null) as Array<Int>
for (i in 0 until cache.size) cache[i] = rnd.nextInt(cache.size)
```

А вот после этого замечательного кода попробуем снова

```
1 as Any
```

```
=> : Any = 146
```

Э-э-э... Упс?