# Числа

## Целочисленные типы

В Kotlin есть набор встроенных типов, которые представляют числа. Для целых чисел существует четыре типа с разными размерами и, следовательно, разными диапазонами значений.

| Тип | Размер (биты) | Минимальное значение | Максимальное значение |
|---|---|---|---|
| `Byte` | 8 | -128 | 127 |
| `Short` | 16 | -32768 | 32767 |
| `Int` | 32 | -2,147,483,648 ($-2^{31}$) | 2,147,483,647 ($2^{31} - 1$) |
| `Long` | 64 | -9,223,372,036,854,775,808 ($-2^{63}$) | 9,223,372,036,854,775,807 ($2^{63} - 1$) |

Все переменные, инициализированные целыми значениями, не превышающими максимальное значение `Int`, имеют предполагаемый тип `Int`. Если начальное значение превышает это значение, то тип `Long`. Чтобы явно указать тип `Long`, добавьте после значения `L`.

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
```

## Типы с плавающей точкой

Для действительных чисел в Kotlin есть типы с плавающей точкой `Float` и `Double`. Согласно стандарту IEEE 754 (https://en.wikipedia.org/wiki/IEEE_754), типы с плавающей точкой различаются своим *десятичным разрядом*, то есть количеством десятичных цифр, которые они могут хранить. С точки зрения IEEE 754 `Float` является одинарно точным, а `Double` обеспечивает двойную точность.

| Тип | Размер (биты) | Значимые биты | Биты экспоненты | Разряды |
|---|---|---|---|---|
| `Float` | 32 | 24 | 8 | 6-7 |
| `Double` | 64 | 53 | 11 | 15-16 |

Вы можете инициализировать переменные `Double` и `Float` числами, имеющими дробную часть. Она должна быть отделена от целой части точкой ( `.` ). Для переменных, инициализированных дробными числами, компилятор автоматически определяет тип `Double` .

```
val pi = 3.14 // Double
// val one: Double = 1 // Ошибка: несоответствие типов
val oneDouble = 1.0 // Double
```

Чтобы явно указать тип `Float` , добавьте после значения `f` или `F` . Если такое значение содержит более 6-7 разрядов, оно будет округлено.

```
val e = 2.7182818284 // Double
val eFloat = 2.7182818284f // Float, фактическое значение 2.7182817
```

Обратите внимание, что в отличие от некоторых других языков, в Kotlin нет неявных преобразований для чисел. Например, функция с `Double` параметром может вызываться только для `Double` , но не для `Float` , `Int` или других числовых значений.

```
fun main() {
    fun printDouble(d: Double) { print(d) }

    val i = 1
    val d = 1.0
    val f = 1.0f

    printDouble(d)
//  printDouble(i) // Ошибка: несоответствие типов
//  printDouble(f) // Ошибка: несоответствие типов
}
```

Чтобы преобразовать числовые значения в различные типы, используйте Явные преобразования.

## Символьные постоянные

В языке Kotlin присутствуют следующие виды символьных постоянных (констант) для целых значений:

- Десятичные числа: `123`
    - Тип `Long` обозначается заглавной `L` : `123L`
- Шестнадцатеричные числа: `0x0F`

- Двоичные числа: `0b00001011`

> ВНИМАНИЕ: Восьмеричные литералы не поддерживаются.

Также Kotlin поддерживает числа с плавающей запятой:

- Тип `Double` по умолчанию: `123.5`, `123.5e10`
- Тип `Float` обозначается с помощью `f` или `F`: `123.5f`

Вы можете использовать нижние подчеркивания, чтобы сделать числовые константы более читаемыми:

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

# Представление чисел в JVM

Обычно платформа JVM хранит числа в виде примитивных типов: `int`, `double` и так далее. Если же вам необходима ссылка, которая может принимать значение null (например, `Int?`), то используйте обёртки. В этих случаях числа помещаются в Java классы как `Integer`, `Double` и так далее.

Обратите внимание, что использование обёрток для одного и того же числа не гарантирует равенства ссылок на них.

```
val a: Int = 100
val boxedA: Int? = a
val anotherBoxedA: Int? = a

val b: Int = 10000
val boxedB: Int? = b
val anotherBoxedB: Int? = b

println(boxedA === anotherBoxedA) // true
println(boxedB === anotherBoxedB) // false
```

Все nullable-ссылки на `a` на самом деле являются одним и тем же объектом из-за оптимизации памяти, которую JVM применяет к `Integer` между "-128" и "127". Но `b` больше этих значений,

поэтому ссылки на `b` являются разными объектами.

Однако, равенство по значению сохраняется.

```
val b: Int = 10000
println(b == b) // Prints 'true'
val boxedB: Int? = b
val anotherBoxedB: Int? = b
println(boxedB == anotherBoxedB) // Prints 'true'
```

## Явные преобразования

Из-за разницы в представлениях меньшие типы *не являются подтипами* бОльших типов. В противном случае возникли бы сложности.

```
// Возможный код, который на самом деле не скомпилируется:
val a: Int? = 1 // "Обёрнутый" Int (java.lang.Integer)
val b: Long? = a // неявное преобразование возвращает "обёрнутый" Long (java.lang.Lon
g)
print(b == a) // Нежданчик! Данное выражение выведет "false" т. к. метод equals() тип
а Long предполагает, что вторая часть выражения также имеет тип Long
```

Таким образом, будет утрачена не только тождественность (равенство по ссылке), но и равенство по значению.

Как следствие, неявное преобразование меньших типов в большие НЕ происходит. Это значит, что мы не можем присвоить значение типа `Byte` переменной типа `Int` без явного преобразования.

```
val b: Byte = 1 // всё хорошо, литералы проверяются статически
// val i: Int = b // ОШИБКА
val i1: Int = b.toInt()
```

Каждый численный тип поддерживает следующие преобразования:

- `toByte(): Byte`
- `toShort(): Short`
- `toInt(): Int`
- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`

- `toChar(): Char`

Часто необходимости в явных преобразованиях нет, поскольку тип выводится из контекста, а арифметические действия перегружаются для подходящих преобразований.

```
val l = 1L + 3 // Long + Int => Long
```

# Операции

Котлин поддерживает стандартный набор арифметических операций над числами: `+`, `-`, `*`, `/`, `%`. Они объявляются членами соответствующих классов.

```
println(1 + 2)
println(2_500_000_000L - 1L)
println(3.14 * 2.71)
println(10.0 / 3)
}
```

Вы также можете переопределить эти операторы для пользовательских классов. См. Перегрузка операторов (operator-overloading.html) для деталей.

## Деление целых чисел

Деление целых чисел всегда возвращат целое число. Любая дробная часть отбрасывается.

```
val x = 5 / 2
// println(x == 2.5) // ОШИБКА: Оператор '==' не может быть применен к 'Int' и 'Doubl
e'
println(x == 2) // true
```

Это справедливо для деления любых двух целочисленных типов.

```
val x = 5L / 2
println(x == 2L) // true
```

Чтобы вернуть тип с плавающей точкой, явно преобразуйте один из аргументов в тип с плавающей точкой.

```
val x = 5 / 2.toDouble()
println(x == 2.5) // true
```

### Побитовые операции

Kotlin поддерживает обычный набор *побитовых операций* над целыми числами. Они работают на двоичном уровне непосредственно с битовыми представлениями чисел. Побитовые операции представлены функциями, которые могут быть вызваны в инфиксной форме. Они могут быть применены только к `Int` и `Long`.

```
val x = (1 shl 2) and 0x000FF000
```

Ниже приведён полный список битовых операций:

- `shl(bits)` – сдвиг влево с учётом знака ( `<<` в Java)
- `shr(bits)` – сдвиг вправо с учётом знака ( `>>` в Java)
- `ushr(bits)` – сдвиг вправо без учёта знака ( `>>>` в Java)
- `and(bits)` – побитовое И
- `or(bits)` – побитовое ИЛИ
- `xor(bits)` – побитовое исключающее ИЛИ
- `inv()` – побитовое отрицание

## Сравнение чисел с плавающей точкой

В этом разделе обсуждаются следующие операции над числами с плавающей запятой:

- Проверки на равенство: `a == b` и `a != b`
- Операторы сравнения: `a < b`, `a > b`, `a <= b`, `a >= b`
- Создание диапазона и проверка диапазона: `a..b`, `x in a..b`, `x !in a..b`

Когда статически известно, что операнды `a` и `b` являются `Float` или `Double` или их аналогами с nullable-значением (тип объявлен или является результатом умного приведения (typecasts.html#smart-casts)), операции с числами и диапазоном, который они образуют, соответствуют стандарту IEEE 754 для арифметики с плавающей точкой (https://en.wikipedia.org/wiki/IEEE_754).

Однако для поддержки общих вариантов использования и обеспечения полного упорядочивания, когда операнды статически *не* объявлены как числа с плавающей запятой (например, `Any`, `Comparable<...>`, параметр типа), операции используют реализации `equals` и `compareTo` для `Float` и `Double`, которые не согласуются со стандартом, так что:

- `NaN` считается равным самому себе
- `NaN` считается больше, чем любой другой элемент, включая "POSITIVE_INFINITY"
- `-0.0` считается меньше, чем `0.0`

# Целые беззнаковые числа

В дополнение к целочисленным типам, в Kotlin есть следующие типы целых беззнаковых чисел:

- `UByte` : беззнаковое 8-битное целое число, в диапазоне от 0 до 255
- `UShort` : беззнаковое 16-битное целое число, в диапазоне от 0 до 65535
- `UInt` : беззнаковое 32-битное целое число, в диапазоне от 0 до $2^{32}$ - 1
- `ULong` : беззнаковое 64-битное целое число, в диапазоне от 0 до $2^{64}$ - 1

Беззнаковые типы поддерживают большинство операций своих знаковых аналогов.

> Изменение типа с беззнакового типа на его знаковый аналог (и наоборот) является *двоично несовместимым* изменением.

## Беззнаковые массивы и диапазоны

> Беззнаковые массивы и операции над ними находятся в стадии бета-тестирования (components-stability.html). Они могут быть несовместимо изменены в любое время.

Как и в случае с примитивами, каждому типу без знака соответствует тип массивов знаковых типов:

- `UByteArray` : массив беззнаковых `byte`
- `UShortArray` : массив беззнаковых `short`
- `UIntArray` : массив беззнаковых `int`
- `ULongArray` : массив беззнаковых `long`

Как и целочисленные массивы со знаком, такие массивы предоставляют API, аналогичный классу `Array` , без дополнительных затрат на оборачивание.

При использовании массивов без знака вы получите предупреждение, что эта функция еще не стабильна. Чтобы удалить предупреждение используйте аннотацию `@ExperimentalUnsignedTypes` . Вам решать, должны ли ваши пользователи явно соглашаться на использование вашего API, но имейте в виду, что беззнаковый массив не является стабильной функцией, поэтому API, который он использует, может быть нарушен изменениями в языке. Узнайте больше о требованиях регистрации (opt-in-requirements.html).

Диапазоны и прогрессии (ranges.html) поддерживаются для `UInt` и `ULong` классами `UIntRange` , `UIntProgression` , `ULongRange` и `ULongProgression` . Вместе с целочисленными беззнаковыми типами эти классы стабильны.

## Литералы

Чтобы целые беззнаковые числа было легче использовать, в Kotlin можно помечать целочисленный литерал суффиксом, указывающим на определенный беззнаковый тип (аналогично `Float` или `Long`):

- `u` и `U` помечают беззнаковые литералы. Точный тип определяется на основе ожидаемого типа. Если ожидаемый тип не указан, компилятор будет использовать `UInt` или `ULong` в зависимости от размера литерала.

```
val b: UByte = 1u  // UByte, есть ожидаемый тип
val s: UShort = 1u // UShort, есть ожидаемый тип
val l: ULong = 1u  // ULong, есть ожидаемый тип

val a1 = 42u // UInt: ожидаемого типа нет, константе подходит тип UInt
val a2 = 0xFFFF_FFFF_FFFFu // ULong: ожидаемого типа нет, тип UInt не подходит конста
нте
```

- `uL` and `UL` явно помечают литерал как `unsigned long`.

```
val a = 1UL // ULong, даже несмотря на то, что ожидаемого типа нет и константа вписыв
ается в UInt
```

### Дальнейшее обсуждение

См. предложения для беззнаковых типов (https://github.com/Kotlin/KEEP/blob/master/proposals/unsigned-types.md) для технических деталей и дальнейшего обсуждения.

# Логический тип

Тип `Boolean` представляет логический тип данных и принимает два значения: `true` и `false`.

При необходимости использования nullable-ссылок логические переменные оборачиваются `Boolean?`.

Встроенные действия над логическими переменными включают:

- `||` – ленивое логическое *ИЛИ*
- `&&` – ленивое логическое *И*
- `!` – отрицание

```
val myTrue: Boolean = true
val myFalse: Boolean = false
val boolNull: Boolean? = null

println(myTrue || myFalse)
println(myTrue && myFalse)
println(!myTrue)
```

> В JVM: nullable-ссылки на логические объекты заключены в рамки аналогично числам.

# Символы

Символы в Kotlin представлены типом `Char`. Символьные литералы заключаются в одинарные кавычки: `'1'`.

Специальные символы начинаются с обратного слеша `\`. Поддерживаются следующие escape-последовательности: `\t`, `\b`, `\n`, `\r`, `\'`, `\"`, `\\` и `\$`.

Для кодирования любого другого символа используйте синтаксис escape-последовательности Юникода: `'\uFF00'`.

```
val aChar: Char = 'a'

println(aChar)
println('\n') // выводит дополнительный символ новой строки
println('\uFF00')
```

Если значение символьной переменной – цифра, её можно явно преобразовать в `Int` с помощью функции `digitToInt()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/digit-to-int.html).

> В JVM: Подобно числам, символы оборачиваются при необходимости использования nullable-ссылки. При использовании обёрток тождественность (равенство по ссылке) не сохраняется.

# Строки

Строки в Kotlin представлены типом `String`. Как правило, строка представляет собой последовательность символов в двойных кавычках ( " ).

```
val str = "abcd 123"
```

Строки состоят из символов, которые могут быть получены по порядковому номеру: `s[i]`. Проход по строке выполняется циклом `for`.

```
for (c in str) {
    println(c)
}
```

Строки являются неизменяемыми. После инициализации строки вы не можете изменить ее значение или присвоить ей новое. Все операции, преобразующие строки, возвращают новый объект `String`, оставляя исходную строку неизменной.

```
val str = "abcd"
println(str.uppercase()) // Создается и выводится новый объект String
println(str) // исходная строка остается прежней
```

Для объединения строк используется оператор `+`. Это работает и для объединения строк с другими типами, если первый элемент в выражении является строкой.

```
val s = "abc" + 1
println(s + "def") // abc1def
```

Обратите внимание, что в большинстве случаев использование строковых шаблонов или обычных строк предпочтительнее объединения строк.

## Строковые литералы

В Kotlin представлены два типа строковых литералов:

- *экронированные* строки с экранированными символами
- *обычные* строки, которые могут содержать символы новой строки и произвольный текст

Вот пример экранированной строки:

```
val s = "Hello, world!\n"
```

Экранирование выполняется общепринятым способом, а именно с помощью обратного слеша ( \ ). Список поддерживаемых escape-последовательностей см. в разделе Символы выше.

Обычная строка выделена тройной кавычкой ( """ ), не содержит экранированных символов, но может содержать символы новой строки и любые другие символы:

```
val text = """
  for (c in "foo")
    print(c)
"""
```

Чтобы удалить пробелы в начале обычных строк, используйте функцию `trimMargin()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/trim-margin.html).

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

По умолчанию | используется в качестве префикса поля, но вы можете выбрать другой символ и передать его в качестве параметра, например, `trimMargin(">")` .

## Строковые шаблоны

Строки могут содержать *шаблонные* выражения, т.е. участки кода, которые выполняются, а полученный результат встраивается в строку. Шаблон начинается со знака доллара ( $ ) и состоит либо из простого имени (например, переменной),

```
val i = 10
println("i = $i") // выведет "i = 10"
```

либо из произвольного выражения в фигурных скобках.

```
val s = "abc"
println("$s.length is ${s.length}") // выведет "abc.length is 3"
```

Шаблоны поддерживаются как в обычных, так и в экранированных строках. При необходимости вставить символ `$` в обычную строку (такие строки не поддерживают экранирование обратным слешом) перед любым символом, который разрешен в качестве начала идентификатора, используйте следующий синтаксис:

```
val price = """
${'$'}_9.99
"""
```

# Массивы

Массивы в Kotlin представлены классом `Array` , обладающим функциями `get` и `set` (которые обозначаются `[]` согласно соглашению о перегрузке операторов), и свойством `size` , а также несколькими полезными встроенными функциями.

```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit

    operator fun iterator(): Iterator<T>
    // ...
}
```

Для создания массива используйте функцию `arrayOf()` , которой в качестве аргумента передаются элементы массива, т.е. выполнение `arrayOf(1, 2, 3)` создаёт массив `[1, 2, 3]` . С другой стороны функция `arrayOfNulls()` может быть использована для создания массива заданного размера, заполненного значениями `null` .

Также для создания массива можно использовать фабричную функцию, которая принимает размер массива и функцию, возвращающую начальное значение каждого элемента по его индексу.

```
// создаёт массив типа Array<String> со значениями ["0", "1", "4", "9", "16"]
val asc = Array(5) { i -> (i * i).toString() }
asc.forEach { println(it) }
```

Как отмечено выше, оператор `[]` используется вместо вызовов встроенных функций `get()` и `set()`.

Обратите внимание: в отличие от Java массивы в Kotlin являются *инвариантными*. Это значит, что Kotlin запрещает нам присваивать массив `Array<String>` переменной типа `Array<Any>`, предотвращая таким образом возможный отказ во время исполнения (хотя вы можете использовать `Array<out Any>`, см. Проекции типов (generics.html#type-projections)).

## Массивы примитивных типов

Также в Kotlin есть особые классы для представления массивов примитивных типов без дополнительных затрат на оборачивание: `ByteArray`, `ShortArray`, `IntArray` и т.д. Данные классы не наследуют класс `Array`, хотя и обладают тем же набором методов и свойств. У каждого из них есть соответствующая фабричная функция:

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

```
// int массив, размером 5 со значениями [0, 0, 0, 0, 0]
val arr = IntArray(5)

// инициализация элементов массива константой
// int массив, размером 5 со значениями [42, 42, 42, 42, 42]
val arr = IntArray(5) { 42 }

// инициализация элементов массива лямбда-выражением
// int массив, размером 5 со значениями [0, 1, 2, 3, 4] (элементы инициализированы св
оим индексом)
var arr = IntArray(5) { it * 1 } …
```