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.

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?`.