KOTLIN

Quick instalation guide

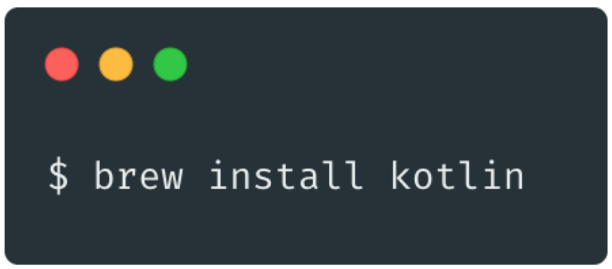
Let's begin with one of the firsts things that everyone does when starting a new technology, **installing it**!

Due to Kotlin being a JVM language, it has the power to call Java code and vice versa. This also means that you need to have Java JDK installed in your machine (I suggest you have installed Java JDK 8).

To install the compiler on Windows, you have to download the kotlinc and unpack the ZIP file. The output folder will contain a subfolder bin with all the required scripts to compile and run Kotlin. Now you need to add this bin subfolder to the PATH in order to call kotlinc without having to specify the full path.

In Linux or OS X, you can do the same as in Windows or run these commands:

If you use OS X with homebrew:



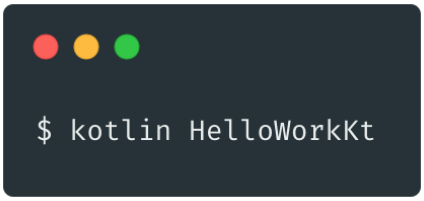
After any of these steps, you can make the mythical **HelloWorld.kt** file with the following code:

```kotlin
fun main(args: Array<String>) {
    println("Hello, World")
}
```

Then invoke the command line:

```
$ kotlinc HelloWorld.kt
```

This will create a **HelloWorkKt.class** file. Thereafter, run the class with:



```
$ kotlin HelloWorkKt
```

and tadaaa! **Welcome to Kotlin**.

Easy, right?

Tip #1: if you want to create a JAR, you can by adding the flag –include-runtime:

```
$ kotlinc HelloWorld.kt -include-runtime -d HelloWorld.jar
```

Then run with:

```
$ java -jar HelloWorld.jar
```

```
$ java -jar HelloWorld.jar
```

```
$ java -jar HelloWorld.jar
```

Tip #2: with the created **HelloWorldKt.class**, you can check its bytecode with the following command:

```
$ javap -c HelloWorldKt.class
```

Tip #3: Typing only kotlinc in the command line you'll have access to a REPL.

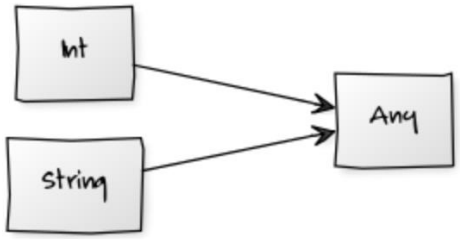
If you are an IDE person, you can use IntelliJ. From version 15.0, IntelliJ comes bundled with Kotlin. If you're using an older version, you can add Kotlin by installing the plugin, Setting > Plugins > Install IntelliJ plugins and type Kotlin.

KOTLIN TYPE SYSTEM

Kotlin's type hierarchy has very few rules to learn. Those rules fit together consistently and predictably. Thanks to them, Kotlin can provide useful, user extensible language features – null safety, polymorphism, and unreachable code analysis – without resorting to special cases and ad-hoc checks in the compiler and IDE.

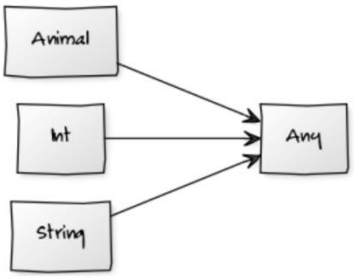
All types of Kotlin object are organized into a hierarchy of **subtype/supertype** relationships.

All types are subtypes of **Any**.



Any is the equivalent Java's Object class. Unlike Java, Kotlin does not distinguish the primitive types from user-created types. If you define a class that is not explicitly derived from another, it will be an immediate subtype of **Any.**

```
1 open class Animal(color : String)
```



If we create a class Cat that is a subtype of Animal class:

```kotlin
class Cat(numberLegs : Int, color : String) : Animal(color)
```

Kotlin enforces a **subtype/supertype** relationship. For instance, you can store a subtype into a supertype:

```kotlin
val animal: Animal = Cat(4, "white")
```

But you cannot store a supertype into a subtype variables:

```
>>> open class Animal(color : String)
>>> class Cat(numberLegs : Int, color : String) : Animal(color)
>>> val cat = Cat(4, "white")
>>> val a: Animal = cat
>>> val a2: Cat = a
error: type mismatch: inferred type is Line_0.Animal but Line_1.Cat was expected
val a2: Cat = a
              ^
```

UNIT

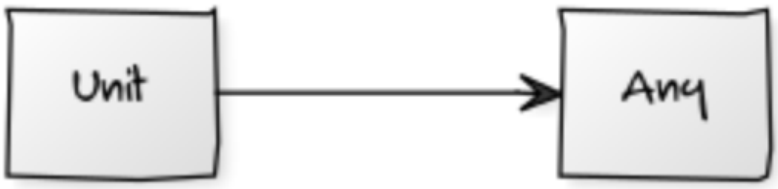
In Kotlin every function must return something, therefore there's no void functions, instead it must be returned **Unit. Unit** is the return type of any function that does not return any meaningful value.

Unlike void, Unit has a value also called **Unit**. You don't need to explicitly specify **Unit** as a return type. If you write a function and don't specify the result type, the compiler will treat it as a Unit function.

```
>>> fun example1() { println("block body and no explicit return type, so return Unit") }
>>> val u1 : Unit = example1()
block body and no explicit return type, so return Unit
>>> fun example2() = println("inline function also with return type Unit")
>>> val u2 : Unit = example2()
inline function also with return type Unit
>>> val u2 : Any = example1()
block body and no explicit return type, so return Unit
>>> val u1 : Any = example2()
inline function also with return type Unit
>>> val u1 : Int = example2()
error: type mismatch: inferred type is Unit but Int was expected
val u1 : Int = example2()
               ^

>>> val u1 : Int = example1()
error: type mismatch: inferred type is Unit but Int was expected
val u1 : Int = example1()
               ^
```

As we can see in the following image, like any other types, **Unit** is a subtype of **Any**.

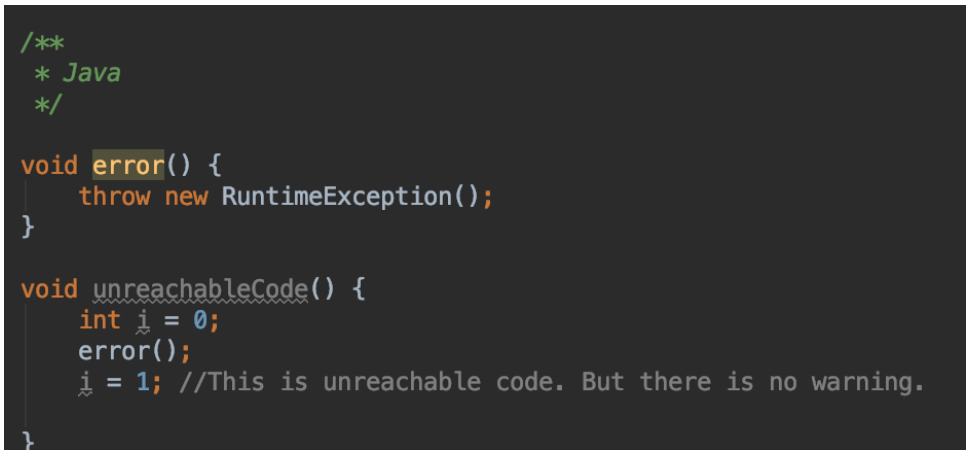


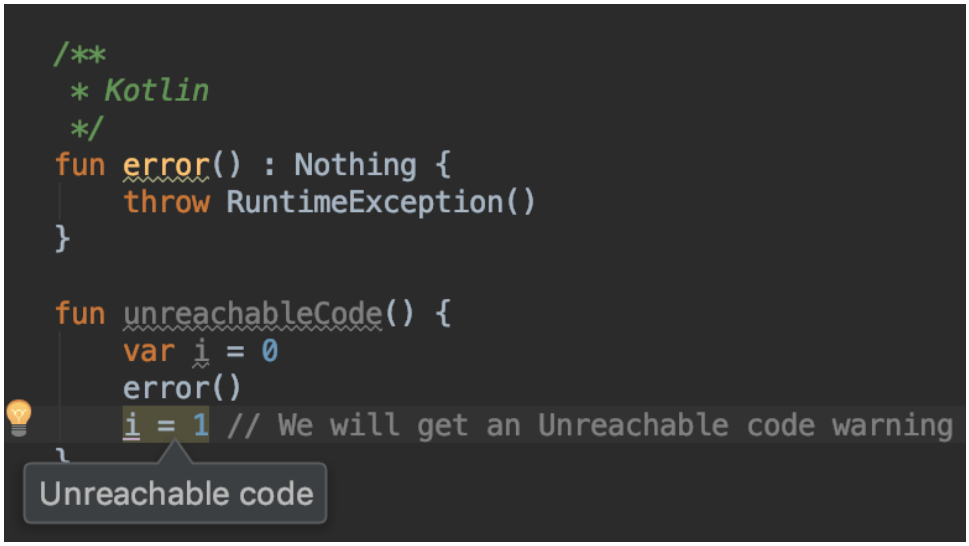
TIPO NOTHING

At the very bottom of the Kotlin type hierarchy there's the **Nothing** type.

There's no type like Nothing in Java. This type is used when a function will never terminate normally and therefore a return value has no sense. While an expression of type Unit results in the singleton value **Unit**, an expression of type **Nothing** never returns at all.

Using **Nothing** as a return type also helps the IDE to detect Unreachable code:





Nothing is a subtype of every other type, this allows any expression to fail. Nothing is useful to throw exception or for testing.

```kotlin
//Kotlin
fun error() : Nothing {
    throw RuntimeException()
}

fun funcExample() : String {
    var i = 0
    error()
    // It will compile
}
```

Don't forget, if you want a function to return **Nothing**, you must explicitly declare it, otherwise Unit will be returned!

VALS E VARS

You should have already noticed that in Kotlin there's only two keywords for declaring variables, **val and var**. Var is a mutable variable, it can be initialized later and its value can change over time. Also, you don't need to specify the variable type:

After you assign a given type to a **var** you cannot change its type.

```kotlin
//Impossible to change the type of the variable
var name : String = "Bruno"
name = 1 //Error: the integer literal does not conform to the expected type String
```

The keyword **val** is used to declare read-only variables. It's the equivalent of declaring a final variable in Java.

A variable with the keyword **val** must be initialized when it's created because its value cannot be changed later.

```kotlin
//Java
private final String name = "Bruno";

//Kotlin
private val name "Bruno"
//or
private val name : String = "Bruno"


name = "Azevedo" //Error: val cannot be reassigned
```

NULLABLE TYPES

Due to Null references being present in the **JVM**, we must deal with them, but Kotlin helps us avoid some common mistakes.

Kotlin distinguishes "non-null" and "nullable" types. "Non-null" types cannot have null, if you are using a "non-null" type you are guaranteeing that you will not encounter a null in it and it will never throw a **NullPointerException**.

```
>>> var aux : String = "Non-null type"
>>> aux = null
error: null can not be a value of a non-null type String
aux = null
      ^
```

If you try to do the previous example it will result in a compile time error.

If you want a variable that can be null, you need to use a nullable type. To do so, you need to add the suffix '?' to the variable type. For example, **Int?** is the nullable equivalent Int and therefore allows all **Int** value plus the null.

```
>>> var i : Int? = 1
>>> i = null
>>> i = 2
>>> i = null
```

Nullable type cannot be infer, if you want a nullable variable you need to specify it. If you don't specify it and you instantiate a variable as null, it will be of the type **Nothing?,** that's right, Nothing can be null too, in fact Nothing? is the type of null.

```
>>> var a = null
>>> a = 1
error: the integer literal does not conform to the expected type Nothing?
a = 1
    ^
```

BASIC TYPES

To wrap this up and now that we know a little more about Kotlin Types, let's talk about Kotlin Basic Types.

In Kotlin everything is an object, and if you are familiar with Java you might be aware that there are special primitive types which are treated differently from objects. For instance, these types do not support method/function calls.
As a work around, Java introduced a wrapper object to these primitive types, for example for the primitive type **boolean** there's **java.lang.Boolean**. In Kotlin this is not needed because these primitives are objects too. For performance reasons, Kotlin compiler will map basic types back to JVM primitive whenever possible. This is not possible if you want a type to be nullable.

So, which are the Kotlin Basic Types? Numbers, Characters, Booleans, Arrays and Strings (there's also Unsigned Integers, but currently they are experimental so I will not talk about them).

NUMBERS

Type	Bit width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

Kotlin does not support automatic widening of numbers, so conversion must be invoked:

```
>>> val int : Int = 123
>>> val long : Long = int.toLong()
>>> println(long)
123
```

The full set of functions for conversions between types is: **toByte(), toShort(), toInt(), toLong(), toFloat(), toDouble(), toChar().**

You can create a number literal using one of the following forms:

```
1 //Number literals
2 val int = 123
3 val long = 123456L
4 val double = 1.23
5 val float = 1.23F
6 val hexadecimal = 0×AA
7 val binary = 0b01010101
```

NOTE: Octal literals are not supported.

For more readability Kolin allows the use of underscore:

```
1 val oneMillion = 1_000_000
2 val creditCardNumber = 1234_5678_9012_3456L
3 val socialSecurityNumber = 999_999_99L
4 val hexBytes = 0×FF_EC_DE_5E
5 val bytes = 0b01010101_10101010_01010101
```

You can also use bitwise operators – left shift, right shift, unsigned right shift, logical and, logical or and exclusive logical or – but unlike Java, they are not built in operators but named

functions instead. (Inverse is a unary operator, and so is invoked using the dot syntax on a number).

```
1 //Bitwise
2 val leftShift = 1 shl 2
3 val rightShift = 1 shr 2
4 val unsignedRightShift = 1 ushr 2
5 val and = 0×10101010 and 0×01010101
6 val or = 0×10101010 or 0×01010101
7 val xor = 0×10101010 xor 0×01010101
8 val inv = 1.inv()
```

CHARACTERS

Represented by the type Char, they cannot be treated as Number:

```
1 fun check(c : Char) {
2     if (c == 1) { //Error: Incompatible types
3           ...
4     }
5 }
```

Char represent a single character, it's literals use single quotes like: '1'. Specials characters can be escaped with: \t, \b, \n, \r, ', ", \\ and \$

BOOLEANS

Like Java Boolean support the usual negation (!), conjunction (&&) and disjunction (||) operations. It has two values: true and false.

Conjunction and disjunction are lazily evaluated, so if the left-hand side satisfies the clause, then the right-hand side will not be evaluated.

ARRAYS

Represented by the Array class with get and set functions and size property and some other useful member functions like Iterator.

To create an array you need to use the library function arrayOf():

```
1 //Array
2 val array = arrayOf(1,2,3,4,5) //Creates array [1,2,3,4,5]
```

You can use the Array constructor to create an Array. The constructor takes the array size and a function that can return the initial value of each array element given its index, the following example will return an Array<String>:

```
>>> val asc = Array(5, { i -> (i * i).toString() })
>>> asc.forEach { println(it) }
0
1
4
9
16
>>> asc[0] is String
res63: kotlin.Boolean = true
>>> asc[0] is Int
error: incompatible types: Int and String
asc[0] is Int
          ^
```

You can use bracket syntax like C:

```
>>> val array = arrayOf(1,2,3)
>>> val element0 = array[0]
>>> val element1 = array[1]
>>> val element2 = array[2]
>>> println(element0)
1
>>> println(element1)
2
>>> println(element2)
3
```

For performance, Kotlin provides alternative array classes that are specialized for each of the primitive types, these classes allow code to use arrays as efficiently as they would do in Java. The classes are: **IntArray, LongArray, DoubleArray, FloatArray, BooleanArray, ByteArray, CharArray, ShortArray**

```
>>> val x : DoubleArray = doubleArrayOf(1.1, 2.2)
>>> x.forEach { println(it) }
1.1
2.2
>>> val x : BooleanArray = booleanArrayOf(true, false)
>>> x.forEach { println(it) }
true
false
```

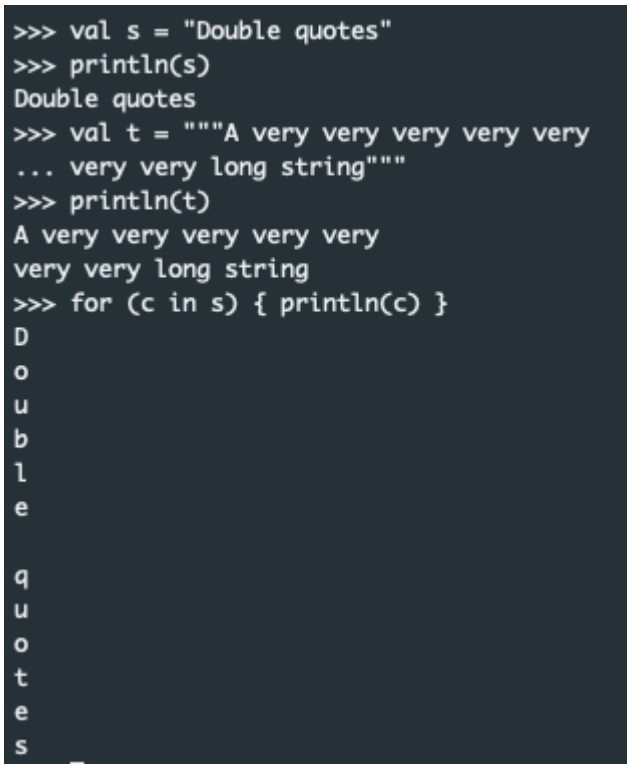

```
>>> val x : DoubleArray = doubleArrayOf(1.1, 2.2)
>>> x.forEach { println(it) }
1.1
2.2
>>> val x : BooleanArray = booleanArrayOf(true, false)
>>> x.forEach { println(it) }
true
false
```

```
>>> val x : DoubleArray = doubleArrayOf(1.1, 2.2)
>>> x.forEach { println(it) }
1.1
2.2
>>> val x : BooleanArray = booleanArrayOf(true, false)
>>> x.forEach { println(it) }
true
false
```


Note: unlike Java, arrays in Kotlin are invariant. This means that Kotlin does not let us assign an Array<String> to an Array<Any>, which prevents a possible runtime failure (but you can use Array<out Any>. Hmmm, Array<out Any> there's something for another post.

STRINGS

Represented by the type String, like in Java, Strings are immutable. They are created using double quotes or triple double quotes. Double quotes create an escaped string, in an escaped string, special characters, such as new line must be escaped. Triple double quotes are used to create a raw string that is useful for strings that span many lines. Elements in a String can be accessed by the indexing operation. Strings also provide an iterator function which can be used in a for loop:



IN SHORT

We've talked about how you can install Kotlin and about Kotlin Type System but this is not the end!! If you want to know more about Kotlin there's great info in the internet:

- Kotlin Official Website: Kotlinlang.org;
- A Medium Blog with hundreds of Kotlin related articles.