KOTLIN

Guia prático de instalação

Começamos com uma das primeiras tarefas que alguém leva a cabo quando pretende utilizar uma nova tecnologia: **a instalação!**

Devido ao facto de Kotlin ser uma linguagem JVM, tem o poder de chamar código Java e vice-versa. Isto também significa que precisa de ter o Java JDK instalado na sua máquina (eu sugiro que tenha a versão 8 do Java JDK).

Para instalar o *compiler* no Windows, é necessário fazer o download do *kotlinc* e abrir o ficheiro ZIP. A pasta irá conter uma subpasta com todos os scripts necessários para compilar e correr o Kotlin. Agora, é necessário adicionar esta subpasta ao PATH, para conseguir chamar o *kotlinc* sem ter de especificar o caminho todo.

Em Linux ou em OS X, pode fazer os mesmo passos do que em Windows ou correr estes comandos:

```
$ curl -s https://get.sdkman.io | bash

$ bash

$ sdk install kotlin
```

Se utiliza o OS X com o *homebrew*:

```
$ brew install kotlin
```

Depois de efetuar estes passos, pode criar o mítico ficheiro **HelloWorld.kt** com o seguinte código:

```
1 fun main(args: Array<String>)  {
2     println(“Hello, World”)
3 }
```

Depois disso, invoque esta linha de comando:

```
$ kotlinc HelloWorld.kt
```

Estas ações irão criar um ficheiro **HelloWorld.class**. De seguida, corra a *class* com:

```
$ kotlin HelloWorkKt
```

E tadaaa! **Bem-vindo ao Kotlin.**

Fácil, certo?

Dica #1: se quiser criar um JAR, pode fazê-lo adicionando uma *flag* –incluide-*runtime*:

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

Depois corra com:

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

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

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

Dica #2: com o ficheiro **HelloWorld.class** criado, pode verificar o seu *bytecode* através do seguinte comando:

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

Dica #3: escrever apenas kotlinc na linha de comando dá acesso ao REPL.

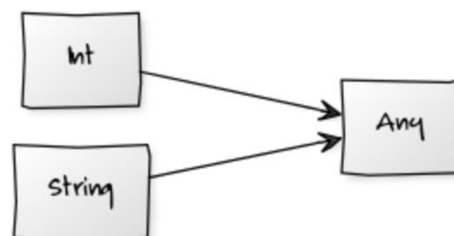
Se é uma pessoa de IDE, pode utilizar IntelliJ. Desde a versão 15.0, o IntelliJ já vem com Kotlin. Se utiliza uma versão mais antiga, pode adicionar Kotlin através da instalação de um *plugin*, **Setting > Plugins > Install IntelliJ** plugins e procurar por Kotlin.

KOTLIN TYPE SYSTEM

A hierarquia de tipos do Kotlin tem pouquíssimas regras para aprender. Essas regras encaixam umas nas outras de forma consistente e previsível. Graças a estas mesmas regras, Kotlin oferece recursos de linguagem úteis – *null safety, polymorphism,* e análise de código inacessível – sem recorrer a casos especiais ou verificações *ad-hoc* no compilador ou no IDE.

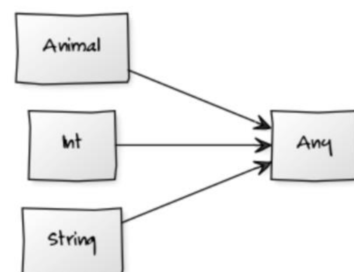
Todos os tipos de objetos do Kotlin estão organizados por hierarquias relacionais de **subtipos/supertipos**.

Todos os tipos são subtipos de **Any**.



Any é o equivalente à classe Object de Java. Ao contrário de Java, Kotlin não distingue os tipos primitivos dos tipos criados pelo usuário. Se definir uma classe que não é explicitamente derivada de outra, será um subtipo de **Any**.

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



Se criar uma *class* Cat, que é um subtipo da *class* Animal:

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

Kotlin reforça uma relação **subtipo/supertipo**. Por exemplo, pode guardar um subtipo num supertipo:

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

Mas não pode armazenar um supertipo nas variáveis de um subtipo:

```
>>> 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

Em Kotlin, todas as funções devem devolver algo, por isso, não existem funções void. Ao invés, deve ser devolvido **Unit**. **Unit** é o tipo de retorno de qualquer função que não retribua nenhum valor significativo.

Ao contrário das funções void, **Unit** tem um valor chamado **Unit**. Não é necessário definir **Unit** como um tipo de retorno, se escrever uma função e não especificar o tipo de resultado, o compilador irá tratá-lo como uma função Unit.

```
>>> 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()
               ^
```

Como podemos ver na seguinte imagem, tal como todos os outros tipos, **Unit** é um subtipo de **Any**.

TIPO NOTHING

Na parte inferior da hierarquia de tipos Kotlin, há o tipo **Nothing**.

Não existe nenhum tipo como este, em Java. Este tipo é utilizado quando uma função não terminará normalmente, e, por isso, um valor de retorno não faz sentido. Enquanto uma expressão do tipo **Unit** resulta no valor *singleton* **Unit**, a expressão **Nothing** nunca retorna.

Utilizar o **Nothing** como um tipo de retorno ajuda o IDE a detetar código inatingível:

```
/**
 * Java
 */

void error() {
    throw new RuntimeException();
}

void unreachableCode() {
    int i = 0;
    error();
    i = 1; //This is unreachable code. But there is no warning.
}
```

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

fun unreachableCode() {
    var i = 0
    error()
    i = 1 // We will get an Unreachable code warning
}
Unreachable code
```

O tipo **Nothing** é subtipo de qualquer outro tipo, o que permite que qualquer expressão falhe. **Nothing** é útil para lançar exceções e fazer testes.

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

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

Não se esqueça, se pretende que uma função devolva o tipo **Nothing**, deve declará-lo explicitamente, se não, será **Unit** a ser devolvida.

VALS E VARS

Já deve ter reparado que em Kotlin só existem duas palavras-chave para declarar variáveis, **val** e **var**. O **var** é uma variável mutável, que pode ser iniciada mais tarde e o seu valor altera-se com o tempo. Para além disso, não precisa de especificar o tipo de variável:

```kotlin
1 //will infer the type
2 var name = "Bruno"
3 //Changing value
4 name = "Azevedo"
5
6 //Late initialization
7 var name : String
8 name = "Bruno"
```

Depois de atribuir determinado tipo a um **var**, não pode mudá-lo de seguida.

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

A palavra-chave **val** é utilizada para declarar variáveis apenas de leitura. É o equivalente a declarar uma variável final em Java.

A variável com a palavra-chave **val** deve ser iniciada quando criada, já que o seu valor não pode ser alterado posteriormente.

```
1 //Java
2 private final String name = "Bruno";
3
4 //Kotlin
5 private val name "Bruno"
6 //or
7 private val name : String = "Bruno"
8
9
10 name = "Azevedo" //Error: val cannot be reassigned
```

TIPOS NULLABLE

Devido ao facto de as referências nulas estarem presentes no JVM, temos mesmo de lidar com elas, sendo que o Kotlin nos ajuda a evitar erros mais comuns.

Kotlin distingue tipos: **non-null** e **nullable**. Os tipos **non-null** não podem ter **null** – ou seja, se estiver a utilizar um tipo **non-null** é garantido que não irá encontrar um **null** e nunca irá gerar um **NullPointerException**.

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

Se tentar aplicar o exemplo prévio, irá resultar num erro de tipo *compile error*.

Se pretende que uma variável seja **null**, deve utilizar um tipo **nullable**. Para escolhê-lo, deve acrescentar o sufixo '?' ao tipo da variável. Por exemplo, Int? é o equivalente **nullable de Int** e, por isso, permite todos os valores **Int** mais os **null**.

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

O tipo **nullable** não pode ser inferido, por isso, se pretende uma variável **nullable**, é então necessário especificá-la. Se não a especificar e se a instanciar como variável **null**, será do tipo **Nothing?**. É verdade, o tipo **Nothing** pode ser **null** também.

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

TIPOS BÁSICOS

No Kotlin tudo é um objeto, e, se estiver familiarizado com Java, é possível que já esteja consciente de que existem tipos primitivos que são tratados de forma diferente da dos objetos. Por exemplo, esses tipos não suportam chamadas de métodos/funções.

Como solução, o Java introduziu um objeto *wrapper* a esses tipos, por exemplo, para o tipo primitivo *Boolean* existe o java.lang.Boolean. Nesta linguagem, este processo não se torna necessário, uma vez que os tipos primitivos são objetos também. Por motivos de desempenho, o compilador de Kotlin fará o mapeamento dos tipos básicos de volta para o primitivo JVM sempre que possível (isto só não será possível se quiser que o tipo seja **nullable)**.

Então… quais são os tipos básicos de Kotlin? Números, characters, Booleans, Arrays e Strings (também existem os Unsigned Integers, mas, neste momento, são apenas experimentais).

NÚMEROS

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

O Kotlin não suporta o alargamento automático de números, portanto, a conversão deve ser invocada:

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

O *set* completo de funções para conversões entre tipos é: **toByte(), toShort(), toInt(), toLong(), toFloat(), toDouble(), toChar().**

Pode criar um número literal utilizando uma das seguintes formas:

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

NOTA: *octal literals* não são suportados.

Para maior legibilidade, o Kotlin permite o uso de *underscores*:

```
1 val oneMillion = 1_000_000
2 val creditCardNumber = 1234_5678_9012_3456L
3 val socialSecurityNumber = 999_999_99L
4 val hexBytes = 0xFF_EC_DE_5E
5 val bytes = 0b01010101_10101010_01010101
```

Também pode optar por utilizar operadores *bit a bit* – *left shift*, *right shift, unsigned right shift, logical and, logical or and exclusive logical or* – mas, ao contrário de Java, eles não são

construídos em operadores, mas em funções nomeadas (o *inverse* é um operador *unary*, e, portanto, é invocado utilizando a sintaxe de pontos em números).

```
//Bitwise
val leftShift = 1 shl 2
val rightShift = 1 shr 2
val unsignedRightShift = 1 ushr 2
val and = 0x10101010 and 0x01010101
val or = 0x10101010 or 0x01010101
val xor = 0x10101010 xor 0x01010101
val inv = 1.inv()
```

CHARACTERS

Representado pelo tipo Char, os Characters não podem ser tratados como Números.

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

O Char é representado por um único *character*, os seus *literals* utilizam *quotes* simples, tais como: '1'. Os caracteres especiais podem ser executados com: \ t, \ b, \ n, \ r, ', ", \\ e \ $

BOOLEANS

Tal como o suporte do Java Boolean, as operações mais usuais são o ***negation (!)***, ***conjunction (&&)*** e ***disjunction (| |)***. Tem dois valores: verdadeiro ou falso.

Conjunction e disjunction são avaliados da seguinte forma: se o lado esquerdo satisfizer a cláusula, o lado direito não vai ser avaliado.

ARRAYS

Representado pela classe Arrays com funções *get* e *set* e com propriedades de tamanho, mas também com outras funções de membro como o ***Iterator***.

Para criar um Array é necessário utilizar a função da biblioteca **arrayOff():**

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

Pode utilizar o construtor do Array para criar um Array. O construtor pega no tamanho e uma das funções que consegue retornar o valor inicial de cada elemento da array, dado o seu índice. O exemplo seguinte mostra como retornará o 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
          ^
```

Pode utilizar a sintaxe *bracket* como em 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
```

Para obter melhor desempenho, o Kotlin fornece classes de Array alternativas que são especializadas para cada um dos tipos primitivos. Estas classes permitem que o código use Arrays com a mesma eficiência que utilizariam em Java. As classes são: **IntArray, LongArray, DoubleArray, 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
```

Nota: ao contrário de Java, os Arrays em Kotlin são invariáveis. Isto significa que Kotlin não nos permite atribuir um Array <String> a um Array <Any>, o que acaba por evitar uma possível falha de tempo de execução (no entanto, pode utilizar Array <out Any> - mas isto é algo para outro post.

STRINGS

Representado pelo tipo String, tal como em Java, as Strings são imutáveis.

São criados utilizando aspas duplas ou triplas. As aspas duplas criam um *escaped string*, num *escaped String*, caracteres especiais, tal como novas linhas devem ser *escaped*.

Aspas duplas e triplas são utilizadas para criar uma *string raw* que possa ser útil para strings que abrangem várias linhas.

Os elementos numa String podem ser encontrados através de uma operação de indexação. Os Strings também fornecem uma função de iteração, que pode ser utilizada em loop:

```
>>> val s = "Double quotes"
>>> println(s)
Double quotes
>>> val t = """A very very very very very
... very very long string"""
>>> println(t)
A very very very very very
very very long string
>>> for (c in s) { println(c) }
D
o
u
b
l
e

q
u
o
t
e
s
```

EM RESUMO

Neste post falámos sobre como instalar **Kotlin** e **Kotlin Type System**, mas, se quiser saber ainda mais sobre esta linguagem, existem ótimos sites para retirar informação:

- Site oficial de Kotlin: Kotlinlang.org;
- Blog do Medium com centenas de artigos relacionados com Kotlin.