



# Kotlin

## Quick start

*Source: <http://kotlinlang.org>*

# Kotlin

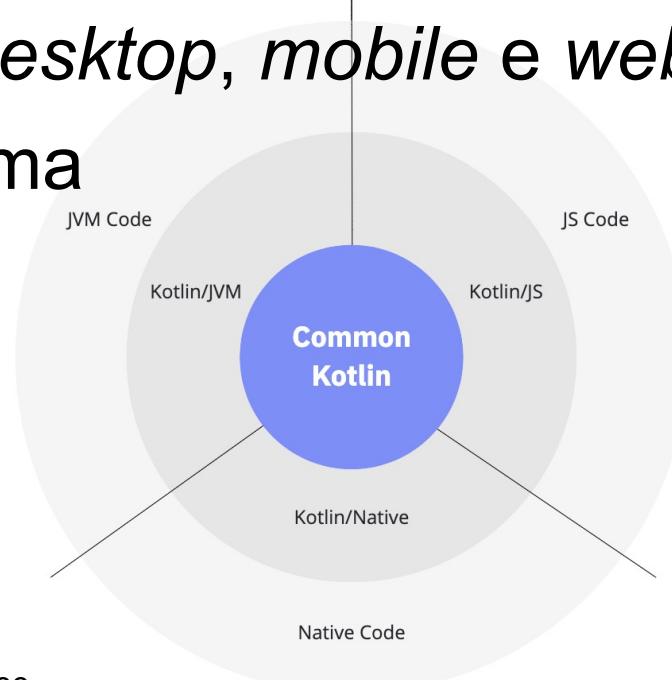
- # Linguagem Kotlin surgiu de um projecto iniciado em 2010-2011 pela Jetbrains no sentido de criar uma linguagem *opensource* mais concisa e mais actual para responder às necessidades dos programadores
- # Inspirada noutras linguagens: Java, C#, Scala, Groovy, ...



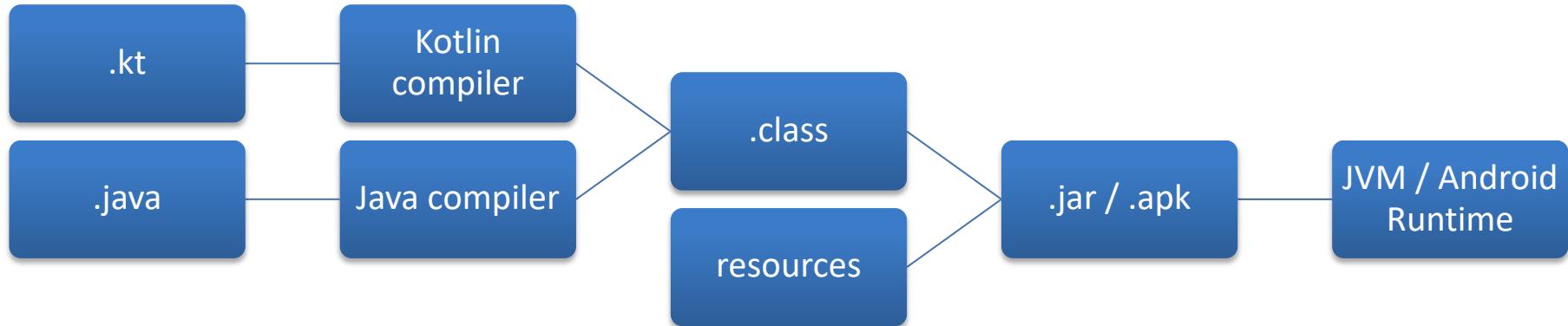
# Kotlin

# Kotlin

- # Reconhecida como a linguagem para desenvolvimento *Android* mas...
  - # Totalmente interoperável com a linguagem Java
  - # Interoperável com muitas outras linguagens e plataformas de desenvolvimento *server-side* ou *client-side* (*desktop*, *mobile* e *web*)
- # Multiplataforma



# Interoperabilidade Java/Kotlin



# Hello world

```
fun main() {  
    println("Hello World!!!")  
}
```

# Conceitos e sintaxe básica

- # Facilmente aprendida por quem já possui alguns conhecimentos de outras linguagens orientadas a objectos
  - # Principalmente se possui prática em Java, C# ou Swift
- # Diferenças óbvias na primeira aproximação à linguagem
  - # Não utilização de ; para finalizar linhas de código
  - # A declaração das variáveis é realizada por ordem inversa à “habitual”
    - # Primeiro indica-se o nome e depois o tipo
      - # Adicionalmente, as declarações são precedidas por uma palavra chave indicativa se é imutável (“val”) ou mutável (“var”)
  - # Funções/métodos iniciados com uma palavra chave (“fun”) e o tipo de retorno é indicado depois dos parâmetros
  - # Permite programação funcional e orientada a objectos
  - # Não é necessário usar uma palavra específica (“new”) para criar instâncias de objectos
  - # Não existe o ciclo for no formato tradicional: `for(ini;cond;inc)`
    - # Existe apenas o ciclo for no formato *for each*: `for( x in xxx)`
  - # O switch é substituído pelo when
    - # Permite a definição de casos mais flexíveis

# Variáveis mutáveis e imutáveis

- # Declaração de variáveis, usando a palavra ‘val’ para variáveis imutáveis e ‘var’ para variáveis mutáveis

```
# {[const] <val> | var} <name> [ : <type> ] [ = <value> ]
var a : Int
val b : Double = 5.5
var c = 123
const val d = "ISEC"
```

- # Os tipos das variáveis podem ser inferidos pelos valores atribuídos

- # Tipos básicos

- # Byte, Short, Int, Long, UByte, UShort, UInt, ULong
- # Float, Double
- # Char, String
- # Boolean

# input e output (consola)

# output

# print

# println

# println("DEIS")

# input

# readLine

# var s = readLine()

# ... também é possível importar e usar a classe  
Scanner normalmente usada no Java

# val sc = Scanner(System.`in`)

# Strings e Ranges

## # Strings

```
val i = 123
println("i = $i ${i+1}")
```

```
var s = """
    texto mais longo
    com varias linhas
"""
```

## # Ranges

```
# 1..20
# 20 downto 1
# 1..20 step 3
# 20 downto 1 step 2
```

# Arrays

# São objectos da classe Array

# Exemplos de criação

```
val tab1 = arrayOf(1,2,3,4,5)
```

```
val tab2 = arrayOf(1,2,3,4.5,6)
```

```
val tab3 = Array(5, { it * it } )
```

```
val tab4 = Array(4) { i -> (i * 10).toString() }
```

# Exemplo de utilização

```
tab1[1] = tab1[0] + 1
```

# Controlo de fluxo: if

- # Funciona de forma similar ao habitual, mas é uma expressão
- # Consequência: não existe o operador `a?b:c` porque não é necessário.
- # Quando usado como expressão o `else` é obrigatório

```
// Traditional usage
var max = a
if (a < b) max = b

// With else
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

// As expression
val max = if (a > b) a else b
```

# Controlo de fluxo: when

# A estrutura de controlo when possui funcionalidades semelhantes ao switch de outras linguagens embora permita outro tipo de flexibilidade na análise de casos

- # É possível usar expressões nos casos
- # Não é necessário usar break entre cada caso
- # Se nenhum dos casos incluir o valor em análise então é executado o código correspondente ao caso 'else'
- # Tal como o if, o when é uma expressão

```
when (x) {  
    1 -> print("x == 1")  
    2 -> print("x == 2")  
    else -> { // Note the block  
        print("x is neither 1 nor 2")  
    }  
}  
  
when (x) {  
    0, 1 -> print("x == 0 or x == 1")  
    else -> print("otherwise")  
}  
  
when (x) {  
    parseInt(s) -> print("s encodes x")  
    else -> print("s does not encode x")  
}  
  
when (x) {  
    in 1..10 -> print("x is in the range")  
    in validNumbers -> print("x is valid")  
    !in 10..20 -> print("x is outside the range")  
    else -> print("none of the above")  
}  
  
when {  
    x.isOdd() -> print("x is odd")  
    y.isEven() -> print("y is even")  
    else -> print("x+y is even.")  
}
```



yole JetBrains Team  
May '17  
We have a tentative plan to support the continue keyword in when statements to support fallthrough.  
It's not scheduled for any specific future version of Kotlin, though.

# Controlo de fluxo: ciclos

## # for

# A estrutura for existente  
em *kotlin* é a  
correspondente ao  
*foreach* de outras  
línguagens

```
for (item in collection) print(item)
```

```
for (i in 1..3) {  
    println(i)  
}  
for (i in 6 downTo 0 step 2) {  
    println(i)  
}
```

```
for (i in array.indices) {  
    println(array[i])  
}
```

```
for ((index, value) in array.withIndex()) {  
    println("the element at $index is $value")  
}
```

```
while (x > 0) {  
    x--  
}  
  
do {  
    val y = retrieveData()  
} while (y != null) // y is visible here!
```

# break, continue e return

# Funcionam de forma similar ao Java mas...

```
loop@ for (i in 1..100) {  
    for (j in 1..100) {  
        if (...) break@loop  
    }  
}
```

)

```
fun foo() {  
    listOf(1, 2, 3, 4, 5).forEach {  
        if (it == 3) return // non-local return directly to the caller of foo()  
        print(it)  
    }  
    println("this point is unreachable")  
}
```

)

```
fun foo() {  
    listOf(1, 2, 3, 4, 5).forEach lit@{  
        if (it == 3) return@lit // local return to the caller of the lambda.  
        print(it)  
    }  
    print(" done with explicit label")  
}
```

# Funções

- # Definidas com o auxílio da palavra fun

```
fun double(x: Int): Int {  
    return 2 * x  
}
```

- # Ao contrário do Java, podem existir funções globais, não encapsulados em qualquer classe

- # Exemplo:

- # função main – primeira função a ser executada

```
fun main() {  
    . . .  
}
```

# Funções

- # As funções podem ter vários parâmetros, os quais podem assumir valores por omissão

```
fun reformat(  
    str: String,  
    normalizeCase: Boolean = true,  
    upperCaseFirstLetter: Boolean = true,  
    divideByCamelHumps: Boolean = false,  
    wordSeparator: Char = ' ',  
) {  
    /*...*/  
}
```

```
reformat('This is a long String!')
```

```
reformat('This is a short String!', upperCaseFirstLetter = false, wordSeparator = '_')
```

```
fun double(x: Int): Int = x * 2
```

```
fun <T> asList(vararg ts: T): List<T> {  
    val result = ArrayList<T>()  
    for (t in ts) // ts is an Array  
        result.add(t)  
    return result  
}
```

# Mais funções e Lambda functions

```
infix fun Int.shl(x: Int): Int { ... }
```

```
// calling the function using the infix notation  
1 shl 2
```

```
// is the same as  
1.shl(2)
```

```
fun dfs(graph: Graph) {  
    val visited = HashSet<Vertex>()  
    fun dfs(current: Vertex) {  
        if (!visited.add(current)) return  
        for (v in current.neighbors)  
            dfs(v)  
    }  
  
    dfs(graph.vertices[0])  
}
```

```
// Parameter types in a lambda are optional if they can be inferred:  
val joinedToString = items.fold("Elements:", { acc, i -> acc + " " + i })  
  
ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```

# Scope functions

Function	Object reference	Return value	Is extension function
let	it	Lambda result	Yes
run	this	Lambda result	Yes
run	-	Lambda result	No: called without the context object
with	this	Lambda result	No: takes the context object as an argument.
apply	this	Context object	Yes
also	it	Context object	Yes

```
fun main() {
    val str = "Hello"
    // this
    str.run {
        println("The receiver string length: $length")
        //println("The receiver string length: ${this.length}")
    }

    // it
    str.let {
        println("The receiver string's length is ${it.length}")
    }
}
```

```
val numberList = mutableListOf<Double>()
numberList.also { println("Populating the list") }
    .apply {
        add(2.71)
        add(3.14)
        add(1.0)
    }
    .also { println("Sorting the list") }
    .sort()
```

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    println("'with' is called with argument $this")
    println("It contains $size elements")
}
```

# Classes e construtores principais

- # As classes declaram-se com a palavra-chave class
- # Na declaração da classe poderão ser indicados parâmetros usados na criação das instâncias
  - # Esta definição opcional corresponde à definição do construtor principal da classe
  - # Definem variáveis da classe
  - # Poderá ser definido código a ser executado através de blocos init
    - # Podem existir vários e serão executados por ordem

```
class InitOrderDemo(name: String) {  
    val firstProperty = "First property: $name".also(::println)  
  
    init {  
        println("First initializer block that prints ${name}")  
    }  
  
    val secondProperty = "Second property: ${name.length}".also(::println)  
  
    init {  
        println("Second initializer block that prints ${name.length}")  
    }  
}
```

# Construtores secundários

- # As classes podem ter 0 ou mais construtores secundários
  - # Definidos com a palavra constructor
  - # Deverão chamar o constructor principal caso este exista

```
class Person {  
    var children: MutableList<Person> = mutableListOf<>()  
    constructor(parent: Person) {  
        parent.children.add(this)  
    }  
}  
  
class Person(val name: String) {  
    var children: MutableList<Person> = mutableListOf<>()  
    constructor(name: String, parent: Person) : this(name) {  
        parent.children.add(this)  
    }  
}
```

# Mais sobre classes

# Podem existir *nested classes*, *inner class* e *anonymous inner classes*

```
class Outer {  
    private val bar: Int = 1  
    class Nested {  
        fun foo() = 2  
    }  
}  
  
val demo = Outer.Nested().foo() // == 2
```

```
class Outer {  
    private val bar: Int = 1  
    inner class Inner {  
        fun foo() = bar  
    }  
}  
  
val demo = Outer().Inner().foo() // == 1
```

# As instâncias dos objectos são criadas sem usar a palavra-chave new típica de outras linguagens

# Propriedades, getters e setters

- # Propriedades definidas com var ou val
- # Devem ser todas iniciadas por omissão ou serem iniciadas nos contrutores

```
class Address {  
    var name: String = "Holmes, Sherlock"  
    var street: String = "Baker"  
    var city: String = "London"  
    var state: String? = null  
    var zip: String = "123456"  
}
```

## # Getters e Setters

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]  
[<getter>]  
[<setter>]  
  
var stringRepresentation: String  
get() = this.toString()  
set(value) {  
    setDataFromString(value) // parses the string  
}
```

- # Pode-se usar a keyword field para referir o próprio valor

# Herança

- # As classes podem herdar as características de outras classes e redefinir comportamentos
- # Todas as classes em Kotlin possuem uma superclasse em comum designada por Any
  - # A classe Any possui 3 métodos já conhecidos do Java e que podem ser redefinidos: equals, hashCode e toString
- # Para que uma classe possa ser herdada por outra tem que incluir a palavra **open** na sua definição
  - # Os métodos que podem ser redefinidos também têm que possuir a etiqueta **open**

# Herança

- # A sintaxe para herdar as características numa nova classe é
  - # class NovaClasse : ClasseBase() { ... }
- # A redefinição de métodos deve ser explícita, indicando a palavra chave **override** na sua definição
- # O acesso a métodos da classe base é realizada com auxílio da palavra **super**
  - # A própria instância é designada por **this**

# Herança

```
open class Shape {  
    open fun draw() { /*...*/ }  
    fun fill() { /*...*/ }  
}  
  
class Circle() : Shape() {  
    override fun draw() { /*...*/ }  
}  
  
open class Rectangle() : Shape() {  
    final override fun draw() { /*...*/ }  
}
```

# Herança: mais exemplos

```
open class Rectangle {  
    open fun draw() { /* ... */ }  
}  
  
interface Polygon {  
    fun draw() { /* ... */ } // interface members are 'open' by default  
}  
  
class Square() : Rectangle(), Polygon {  
    // The compiler requires draw() to be overridden:  
    override fun draw() {  
        super<Rectangle>.draw() // call to Rectangle.draw()  
        super<Polygon>.draw() // call to Polygon.draw()  
    }  
}
```

```
open class Polygon {  
    open fun draw() {}  
}
```

```
abstract class Rectangle : Polygon() {  
    abstract override fun draw()  
}
```

# Interfaces

```
interface MyInterface {  
    fun bar()  
    fun foo() {  
        // optional body  
    }  
}  
  
class Child : MyInterface {  
    override fun bar() {  
        // body  
    }  
}
```

```
interface MyInterface {  
    val prop: Int // abstract  
  
    val propertyWithImplementation: String  
        get() = "foo"  
  
    fun foo() {  
        print(prop)  
    }  
  
}  
  
class Child : MyInterface {  
    override val prop: Int = 29  
}
```

```
interface A {  
    fun foo() { print("A") }  
    fun bar()  
}  
  
interface B {  
    fun foo() { print("B") }  
    fun bar() { print("bar") }  
}  
  
class C : A {  
    override fun bar() { print("bar") }  
}  
  
class D : A, B {  
    override fun foo() {  
        super<A>.foo()  
        super<B>.foo()  
    }  
  
    override fun bar() {  
        super<B>.bar()  
    }  
}
```

# Outros conceitos

- # enums
- # extensions
- # data classes
- # sealed classes
- # object
  - # singleton
- # companion objects
  - # Permite obter as funcionalidades associados ao static do Java
- # coroutines
- # collections
  - # List, Set, Map, ...
- # delegação
- # generics
- # exceções

```
enum class Direction {  
    NORTH, SOUTH, WEST, EAST  
}  
  
enum class Color(val rgb: Int) {  
    RED(0xFF0000),  
    GREEN(0x00FF00),  
    BLUE(0x0000FF)  
}  
  
class Example {  
    fun printFunctionType() { println("Class method") }  
}  
  
fun Example.printFunctionType(i: Int) { println("Extension function") }  
  
Example().printFunctionType(1)
```

```
class MyClass {  
    companion object Factory {  
        fun create(): MyClass = MyClass()  
    }  
}  
  
val instance = MyClass.create()
```

```
val numbersSet = setOf("one", "two", "three", "four")  
val emptySet = mutableSetOf<String>()
```

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)  
  
data class User(val name: String, val age: Int)
```