estt.ipt
Escola Superior
de Tecnologia de Tomar
Instituto Politécnico de Tomar

Kotlin syntax

(Sintaxe da Linguagem Kotlin)

José Casimiro Pereira

What is Kotlin used for?

- Mobile applications (especially **Android** apps)
- Web development
- Server-side applications
- Data science
- etc., etc.

Why Use Kotlin?

- Kotlin is fully compatible with Java
- Kotlin works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- Kotlin is concise and safe
- Kotlin is easy to learn, especially if you already know Java
- Kotlin is free to use
- Big community/support

The Kotlin Programming Language

- Developed by JetBrains in 2011, but released only in 2016
- In 2017 Google allows the development of mobile applications in Kotlin. And, in 2019 it becomes the default language, replacing Java.
- It is a multiplatform language, it compiles to Java Byte Code, and at least needs a Java Virtual Machine to run the compiled programs.

The Kotlin Programming Language

- Allows programming according
  - the procedural paradigm
  - the object-oriented programming paradigm
- As said, it allows interoperability with Java
- Those who already know how to program in Java quickly learn Kotlin

Variables and Constants

- **var** is reserved for defining a *variable*
- **val** is a reserved word to define a *constant*
- You don't need to specify the type of variable (or return function)
  unless it is not initialized

Variables and Constants

```
var variableName = value
val variableName = value

var name = "John"
val birthyear = 1975

var name: String = "John"
val birthyear: Int = 1975

var name: String
val birthyear: Int
```


Variable Names

The general rule for Kotlin variables are:

- Names can contain letters, digits, underscores, and dollar signs
- Names should start with a letter
- Names can also begin with $ and _
- Names are case sensitive (`myVar` and `myvar` are different variables)
- Names should start with a lowercase letter and cannot contain whitespace. Use the **camelCase** notation.
- Reserved words (like Kotlin keywords, such as *`var`* or *`String`*) cannot be used as names

Data types

- **Numbers**:
  - **Byte**: -128 to 127
  - **Short**: -32,768 to 32,767
  - **Int**: -2,147,483,648 to 2,147,483,647
  - **Long**: -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807
  - **Float**: precision to 6 decimal places
  - **Double**: precision to 15 decimal places

Data types

- **Logic**:
  - **Boolean**: true or false
- **Text**:
  - **Char**: a single character
  - **String**: a chain of characters

Data types

```
val myNum = 5           // Int
val myDoubleNum = 5.99 // Double
val myLetter = 'D'      // Char
val myBoolean = true    // Boolean
val myText = "Hello"    // String
```

```
val myNum: Int = 5              // Int
val myDoubleNum: Double = 5.99 // Double
val myLetter: Char = 'D'        // Char
val myBoolean: Boolean = true  // Boolean
val myText: String = "Hello"    // String
```


Arithmetic Operators

Operator	Name	Example
+	addition	x + y
–	subtraction	x – y
*	multiplication	x * y
/	division	x / y
%	rest of division	x % y
++	increment	++x
– –	decrement	– –x

Arithmetic and Assignment Operators

Operator	Example	The same as
=	x = 5	x = 5
+=	x += 3	x = x + 3
–=	x –= 3	x = x – 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

Comparison Operators

Operator	Name	Example
==	equal	x == y
!=	not equal	x != y
>	bigger than	x > y
<	less than	x < y
>=	greater than or equal to	x >= y
<=	less than or equal to	x <= y

Logical Operators

Operator	Name
&&	logical AND
||	logical OR
!	negation

Strings

- String characters can be accessed with the **[ ]** operator, like if they are an array
- ***length*** property displays the length of the string
- A string is treated like an object , so it has several methods to expose its data or behavior

Strings

```
var txt = "Hello World"
println(txt[0]) // H
println(txt[2]) // l

println("Tamanho:" + txt.length)

println(txt.toUpperCase())
println(txt.toLowerCase())
```


17

Strings

- Relational operators cannot be used to compare strings and other objects .
  You must use the **compareTo()** method

```
var txt1 = "Hello World"
var txt2 = "Hello World"
println(txt1.compareTo(txt2))
```


18

Strings

- Kotlin allow us to reference variables in a string

```
var firstName = “John”
var lastName = “Doe”
println (“My name is “ + firstName + ” “ + lastName)
println (“My name is $firstName $lastName”)
```


Selection statements

- **if**, **if** + **else**, **if** + **else** + **if**

```
if (20 > 18) {
    println("20 é maior do que 18")
}
```

```
val time = 21
if (time < 20) {
    println("Bom dia.")
} else {
    println("Boa noite.")
}
```

```
val time = 22
if (time < 12) {
    println("Bom dia.")
} else if (time < 20) {
    println("Boa tarde.")
} else {
    println("Boa noite.")
}
```

Selection statements

- **when** , is a structure similar to *switch*, but more flexible

```
when (x) {
    2 -> println("This is 2")
    3,4,5,6,7,8 -> println("This is 3,4,5,6,7 or 8")
    in 9..15 -> println("This is 9,10,11,12,13,14 or 15")
    in 20..24 -> {
        println("This is 20")
        println("or is 21")
        println("or is 22")
        println("or is 23")
        println("or is 24")
}
    else -> println("invalid number")
}
```


Selection statements in *expressions*

- When you use the **IF** in an *expression*, you MUST specify the **ELSE** clause

```
val time = 10
val greeting = if (time < 20) {
    "Bom Dia."
} else {
    "Boa Tarde."
}
println(greeting)
```

Selection statements in *expressions*

```
val day = 4
val result = when (day) {
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
    7 -> "Sunday"
    else -> "Invalid day."
}
println(result)
```


Iteration statements

- We have the **while** , the **do while** and the **for** loop
- The jump statements are available:
  - **break**: jumps out of the loop
  - **continue**: move to the beginning of next iteration of the loop

```
var i = 0
while (i < 5) {
    println(i)
    i++
}
```

```
var i = 0
while (i < 5) {
    if (i % 2 == 0) {
        continue
    }
    println(i)
    i++
}
```

```
var i = 0
do {
    println(i)
    i++
} while (i < 5)
```

Iteration statements

- There is no traditional syntax on **for** statement

```
for(initialization, exit condition, increment){
    statements...
}
```

- **for** is mainly used to go through an interval or progression

```
for (chars in 'a'..'x') {
    println(chars)
}

for (nums in 5..15) {
    println(nums)
}
```

```
for (nums in 15 downTo 5) {
    println(nums)
}

for (nums in 5..15 step 2) {
    println(nums)
}

for (nums in 5 until 15) {
    println(nums)
}
```

The 15 is not shown

Arrays

- The creation of arrays has some peculiarities
- Access to each element is done by the operator **[ ]**
- An array has a **`size`** attribute,
  with the number of array elements
- There are new operators to search arrays in a simple and fast way

Arrays

```
val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
println(cars[0])

cars[0] = "Opel"
println(cars[0])

println(cars.size)
```

```
if ("Volvo" in cars) {
    println("It exists!")
} else {
    println("It does not exist.")
}
```

```
for (x in cars) {
    println(x)
}
```


Arrays

- Creating an array with a defined size, implies initializing all its elements through a lambda function

```
// fills an array with 10 zeros
val numbers = Array (10, { i -> 0 } )

// fills an array with 0,1,2,3,4,5,6,7,8,9
val numbers = Array (10, { i -> i } )

// fills an array with 0,2,4,6,8,10,12,14,16,18
val numbers = Array (10, { i -> i*2 } )
```

Functions

- **fun** is the reserved word to define a function
- The function must have a **name (**and parameters**)**
- Function code is written inside braces **{ }**
- As you have already seen, the semicolon **;** at the end of the instructions is optional

```
fun myFunction() {
    println("I just got executed!")
}
```

```
fun main() {
    myFunction()
    myFunction()
    myFunction()
}
```


The **main** function

- First function to be executed

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

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

Functions

- It is possible to define the *default* values for parameters

```
fun myFunction(fname: String = "John", age: Int = 35) {
    println(fname + " is " + age)
}
fun main() {
    myFunction("John", 35)
    myFunction("Jane")
    myFunction()
}
```


Functions

- If a function returns something, you MUST specify what

```
fun myFunction ( x : Int , y : Int ) : Int {
    return ( x + y )
}

fun myFunction2 ( x : Int , y : Int ) : Int = x + y


fun main () {
    var result = myFunction ( 3 , 5 )
    println ( result )
}
```

Classes

- Class attributes are public by default

```
class Car {
    var brand = ""
    var model = ""
    var year  = 0
}
```


Classes

```
fun main () {
// Create a c1 object of the Car class
    val c1 = car ()

    // Access the properties and add some values to it
    c1 . brand = "Ford"
    c1 . model = "Mustang"
    c1 . year  = 1969

    println ( c1.brand )  // Output Ford
    println ( c1.model )  // Output Mustang
    println ( c1.year )   // Output 1969
 }
```

Classes

- In Kotlin, you do not need to define *constructors*

```
class Car (
  var brand : String ,
  var model : String ,
  var year  : Int
)

fun main () {
    val c1 = Car ( "Ford" , "Mustang" , 1969 )
}
```


Classes

```
class Car (
  var brand : String = "BMW" ,
  var model : String = "series5" ,
  var year  : Int = 2000 )

fun main () {
    val c1 = Car ( "Ford" , "Mustang" , 1969 )
    val c2 = Car ( "Ford" , "Mustang" )
    val c3 = Car ( "Ford" )
    value c4 = car ()
}
```

Classes

- This does not mean that you cannot define a *constructor*

```
class Car {
    var brand = "" ; var model = "" ; var year = 0 ;

   constructor ( _brand: String, _model: String ) {
         this ( _brand, _model, 2000 )
   }

   constructor ( _brand: String, _model: String, _ year:int){
      brand = _brand
      model = _model
      year = _year
   }
}
```


Classes

- **Methods** follow the same syntax as functions. When placed inside a class, they automatically access its attributes.

```
class Car ( var brand : String ,
   var model : String , var year : Int ) {

   fun drive () {
       println ( " Wrooom !" )
   }

   fun speed ( maxSpeed : Int ) {
      println ( "Max speed of $model is $maxSpeed km" )
   }
}
```

Classes

- **Methods** follow the same syntax as functions. When placed inside a class, they automatically access its attributes.

```
fun main () {
    val c1 = Car ( "Ford" , "Mustang" , 1969 )

    // Call the functions
    c1 . drive ()
    c1 . speed ( 200 )
}
```

```
class Car ( var brand : String ,
    var model : String , var year : Int ) {

    fun drive () {
        println ( " Wrooom !" )
    }

    fun speed ( maxSpeed : Int ) {
        println("Max speed of $model is $maxSpeed km")
    }
}
```


Classes

- Kotlin defines classes as **final** by default. This means that it is impossible to inherit the class.
- To use inheritance, you must write the reserved word **open** before the *name* of the class.

Classes

```
// superclass
open class MyParentClass {
    value x = 5
}

// subclass
class MyChildClass : MyParentClass () {
    fun myFunction () {
        println ( x ) // x is now inherited
                      //   from the superclass
    }
}
                          fun main () {
                              val myObj = MyChildClass ()
                              myObj . myFunction ()
                          }
```


Bibliography

- https://www.w3schools.com/kotlin/
- https://www.programiz.com/kotlin-programming
- https://kotlinlang.org/

- Special thanks to Professor Paulo Santos