

# **Scala: breve introdução**

**Alexandre Mota**

**a cm @ cin.ufpe.br**



UNIVERSIDADE  
FEDERAL  
DE PERNAMBUCO

---

**CIn.ufpe.br**

# Sobre Scala

- Criada em 2001 por Martin Odersky
- Scala (Scalable language) é uma linguagem de programação de propósito geral (multiparadigma), projetada para expressar padrões de programação comuns de uma forma concisa, elegante e type-safe
- Incorpora recursos de linguagens orientadas a objetos e funcionais
- É plenamente interoperável com Java
- Apesar de recente, conseguiu ser adotada pela Twitter e Foursquare

# Instalação

- Tradicionalmente, Scala pode ser obtida aqui
  - <http://www.scala-lang.org/download/>
- Mas para o propósito da disciplina, usaremos uma instalação em Eclipse já embutindo o CSO que encontra-se aqui
  - <https://gist.github.com/AbigailBuccaneer/1704860>

# **SCALA POR EXEMPLOS SIMPLES**

# Hello World

- **/\*\* O 1º programa padrão \*/**  
**object HelloWorld {**  
    **def main(args: Array[String]) {**  
        **println("Hello, World!")**  
    **}**  
**}**
- Armazene em arquivo chamado **HelloWorld.scala**
- Compile usando **scalac HelloWorld.scala** (ou de uma IDE)
- Execute com **scala HelloWorld** (ou de uma IDE)
- O código acima cria um único objeto
  - Tudo no objeto é semelhante ao **static** de Java
  - Apesar de Scala não ter **static**
- Scala tem classes (Veremos adiante)

# Comentários

- **// e /\*...\*/ como em Java e C**
  - **/\*\* Scaladoc comments are similar to Javadoc \* comments. As in Javadoc, the first sentence \* is a summary sentence, and additional \* sentences give more detail. However, the \* formatting convention is slightly different, \* and wiki markup is used in preference to \* HTML markup. Most of the same flags \* (@author, etc.) are used. \*/**
- ```
def doNothing = ()
```

# Scala e Java

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._
object FrenchDate {
    def main(args: Array[String]) {
        val now = new Date
        val df = getDateInstance(LONG, Locale.FRANCE)
        println(df format now)
    }
}
```

df.format(now)

# Tudo é um objeto...

- A expressão

- $1 + 2 * 3 / x$

- Pode ser descrita assim

- $1.+(2.*(3./(x)))$

# Funções são objetos

Para passar função

```
object Timer {  
    def oncePerSecond(callback: () => Unit) {  
        while (true) { callback(); Thread sleep 1000 }  
    }  
    def timeFlies() {  
        println("o tempo corre como um raio...")  
    }  
    def main(args: Array[String]) {  
        oncePerSecond(timeFlies)  
    }  
}
```

# Funções anônimas

```
object TimerAnonymous {  
    def oncePerSecond(callback: () => Unit) {  
        while (true) { callback(); Thread sleep 1000 }  
    }  
    def main(args: Array[String]) {  
        oncePerSecond(() =>  
            println("o tempo corre como um raio..."))  
    }  
}
```

# Definição recursiva do while

```
def whileLoop(condition: => Boolean)(command: => Unit) {  
    if (condition) {  
        command; whileLoop(condition)(command)  
    } else ()  
}
```

# Classes

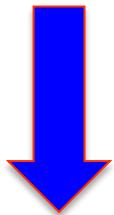
```
class Complex(real: Double, imaginary: Double) {  
    def re() = real  
    def im() = imaginary  
}
```



```
new Complex(1.5, 2.3)
```

# Classes

```
class Person(private var _name: String) {  
    def name = _name                                // accessor  
    def name_=(aName: String) { _name = aName } // mutator  
}
```



```
val p = new Person("Jonathan")  
p.name = "Jony"    // setter  
println(p.name)   // getter
```

# Herança e polimorfismo

```
class Complex(real: Double, imaginary: Double) {  
    def re = real  
    def im = imaginary  
    override def toString() =  
        "" + re + (if (im < 0) "" else "+") + im + "i"  
}
```

# **ALGUNS DETALHES TÉCNICOS**

# Tipos

## ■ A hierarquia de tipos é um reticulado (não uma árvore)

- Isto é, tem um elemento “bottom” e um “top”

## ■ Any

- AnyVal
  - Boolean, Char, Byte, Short, Int, Long, Float, Double
    - Scala não tem primitivos—todos criam objetos
  - Unit (tem apenas um único valor: () )
    - Unit é o valor padrão de funções que “não têm retorno” (ex. `Println`)

# Tipos

## ■ Any

- AnyRef (corresponde a Object de Java)
  - Todos os tipos de referência Java, por exemplo, String
  - ScalaObject
    - Todos os tipos de referências Scala, incluindo Array e List
- Null (bottom de todos os objetos AnyRef)

## ■ Nothing (bottom de Any)

# Declaração de variáveis e valores

- A sintaxe é

```
var name: type = value // declara uma variável
```

```
val name: type = value // declara um valor
```

- Valores são **imutáveis**: eles não podem ser alterados (paradigma funcional)
- O fragmento `:` **type** em geral pode ficar ausente—o tipo é inferido do valor
- O fragmento `=` **value** sempre deve ser fornecido (para a inferência funcionar)
- Isto também funciona: `var x: Double = 5`

# Declaração de variáveis e valores

- Identificadores são como em Java, da mesma forma como as regras de escopo
- Operadores aritméticos, de comparação, e lógicos também acompanham os de Java
- Indentação é feita com 2 espaços em branco (Comandos não usam ; como terminação e a indentação serve como escopos {} de Java )

# “Comandos”

- “Comandos” em Scala na verdade agem como “expressões” porque sempre possuem valor associado
  - Por exemplo, o valor de `a = 5` é `5`
- O valor de muitos comandos é simplesmente `()`
  - `()` é um valor do tipo `Unit`
  - `()` é o único valor do tipo `Unit`
- O valor de um bloco, `{...}`, é o último valor computado no bloco
- Um comando termina com o fim de linha (e não com `;`) exceto se estiver de fato incompleto
  - Por exemplo, `x = 3 * (2 * y +` está obviamente incompleto
  - Uma vez que Scala permite ausência de muita terminação desnecessária, as vezes uma linha que se pode pensar estar completa, está incompleta (e vice versa)
- Você pode terminar comandos com `;`, mas não é boa prática em Scala

# Tipos de comandos frequentes

## ■ Os mais comuns são:

- **variável** = **expressão** // também **+<sub>=</sub>**, **\*<sub>=</sub>**, etc.
  - O valor do comando é o valor da variável
- **while (condição) { comandos }**
  - O valor do comando é **()**
- **do { comandos } while (condição)**
  - O valor é **()**
- **if (condição) { comandos }**
  - O valor é o do último comando executado
- **if (condição) { comandos1 } else { comandos2 }**
  - Se **else** omitido e a condição for false, o valor do **if** será **()**

# O comando for

- O comando **for** de Scala é mais poderoso que o de Java
  - Assim, ele é usado com mais frequência que outros tipos de laços
- Alguns exemplos abaixo...
- **for (i <- 1 to 10) { println(i) }**
  - Mostra os números de 1 a 10
- **for (i <- 1 until 10) { println(i) }**
  - Mostra os números de 1 a 9
- **for (x <- 0 until myArray.length) {  
  println(myArray(x)) }**
  - Mostra todos os valores de myArray

# O comando for

- **for (x <- myArray) { println(x) }**
  - Mostra todos os valores de myArray
- **for (x <- myArray  
if x % 2 == 0) { println(x) }**
  - Mostra todos os números pares existentes em myArray

# Arrays

- **Arrays em Scala são tipos parametrizados**
  - `Array[String]` é um Array de Strings, onde `String` é um *parâmetro de tipo*
  - Em Java chamaríamos isto de “tipo genérico”
- **Quando valores iniciais não são dados, `new` é requerido, bem como um tipo explícito:**
  - `val ary = new Array[Int](5)`
- **Quando valores iniciais são dados, `new` não é permitido:**
  - `val ary2 = Array(3, 1, 4, 1, 6)`

# Arrays

- **Arrays em Scala são apenas um outro tipo de objeto; eles não têm sintaxe especial**
- **Listas em Scala são mais úteis e usadas que Arrays**
  - `val list1 = List(3, 1, 4, 1, 6)`
  - `val list2 = List[Int]() // Uma lista vazia deve ter um tipo explícito`

# Operações simples sobre List

## ■ Por padrão, Lists, como Strings, são imutáveis

- Operações sobre uma List imutável retorna uma nova List

## ■ Operações básicas:

- *lista.head* (ou *lista head*) retorna o primeiro elemento da lista
- *lista.tail* (ou *lista tail*) retorna uma lista sem o primeiro elemento
- *lista(i)* retorna o *i*<sup>ésimo</sup> elemento (iniciando em 0) da lista
- *lista(i) = valor* é **ilégal** (imutável, lembra?)
- *valor :: lista* retorna uma lista com *valor* adicionado à frente
- *lista1 :: lista2* concatena duas listas
- *lista.contains(valor)* (ou *lista contains valor*) testa se *valor* existe em *lista*

# Operações simples sobre List

- Uma operação sobre uma List pode retornar uma List de um tipo diferente
  - scala> "abc" :: List(1, 2, 3)  
res22: List[Any] = List(abc, 1, 2, 3)
- Há mais de 150 operações pré-def sobre Lists--use a API!

# T-uplas

## ■ Scala tem t-uplas (t no máximo 22)

- scala> val t = Tuple3(3, "abc", 5.5)  
t: (Int, java.lang.String, Double) = (3,abc,5.5)
- scala> val tt = (3, "abc", 5.5)  
tt: (Int, java.lang.String, Double) = (3,abc,5.5)

## ■ T-uplas são referenciadas *iniciando em 1, usando \_1, \_2, ...*

- scala> t.\_1  
res28: Int = 3
- t \_1 também funciona (o . é opcional)

## ■ T-uplas, como listas, são imutáveis

## ■ T-uplas são alternativa para retornar mais de um valor

# Mapas

- **scala> val m = Map("apple" -> "red", "banana" -> "yellow")**  
**m:**  
**scala.collection.immutable.Map[java.lang.String,java.lang.String] = Map((apple,red), (banana,yellow))**
  - Note que um Map é de fato uma lista de t-uplas
  - O **->** é usado como uma sintaxe mais legível
- **scala> m("banana")**  
**res2: java.lang.String = yellow**
- **scala> m contains "apple"**  
**res3: Boolean = true**
- **scala> m("cherry")**  
**java.util.NoSuchElementException: key not found: cherry**

# Simples definições de funções

- **def isEven(n: Int) = {**  
    **val m = n % 2**  
    **m == 0**  
**}**

- O resultado é o último valor (neste caso, um Booleano)

- **def isEven(n: Int) = n % 2 == 0**

- O resultado é expressão simples e então não precisa das chaves

- **def countTo(n: Int) {**  
    **for (i <- 1 to 10) { println(i) }**  
**}**

- Não é bom estilo omitir o **=** quando o resultado for **()**
  - Se o **=** for omitido, o resultado será **()**

- **def half(n: Int): Double = n / 2**  
    — O tipo de retorno pode ser dado explicitamente  
    — Neste exemplo, **half(7)** retornará **3.5** (!)

- **def half(n: Int): Int = return n / 2**

- Se **return** for usado, deve-se colocar o tipo explicitamente

# Funções = objetos de 1a classe

- Funções são valores (como inteiros, etc.) e podem ser atribuídos a variáveis, passados a e retornados de funções
- Toda vez que o símbolo => surgir, trata-se de função
- Exemplo (atribuição de função a variável **foo**):
  - scala> val foo = (x: Int) => if (x % 2 == 0) x / 2 else 3 \* x + 1  
foo: (Int) => Int = <function1>

```
scala> foo(7)  
res28: Int = 22
```

- A sintaxe básica de uma função é lista\_de\_parâmetros => corpo\_da\_função
- Neste exemplo, **foreach** é uma função que usa função como parâmetro:
  - myList.foreach(i => println(2 \* i))

# Funções como parâmetros

- Ao definir função, deve-se especificar os tipos de cada um de seus parâmetros
- Então, tem-se as opções:
  - (*tipo1, tipo2, ..., tipoN*) => *tipo\_de\_retorno*
  - *tipo* => *tipo\_de\_retorno* // Apenas um parâmetro

## ■ Exemplo:

— scala> def doTwice(f: Int => Int, n: Int) = f(f(n))  
doTwice: (f: (Int) => Int, n: Int)Int

```
scala> def collatz(n: Int) = if (n % 2 == 0) n / 2 else 3 * n + 1  
collatz: (n: Int)Int
```

```
scala> doTwice(collatz, 7)  
res2: Int = 11
```

```
scala> doTwice(a => 101 * a, 3)  
res4: Int = 30603
```

# Métodos de alta-ordem sobre listas

## ■ **map** aplica função de 1-parâmetro a todo elemento de lista, retornando nova lista

- scala> def double(n: Int) = 2 \* n  
double: (n: Int)Int
- scala> val ll = List(2, 3, 5, 7, 11)  
ll: List[Int] = List(2, 3, 5, 7, 11)
- scala> ll map double  
res5: List[Int] = List(4, 6, 10, 14, 22)
- scala> ll map (n => 3 \* n)  
res6: List[Int] = List(6, 9, 15, 21, 33)
- scala> ll map (n => n > 5)  
res8: List[Boolean] = List(false, false, false, true, true)

## ■ **filter** aplica teste de 1-parâmetro a todo elemento de lista, retornando uma lista dos elementos que passam no teste

- scala> ll filter(n => n < 5)  
res10: List[Int] = List(2, 3)
- scala> ll filter (\_ < 5) // função abreviada onde parâmetro simples é usado 1-vez  
res11: List[Int] = List(2, 3)

# Casamento de padrão

## ■ Casamento de padrão sobre valores:

```
— today match {  
    case "Saturday" => println("Party! Party! Party!")  
    case "Sunday" => println("Pray....")  
    case day => println(day + " is a workday. :( ")  
}
```

## ■ Casamento de padrões sobre tipos:

```
— something match {  
    case x: Int => println("I'm the integer " + x)  
    case x: String =>  
        println("I'm the String \"\" + x + "\"")  
        println("My length is " + x.length)  
    case _ => println("I don't know what I am! :( ")  
}
```

# O tipo Option

- Scala tem **null** devido ao intercâmbio com Java; não precisa ser usado em outros momentos
- No lugar, usa o tipo **Option**, com valores **Some(valor)** and **None**
  - ```
def max(list: List[Int]) = {  
    if (list.length > 0) {  
        val biggest = (list(0) /: list) { (a, b) => if (a > b) a else b }  
        Some(biggest)  
    } else {  
        None  
    }  
}
```
  - ```
max(myList) match {  
    case Some(x) => println("Maior número é " + x)  
    case None => println("Não há números aqui!!!")  
}
```

# Os métodos require e assert

- **require** e **assert** são métodos que levantam exceção quando seus argumentos são **false**
- **Require** é usados para documentar que algo deve ser válido para o código funcionar corretamente
  - `def sqrt(x: Double) = { require(x >= 0); ... }`
  - **Require** é frequentemente usado no início de um método
- **Assert** é usado para documentar que algo “conhecido” deve ser válido
  - `takeCis700course`  
`assert(languagesIKnow contains "Scala")`
  - **Assert** é frequentemente usado ao final de um bloco de código, para informar o que o código alcançou

# Exceções

- A criação e lançamento de exceções em Scala é como em Java:
  - class RottenEggException extends Exception
  - throw new RottenEggException

- Capturar uma exceção levantada usa casamento de padrão:

```
– try {  
    makeAnOmlet  
} catch {  
    case ex: RottenEggException => println("#$%&%@")  
    case ex: Exception => println("What went wrong?")  
}
```

# **Scala: breve introdução**

**Alexandre Mota**

**a cm @ cin.ufpe.br**



UNIVERSIDADE  
FEDERAL  
DE PERNAMBUCO

---

**CIn.ufpe.br**