Introdução à Linguagem Scala

Carlos Bazilio

carlosbazilio@puro.uff.br

Departamento de Computação
Instituto de Ciência e Tecnologia
Universidade Federal Fluminense

Tópicos Abordados

- Origem e Definições
- Conceitos Básicos
- Funções, Classes e Traits
- Casamento de Padrões
- XML
- Programação Concorrente
- Considerações Finais

Martin Odersky

Professor na EPFL (Suiça)

Primeiro *release* de Scala em 2003

Também conhecido pelo Generic Java

Scala é ...

- Linguagem que mistura os paradigmas Funcional e OO
- Linguagem de tipagem estática
- Linguagem que roda sobre a JVM e interopera com a linguagem Java
- Linguagem com sintaxe flexível e menos verbosa que Java
- Linguagem com construções adequadas para a programação concorrente

Uso comercial

- Adoção pelo Twitter em detrimento a Ruby por questões de desempenho
- Empresa Typesafe criada por Odersky para suporte à linguagem
- Outros usuários: LinkedIn, Twitter, Novell, The Guardian, Xebia, Xerox, FourSquare, Sony, Siemens, Thatcham, OPower, GridGain, AppJet, Reaktor, EDFT

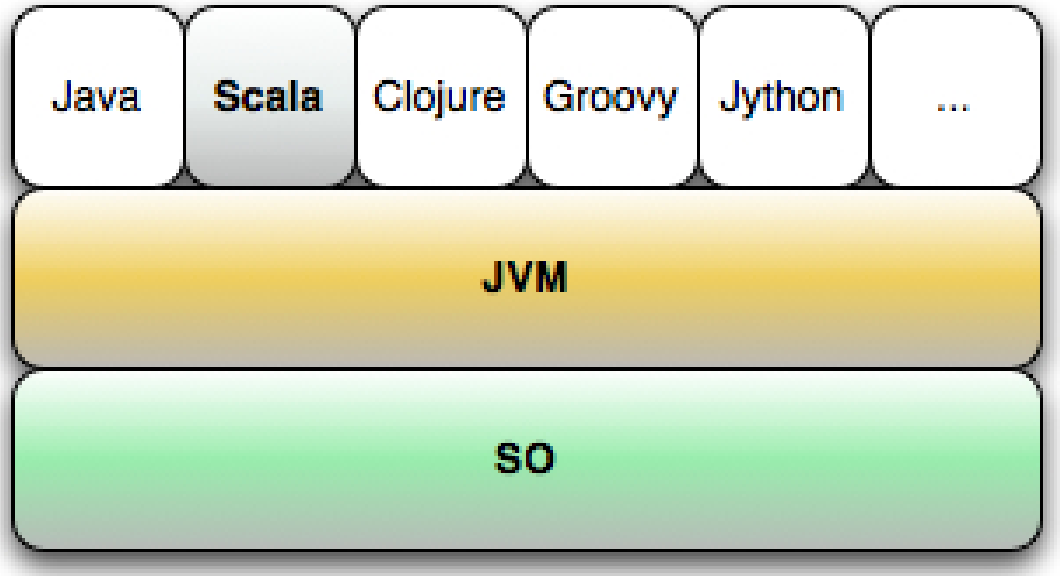
Outras linguagens sobre a JVM

Groovy – Linguagem de Script

JRuby – Implementação Ruby

Jython – Implementação Python

Clojure – Linguagem Funcional similar a LISP



Conceitos Básicos

```
val x = 3 + 4
```

Conceitos Básicos
REPL (read-eval-print loop)

```
> val x = 3 + 4
```

Conceitos Básicos - Imutabilidade

```
> val x = 3 + 4
```

Conceitos Básicos - Imutabilidade

```
> var x = 3 + 4
```

Conceitos Básicos – Inferência de Tipos

```
> val x = 3 + 4
x: Int = 7
```

Conceitos Básicos – Inferência de Tipos

```
> val x : Int = 3 + 4
x: Int = 7
```

Conceitos Básicos – Operadores x Funções

```
> val x = 3.+(4)
x: Int = 7
```

Conceitos Básicos – Delimitadores Opcionais

```
> val x = 3 + 4;
x: Int = 7
```

Funções

Função

```
(x : Int) => x + 1
```

Funções como valores

```
val inc = (x : Int) => x + 1
```

Funções como parâmetros

```
List(1,2).map( (x : Int) => x + 1 )
List(1,2).map( x => x + 1 )
List(1,2).map( _ + 1 )
```

Funções

```
def nomeFunção (par1:Tipo1, .., parn:Tipon):TipoRet = {
    Corpo da Função
}
```

Praticamente todos os elementos acima são opcionais:

- TipoRet pode ser inferido pelo compilador
- A lista de parâmetros é opcional
- O bloco de comandos {...} pode ser substituído por uma constante

Funções

```
def nomeFunção = (par1:Tipo1,..,parn:Tipon):TipoRet => {
    Corpo da Função
}
```

Este formato de definição atribui à função *nomeFunção* uma função anônima (função definida após o símbolo “=”)

Funções – Exemplo

```
def fatorial(n : Int) : Int = {
    var res : Int = 1
    if (n >= 1)
        res = n * fatorial(n - 1)
    res
}
```

Funções – Exemplo

```
def fatorial(n : Int) = {
    var res = 1
    if (n >= 1)
        res = n * fatorial(n - 1)
    res
}
```

Funções – Exemplo

```
def fwhile (cond: => Boolean)(corpo: => Unit) : Unit = {
    if (cond) {
        corpo
        fwhile(cond)(corpo)
    }
}
```

```
var i = 5
fwhile (i > 0) {
    println("Hello nro: " + i + "!!")
    i = i - 1
}
```

A função acima define o comando *while*

Esta possui 2 listas de parâmetros

O tipo Unit representa o tipo sequência de comandos em Scala

=> representa “passagem por nome”

Funções de Alta Ordem

Mapeia, Filtra e Reduz

```
1: scala> List(1, 2, 3).map(x => x + 1)
2: res0: List[Int] = List(2, 3, 4)

3: scala> List("a", "abc", "abcdef") map { x => x.length() }
4: res1: List[Int] = List(1, 3, 6)

5: scala> List("a", "abc", "abcdef")
                filter { x => x.length() >= 3 }
6: res2: List[java.lang.String] = List(abc, abcdef)

7: scala> List("a", "abc", "abcdef") map { x => x.length() }
                reduceLeft { (x, y) => x + y }
8: res3: Int = 10
```

Classes – Exemplo

```
class Empregado(val nome : String, var salario : Double)
```

```
val emp = new Empregado("João da Silva", 1500)
println(emp.nome + ", " + emp.salario)
```

Na declaração de uma classe precisamos explicitar a mutabilidade dos atributos

O construtor da classe que inicializa seus campos é criado automaticamente

Classes – Exemplo

```
class Empregado(val nome : String, var salario : Double) {
    def this (val n : String) = this(n, 0)
}
```

```
val emp = new Empregado("João da Silva")
println(emp.nome + ", " + emp.salario)
```

Método this na classe Empregado representa um construtor alternativo para a classe

Classes – Exemplo

```
class Empregado(val nome : String, var salario : Double = 0)
```

```
val emp = new Empregado("João da Silva")
println(emp.nome + ", " + emp.salario)
```

Campo salario possui valor padrão (*default*)

Métodos e Encapsulamento

```
class Empregado(val nome : String,
        private var salario : Double) {
    def aumentarSalario() = { salario = salario * 1.1 }
    def obterSalario() = salario
}
```

Métodos *aumentarSalario* e *obterSalario* manipulam o atributo privador *salario*

Classes – Herança

```
class Gerente(override val nome : String, private var
salario : Double, var area : String)
extends Empregado(nome, salario)
```

```
val ger = new Gerente("Jose Raimundo", 2000, "Financeiro")
ger.aumentarSalario()
println(ger.nome + ", " + ger.obterSalario + ", " + ger.area)
```

Classe Gerente estende a Empregado

Classes Estáticas

```
object Presidente extends Empregado ("Dilma Rousseff", 5000)
Presidente.aumentarSalario()
println(Presidente.nome + ", " + Presidente.obterSalario)
```

Objeto Presidente representa um *singleton*

Este objeto acima estende a classe Empregado

Classes Genéricas

```
1: scala> List(1, 2, 3)
2: res0: List[Int] = List(1, 2, 3)
3: scala> List("a", "b", "c")
4: res1: List[java.lang.String] = List(a, b, c)
5: scala> List(1, "a", 2, "b")
6: res2: List[Any] = List(1, a, 2, b)
```

- Classe List é um exemplo de classe genérica
- Nos exemplos acima o compilador está inferindo o tipo da lista manipulada

Classes Genéricas

```
1: scala> List[Int](1, 2, 3)
2: res4: List[Int] = List(1, 2, 3)
3: scala> List[String](1, "a", 2, "b")
4: <console>:6: error: type mismatch;
 found   : Int(1)
 required: String
      List[String](1, "a", 2, "b")
5: <console>:6: error: type mismatch;
 found   : Int(2)
 required: String
      List[String](1, "a", 2, "b")
```

Nestes exemplos está sendo informado o tipo de lista que se deseja manipular

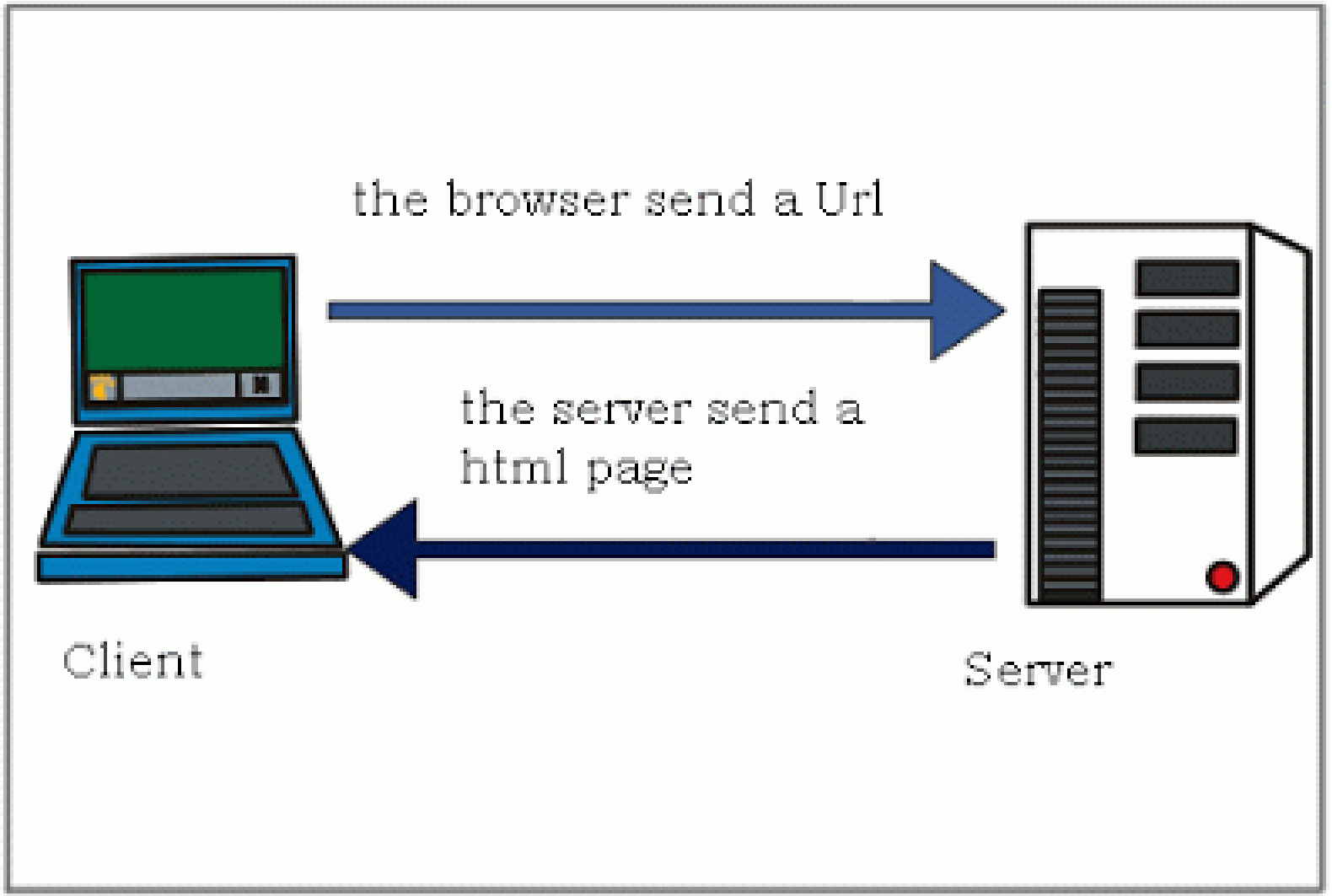
Web Crawler

Um programa que funciona como um robô, o qual varre a internet em busca de conteúdos específicos

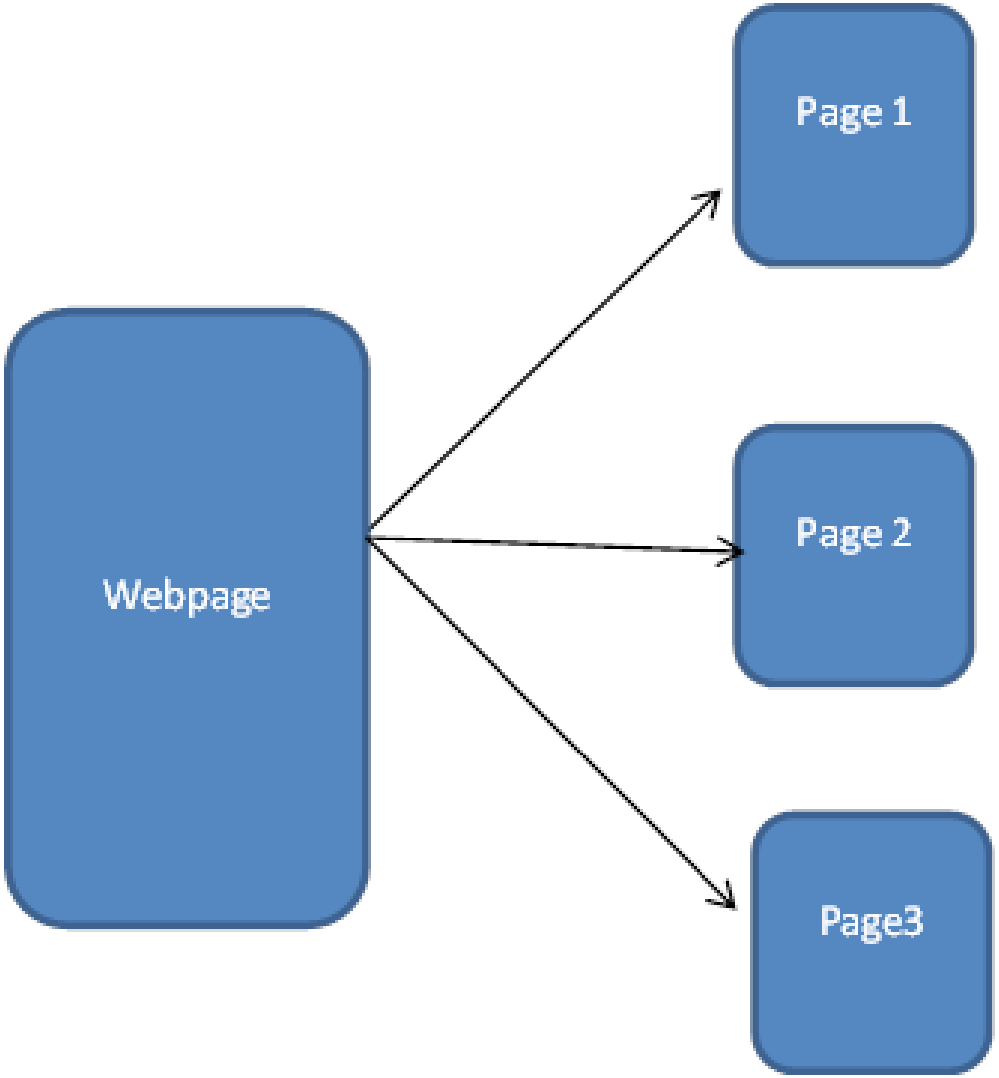
Sua aplicação mais comum é a indexação de páginas web, a qual é usada como referência para os buscadores

Exemplos: PageRank (Google) e Internet Archive (http://archive.org)

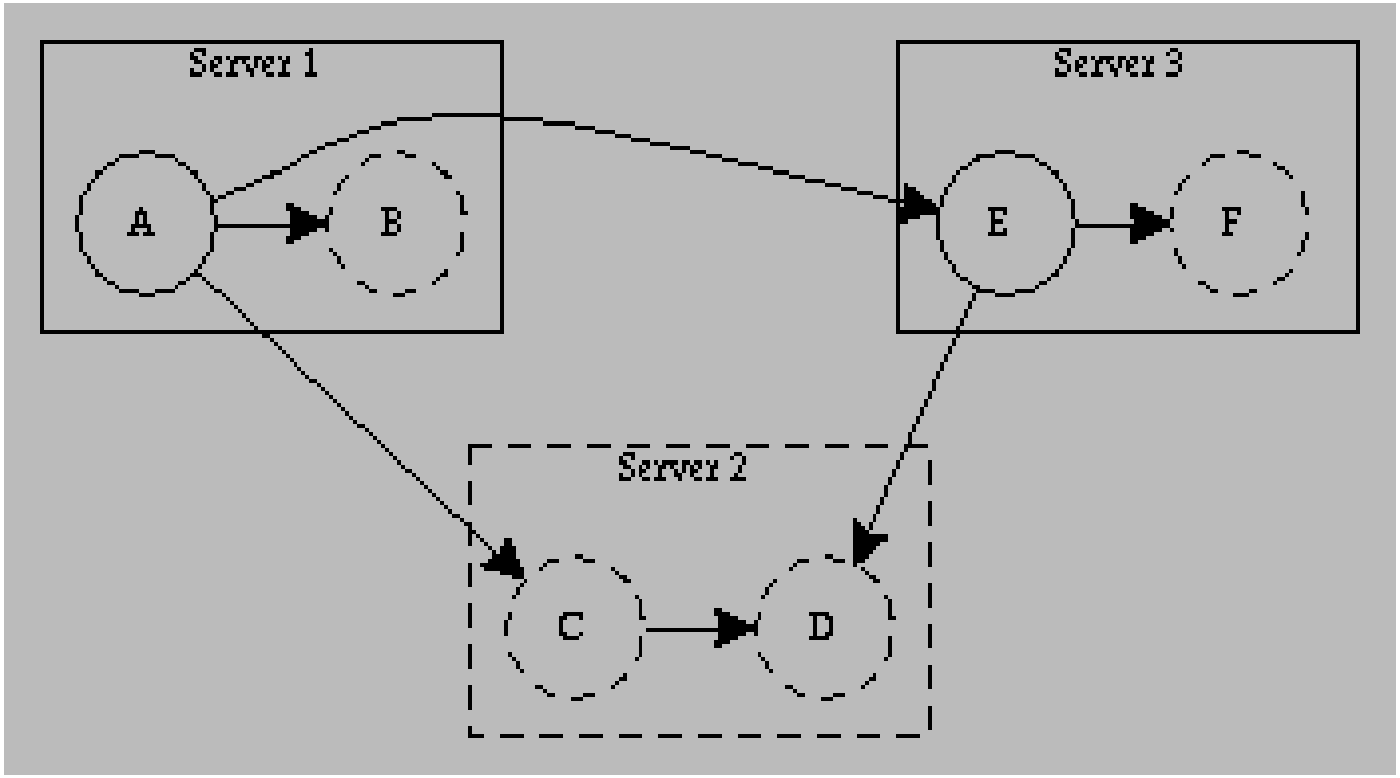
Implementação de um Web Crawler (Java X Scala)



Implementação de um Web Crawler (Java X Scala)



Implementação de um Web Crawler (Java X Scala)



Implementação de um Web Crawler (Java X Scala)

Web Crawler em Java

Web Crawler em Scala

Documentação da classe Source

Traits

Recurso de Scala que permite implementar herança múltipla

```
trait Encrypt {
    def encrypt = {
        // some complex logic to encrypt the data
        toString.reverse
    }
}
class Customer(val firstName : String, lastName : String) extends Encrypt {
    override def toString = "Details for customer " + firstName + " - " + lastName
}
class Security(val name : String)
class Stock(override val name : String) extends Security(name) with Encrypt {
    override def toString = "Details for stock " + name
}
val stock = new Stock("XYZ")
println(stock.encrypt)
```

Traits

Também podemos usar *traits* com instâncias

```
trait Encrypt {
    def encrypt = {
        // some complex logic to encrypt the data
        toString.reverse
    }
}
class CheckingAccount(val number : Int) {
    override def toString = "Details for account " + number
}
val anAccount = new CheckingAccount(1)
val secretAccount = new CheckingAccount(2) with Encrypt
println(secretAccount.encrypt)
```

Traits – Decorator Pattern

```
case class Person(first: String, last: String) {
	override def toString = first + " " + last
}
abstract class Checker {
	def check(applicant : Person) : Boolean
}
def evaluateApplicant(applicant : Person, checker: Checker) = {
	println("Received application for " + applicant)
	val result = if(checker.check(applicant)) "approved" else "disapproved"
	println("Application " + result)
}
class EducationChecker extends Checker {
	override def check(applicant : Person) = {
	println("checking education...")
	//...
	applicant.last.length > 1
	}
}
val john = Person("John", "X")
evaluateApplicant(john, new EducationChecker)
```

Traits – Decorator Pattern

```
trait CreditChecker extends Checker {
    abstract override def check(applicant : Person) = {
        println("checking credit...")
        //...
        super.check(applicant) && (applicant.last.length > 1)
    }
}
trait CrimeChecker extends Checker {
    abstract override def check(applicant : Person) = {
        println("checking criminal records...")
        //...
        super.check(applicant) && (applicant.last.length > 1)
    }
}
evaluateApplicant(Person("Mark", "Who"), new EducationChecker
    with CrimeChecker with CreditChecker)
```

Casamento de Padrões

```
def process(msg : Any) = {
    msg match {
        case "hello" => println("hello recebido")
        case x : String => println("String " + x + " recebida)
        case (a, b) => println("Tupla (" + a + ", " + b + ") recebida")
        case 22 => println("22 recebido")
        case x : Int if x < 0 => println("Recebido numero negativo: " +
x)
        case y : Int => println("Numero recebido: " + y)
        _ => println("Padrao nao reconhecido!")
    }
}
```

Padrões podem conter valores específicos ou restringir tipos

Conversões Implícitas de Tipo

```
class Rational(n: Int, d: Int) { ...
    def + (that: Rational): Rational = ...
    def + (that: Int): Rational = ...
}

scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2
scala> oneHalf + oneHalf
res4: Rational = 1/1
scala> oneHalf + 1
res5: Rational = 3/2
scala> 1 + oneHalf
<console>:6: error: overloaded method value + with alternatives
(Double)Double <and> ... cannot be applied to (Rational)
1 + oneHalf
```

Conversões Implícitas de Tipo

```
class Rational(n: Int, d: Int) { ...
    def + (that: Rational): Rational = ...
    def + (that: Int): Rational = ...
}

scala> implicit def intToRational(x: Int) = new Rational(x, 1)
intToRational: (Int)Rational
scala> 1 + oneHalf
res6: Rational = 3/2
```

Inicialmente o compilador busca em Int por uma operação + que recebe um Rational

Não encontrando, este procura por definição que converta para Rational

XML

```
val greetings = <greet>hello</greet>
println(greetings)
println(greetings.label + ":" + greetings.text)

greetings match {
    case <greet>{msg}</greet> => println("found " + msg) //found hello
}

val message = <message priority="urgente">retornar ligacao</message>
println("Recebida " + message\"@priority" + " mensagem: " +
    message.text)
```

XML é manipulado como um valor de primeira classe em Scala

Programação Concorrente e Paralela

Programação Concorrente – Multiplicação de Matrizes

$$A = \begin{bmatrix} 4 & 0 & 5 \\ 1 & 1 & 3 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 \\ 2 & 5 \\ 1 & 0 \end{bmatrix} \quad AB = \begin{bmatrix} 9 & 8 \\ 6 & 7 \end{bmatrix}$$

Dada as matrizes A e B, encontrar a matriz AB

Programação Concorrente – Multiplicação de Matrizes

```
class Matriz (val linhas : Int, val colunas : Int) {
   var valores = Array.ofDim[Float](linhas, colunas)
}
```

Programação Concorrente – Multiplicação de Matrizes

```
class Matriz (val linhas : Int, val colunas : Int) {
   var valores = Array.ofDim[Float](linhas, colunas)
}

object Matriz {
   def random(m: Int, n: Int): Matriz = {
      var matriz = new Matriz(m, n)

      for (i <- 0 to m - 1)
         for (j <- 0 to n - 1)
            matriz.valores(i)(j) = Math.random.toFloat
      matriz
   }
}
```

Programação Concorrente – Multiplicação de Matrizes

```
class Matriz (val linhas : Int, val colunas : Int) {
   var valores = Array.ofDim[Float](linhas, colunas)

   def x(m: Matriz): Matriz = {
      var m3 = new Matriz(this.linhas, m.colunas)

      if (this.colunas == m.linhas)
         for (i <- 0 to this.linhas - 1)
            for (j <- 0 to m.colunas - 1)
               for (k <- 0 to m.linhas - 1)
                  m3.valores(i)(j) = m3.valores(i)(j) +
valores(i)(k)*m.valores(k)(j)
      m3
   }
}
```

Programação Concorrente – Multiplicação de Matrizes

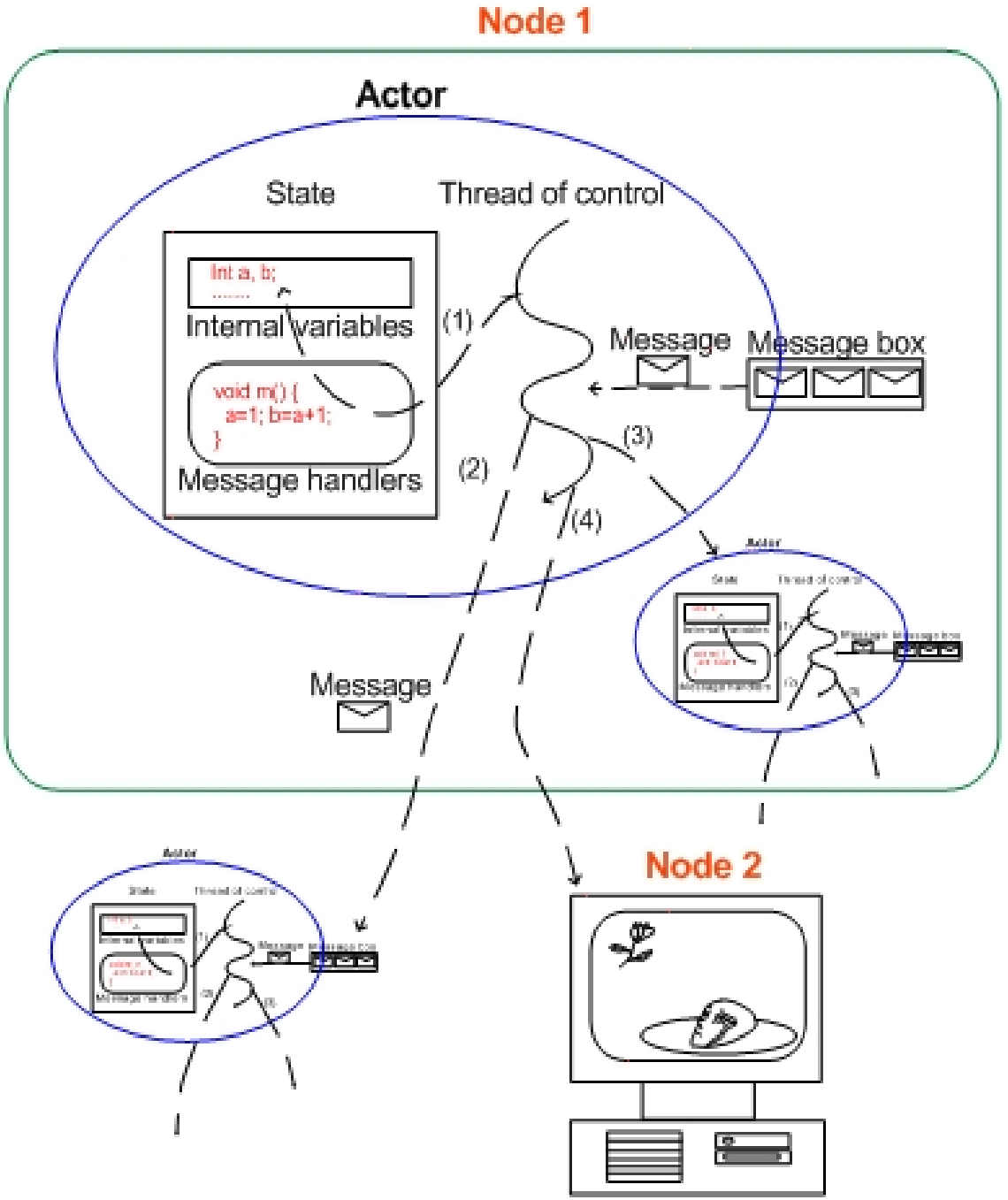
```
object MatrizUso extends App {
   var A : Matriz = null
   var B : Matriz = null

   override def main(args: Array[String]) = {
      var C : Matriz = null
      A = Matriz.random(500,500)
      B = Matriz.random(500,500)

      C = A x B

   }
}
```

Modelo de Atores



```
class Matriz (val linhas : Int, val colunas : Int) {
   var valores = Array.ofDim[Float](linhas, colunas)

   def xx(m: Matriz): Matriz = {
      val mestre = self
      var m3 = new Matriz(this.linhas, m.colunas)

      if (this.colunas == m.linhas) {
         for (i <- 0 to this.linhas - 1) {
            actor {
               for (j <- 0 to m.colunas - 1)
                  for (k <- 0 to m.linhas - 1)
                     m3.valores(i)(j) = m3.valores(i)(j) + valores(i)
(k)*m.valores(k)(j)
               mestre ! i
            }
         }
         for (i <- 0 to this.linhas - 1) {
            receive {
               case ind : Int => {}
               case _ => println ("Caso nao previsto!")
            }
         }
      }

      m3
   }
}
```

Programação Concorrente – Multiplicação de Matrizes

```
object MatrizUso extends App {
   var A : Matriz = null
   var B : Matriz = null

   override def main(args: Array[String]) = {
      var C : Matriz = null
      A = Matriz.random(500,500)
      B = Matriz.random(500,500)

      C = A xx B

   }
}
```

Programação Concorrente – Coleções (I)mutáveis

Em Scala muitas coleções são disponibilizadas de forma mutável e imutável

Estas estão disponíveis nos pacotes: scala.collection.immutable e scala.collection.mutable

Considerações Finais

Linguagem que mistura programação OO e Funcional de maneira natural

Vantagem de interoperação com Java

Linguagem bastante extensível, escalável

Ser flexível e sucinta não necessariamente representa ser simples na programação

Prompt de linha de comando que facilita a depuração

Comunidade ativa

Referências

M. Odersky, L. Spoon, and B. Venners. Programming in Scala. artima, Mountain View, California, 2008.

M. Odersky. Scala By Example - DRAFT. Programming Methods Laboratory, EPFL, Switzerland, 2011.

V. Subramaniam. Programming Scala - Tackle Multi-Core Complexity on the Java Virtual Machine. The Pragmatic Bookshelf, Dallas, Texas, 2008.

http://www.scala-lang.org/

http://en.wikipedia.org/wiki/Scala_(programming_language)

Programação Concorrente em Scala

Carlos Bazilio

carlosbazilio@puro.uff.br