Centro tecnológico
Departamento de Informática

Integrantes:
Enéias
Igor
Alan

Centro tecnológico
Departamento de Informática

Agenda:

- ✓ História
- ✓ Algumas caracteristicas
- ✓ Ranking da Linguagem
- ✓ Criador do Java
- ✓ Programação Funcional
- ✓ Sintaxe Básica
- ✓ Tipos de Dados
- ✓ Imports
- ✓ Operadores
- ✓ Curto-Circuito
- ✓ Variáveis e Constantes
- ✓ Declaração de Classes
- ✓ Controle de Erro
- ✓ Modificadores de Acesso
- ✓ Redigibilidade vs Legibilidade
- ✓ Funções e expressões Simples
- ✓ Funções de Primeira Classe
- ✓ Tuplas
- ✓ Traits
- ✓ Collections
- ✓ Listas
- ✓ Maps
- ✓ Closures
- ✓ Valores Preguiçosos (LAZY)
- ✓ Casamento de Padrões
- ✓ Serialização
- ✓ Desserialização
- ✓ Concorrência
  - ✓ Monitores
  - ✓ Semaforos
  - ✓ Actors

Tempo estimado:
1h20min ± ∞

ERRO! :D

Scala

História

Martin trabalhou como um estudante Ph.D. com Niklaus Wirth em ETH Zuerich como um estudante Ph.D. em Modula-2 e Oberon.

Se juntou com Philip Wadler para escrever uma linguagem de programação funcional que compila Java bytecode.

E então Scala foi desenvolvida em 2001 por Martin Oldersky e pelo grupo dele;

Scala 1.0 Lançada em 2003
Scala 2.8 Lançada em 2010

Scala

Scalable Language

Algumas Características

- Multiplataforma;
  - ✓ JVM E CLR.
- Puro OO;
- Funcional;
- Escalável;
- Concisa;
- Códigos reduzidos;
- Ótimo suporte IDE;
  - ✓ Netbeans,
  - ✓ Eclipse ,
  - ✓ Intellij.
- Estaticamente Tipada
- Compativel com Java
  - ✓ Imports/Bibliotecas
  - ✓ JVM
- Fácil para usar e aprender
- Partten Matching
- Concorrência(Actors)
- Funções Anônimas
- Classes Genéricas
- Closures , Listas , Tuplas...

Scala

Most Popular Coding Languages of 2013

Ranking da LP Scala

Position	Programming Language	Ratings
27	COBOL	0.461
28	Fortran	0.412
29	Lua	0.375
30	Ladder Logic	0.369
31	C shell	0.355
32	Tcl	0.351
33	Scala	0.337
34	RPG (OS/400)	0.337
35	MaxMSP	0.331
36	Go	0.326

TIOBE SOFTWARE

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Scala

Onde Scala está sendo usada?

Scala

Onde Scala está sendo usada?

E outros …

"If i were to pick a language to use today other than java it would be scala."

-James Gosling, criador do Java

SCALA

Programação funcional

Programação funcional é um paradigma de programação que trata a computação como uma avaliação de funções matemáticas e que evita estados ou dados mutáveis. Ela enfatiza a aplicação de funções, em contraste da programação imperativa, que enfatiza mudanças no estado do programa.

SCALA

PF + OO

Scala é uma linguagem compilada, projetada para rodar em um ambiente gerenciado como a JVM ou o CLR do .NET e que oferece a união dos paradigmas funcional e orientado a objetos, principalmente na plataforma Java.

SCALA

Exemplos:

```
var factor = 5
val multiplier = (x : Int ) => x * factor
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

println(numbers.map(multiplier))
factor = 10
println(numbers.map(multiplier))
```

```
def prime(x : Int) : Boolean = 2.to(x-1).filter(x % _ == 0).isEmpty
println(prime(1))
println(prime(5))
println(prime(9))
println(prime(11))
```

Sintaxe Básica

- Case Sensitive;
- Nas classes a primeira letra do nome deve ser maiuscula;
- Todos os nomes de métodos devem começar com uma letra minúsculas.

- **Comentários**

```
object HelloWorld {
/* Programa simples em scala
* Simples e igual ao Java
*/
def main(args: Array[String]) {
// Comentário em uma linha
println("Hello, world!") } }
```

- **Ponto e Virgula**

Ponto e virgula(;) opcional, mas pode ser usado , fazendo tudo em apenas uma linha:

```
val s = "hello"; println(s)
```

Sintaxe Básica

- Estruturas de Controle (if-then-else) & match/switch

```
//Scala

If (condicao){
        ...
}else if (outracondicao){
        ...
}else{
        ...
}
var myVar = “Valor5";

myVar match {
  case “Valor1" => comando1
  case “Valor2" => comando2
  case "Valor3" => comando3
  case "Valor4" => comando4
  case _ => comandoqualquer
}
```

```
//Java

If (condicao){
        ...
}else if (outracondicao){
        ...
}else{
        ...
}
char operacao = ‘+’;

switch( operacao) {
   case “+”: ... break;
   case  “-”: ... break;
   case “*”: ... break;
   case “/”: ... break;
  default: ....
}
```

Sintaxe Básica

- **Estruturas de Repetição**

✓ **For & While & do…While**

```
//Scala

for (i < 0 to 3){
    ...
}
for (s <- args)
        println(s)
while(condition){
    ...
}
do{
            println(a);
            a = a + 1;
}while( a < 20 )
```

```
//Java

for(int i=0; i < 4;i++){
    ...
}
for(String s:args)
        System.out.println(s)
while(condition){
    ...
}
do{
            System.out.println(a);
            a = a + 1;
}while( a < 20 )
```

Sintaxe Básica

- ✓ ***Palavras reservadas***

abstract	case	catch	class
def	do	else	extends
false	final	finally	for
forSome	if	implicit	import
lazy	match	new	null
object	override	package	private
protected	return	sealed	super
this	throw	trait	try
true	type	val	var
while	with	yield	
-	:	=	=>
<-	<:	<%	>:
#	@		

Scala

Tipos de Dados

Data Type	Description
Byte	8 bit signed value. Range from -128 to 127
Short	16 bit signed value. Range -32768 to 32767
Int	32 bit signed value. Range -2147483648 to 2147483647
Long	64 bit signed value. -9223372036854775808 to 9223372036854775807
Float	32 bit IEEE 754 single-precision float
Double	64 bit IEEE 754 double-precision float
Char	16 bit unsigned Unicode character. Range from U+0000 to U+FFFF
String	A sequence of Chars
Boolean	Either the literal true or the literal false

Scala

Tipos de Dados

Inteiros:
0
035
21
0xFFFFFFFF
0777L

Float:
0.0
1e30f
3.14159f
1.0e100
.1

Bollean:
true
false

Caracteres:
'a'
'\u0041'
'\n'
'\t'

String
"Programando em Scala"

Não há diferenças significativas ...

Imports

Totalmente compatível com as bibliotecas do Java...

Scala

```
package logica

import scala.collection._

Import java.util.{List ArrayList}
Import java.io._
```

Java

```
package logica

Import java.util.List
Import java.util.ArrayList

Import java.io.*
```

Scala

Operadores

Aritméticos:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0

Scala

Operadores

Lógicos:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Scala

Operadores

Relacionais:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Operadores

Bit a Bit

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Supondo que:

A = 0011 1100
B = 0000 1101

Temos alguns resultados:

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001

Curto-Circuito

Assim como em java , se o lado esquerdo de uma expressão lógica e avaliada como falsa, o resultado da expressão será definitivamente falsa, então o lado direito não é avaliado.

Da mesma forma, se o lado esquerdo de uma expressão lógica ou é avaliada como verdadeira, o resultado da expressão será definitivamente ser verdade.

Exemplo:

```
scala> a = false
scala> b= true

scala> a && b
Result false

scala> b || a
Result true
```

CURTO CIRCUITO!

Variáveis e Constantes

As declarações de variáveis em Scala são sempre precedidas de **val ou var**, e o tipo da variável é separado do nome por ":", assim:

```
var nome: String = "Jose da Silva";
var idade: Int = 28;
```

Variáveis dinâmicas, facilitando a declaração, reduzindo a preocupação com os tipos (no entanto ela deve ser sempre inicializada durante a declaração):

```
val cidade = "Rio de janeiro"
val taxa = 18.7 // Depois de declarado não pode mudar
```

Declaração de Classes

Classes em Scala são declaradas usando uma sintaxe próxima da sintaxe de Java.
As classes em Scala podem ter parâmetros...

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

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

Metodo sem parâmetros!
Caracteristica da Lp!

Estes argumentos devem ser passados ao se usar new para criar uma nova instância da classe Complex:

```
val c = new Complex(1.2, 3.4)
```

Scala

Modificadores de Acesso

Assim como java , os modificadores de acesso são:

Visibilidade aberta , esse é o padrão;

```
public class mochila(cor: String , capacidade: Double)
        {
        ....
        private def elemento(x :String, y: Double){
                ....
        }
        protect def  desarma(x :String, y: Double)
                {
                ....
        }
}
```

Visibilidade apenas dentro da classe;

Visibilidade dentro do packate , subclasses ...

Herança e Sobrescrita

Scala como o Java, não suporta Herança multipla, e para definir isto em scala também é bem parecido, veja o exemplo abaixo:

```
abstract class Animal {
  def breathe : Unit;
}
class Dog extends Animal {
  def breathe {
    println("Dogs breath!");
  }
}
class Person(val fullName : String, val age : Int) extends Animal {
  def breathe = println("Dogs breath!");
  override def toString = "[Person name:" + fullName + "]"
}
class Student(fullName : String, age : Int) extends Person(fullName, age) {
}
```

Classe Abstrata , Mesmas propriedades do Java.

Necessário uso de override para sobrescrita!

Controle de Erros

Lançando exceções:

Muito semelhante ao Java , você cria um objeto de exceção e em seguida, você lança com throw:

```
throw new IllegalArgumentException
```

Capturando exceções:

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Test {
        def main (args: Array[String]){
                try{
                  val f = new FileReader("input.txt")
                }catch{
                  case ex: FileNotFoundException => {
                  println("Missing file exception")
                }
                 case ex: IOException => {
                 println("IO Exception")
                }                          }
        }
}
```

Controle de Erros

Finally:

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Test {
        def main (args: Array[String]){
                try{
                  val f = new FileReader("input.txt")
                }catch{
                  case ex: FileNotFoundException => {
                  pintln("Missing file exception")
                }
                 case ex: IOException => {
                 println("IO Exception")
                }finally{
                  println("Exiting finally...")
                }
        }
}
```

Redigibilidade vs Legibilidade

Código em Java:

```java
1  public class Carro {
2      private final int ano;
3      private int kilometro;
4
5      public Carro(final int ano, final int kilometro){
6          this.ano = ano;
7          this.kilometro = kilometro;
8      }
9
10     public Carro(final int ano){
11         this.ano = ano;
12         this.kilometro = 0;
13     }
14
15     public int getAno(){
16         return this.ano;
17     }
18
19     public int getKilometro(){
20         return this.kilometro;
21     }
22
23     public void setKilometro(final int kilometro){
24         this.kilometro = kilometro;
25     }
26
27     public void dirigir(final int distancia){
28         this.kilometro += distancia;
29     }
30
31     public void virar(final String direcao){
32         System.out.println("Virando: " + direcao);
33     }
34
35     public static void main(String[] args) {
36         Carro a = new Carro(2014, 10);
37         Carro b = new Carro(2013);
38
39         System.out.println(b.getKilometro());
40         b.dirigir(25);
41         System.out.println(b.getKilometro());
42     }
43 }
```

Redigibilidade vs Legibilidade

Código em Scala …

```
1   class Carro(val ano:Int, var kilometro:Int){
2
3       def this(ano:Int){
4           this(ano, 0)
5       }
6
7       def dirigir(distancia:Int){
8           kilometro += distancia
9       }
10      def virar(direção:String){
11          println("Virando: " + direção)
12      }
13  }
14
15  var a = new Carro(2014, 10)
16  var b = new Carro(2013)
17
18  println(b.kilometro)
19  b.dirigir(25)
20  println(b.kilometro)
```

```
0
25
```

Funções e Expressões Simples

Scala vem com um interpretador que pode ser visto como uma calculadora sofisticada. O usuário interage com a calculadora digitando expressões. Exemplos:

```
scala> 87 + 145
unnamed0: Int = 232

scala> 5 + 2*3
unnamed1: Int = 11

scala> "hello" + " world!"
unnamed2: java.lang.String = hello world!
```

Funções e Expressões Simples

Scala vem com um interpretador que pode ser visto como uma calculadora sofisticada.
O usuário interage com a calculadora digitando expressões.
Exemplos:

```
scala> 87 + 145
unnamed0: Int = 232

scala> 5 + 2*3
unnamed1: Int = 11

scala> "hello" + " world!"
unnamed2: java.lang.String = hello world!
```

Também é possível dar nome a uma sub-expressão e usar o nome, ao invés da expressão, nas expressões seguintes:

```
scala> def scale = 5

scale: Int scala> 7*scale
unnamed3: Int = 35
```

Funções de Primeira Classe

Uma função em Scala é um valor de primeira classe. Como qualquer outro valor, pode ser passado como um parâmetro ou retornado como um resultado.

Exemplos:

Função que soma todos os inteiros entre dois números dados

```
def sumInts(a: Int, b: Int): Int =
if (a > b)   0
else a + sumInts(a + 1, b)
```

Soma os quadrados de todos os inteiros entre dois dados números, a e b

```
def square(x: Int): Int = x * x

def sumSquares(a: Int, b: Int): Int =
if (a > b) 0
else square(a) + sumSquares(a + 1, b)
```

Funções de Primeira Classe

Funções Anônimas

Uma função anônima é uma expressão que é avaliada para uma função; a função é definida sem receber um nome. Como exemplo considere a função anônima quadrado:

```
(x: Int) => x * x
```

A parte anterior a flecha '=>' são os parâmetros da função, ao passo que a parte seguinte a '=>' é o seu corpo.

Scala

Funções de Primeira Classe

Currying

A última formulação das funções de soma já são bem compactas. Mas podemos fazer ainda melhor. Observe que a e b aparecem como parâmetros e argumentos de cada função, mas não parecem tomar parte de combinações interessantes. Há algum modo de nos livrarmos delas?

```
def sumInts = sum(x => x)
def sumSquares = sum(x => x*x)
def sumPowersOfTwo = sum(powerOfTwo)
```

Agora usando as funções obtemos:

```
scala> sumSquares(1, 10) + sumPowersOfTwo(10, 20)
unnamed0: Int = 2096513
```

Tuplas

As vezes, uma função precisa retornar mais de um resultado. Por exemplo, suponha a função divmod que retorna o quociente inteiro e o resto de dois argumentos inteiros dados!
Scala resolve isso com o Tuple2 :

```
def divmod(x: Int, y: Int) = new Tuple2[Int, Int](x / y, x % y)
```

Tuplas são tão convenientes que Scala define uma sintaxe especial para elas. Para formar uma tupla com n elementos x1, ..., xn pode-se escrever (x1, ..., xn).
Isto é equivalente a Tuplen(x1, ..., xn)

Traits

Semelhante à interfaces, mas que podem conter implementação e são acumuláveis.
Um Trait se assemelha à combinação de uma classe abstrata e uma interface em uma única definição.
Esse exemplo mostra como implementar uma fila de números inteiros e combinar traits para dobrar, incrementar e filtrar:

```scala
import scala.collection.mutable.ArrayBuffer

//Type Definition
abstract class IntQueue {
  def get(): Int
  def put(x: Int)
  def size(): Int
}

//ArrayBuffer implementation
class IntQueueImpl extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get = buf remove 0
  def put(x: Int) { buf += x }
  def size = buf.length
}

trait Doubling extends IntQueue {
  abstract override def put(x: Int) {
    super.put(2 * x)
  }
}
```

```scala
trait Incrementing extends IntQueue {
  abstract override def put(x: Int) {
    super.put(x + 1)
  }
}

trait Filtering extends IntQueue {
  abstract override def put(x: Int) {
    if (x > 0) super.put(x)
  }
}
```

Scala.collection hierarquia

Scala.collection imutável

Scala.collection mutável

Arrays

Scala proporciona uma estrutura de dados, o array, que armazena um tamanho fixo sequencial de elementos do mesmo tipo.

Declarando:

```
var z:Array[String] = new Array[String](3)
ou
var z = new Array[String](3)
```

Inicializando:

```
var z = Array("Zara", "Nuha", "Ayan")
```

Array multidimensional:

Declarando:

```
var Matriz= ofDim[Int](3,3)
```

Inicializando:

```
for (i <- 0 to 2) {
        for ( j <- 0 to 2) {
                Matriz(i)(j) = j;
        }}
```

Collections - List

```
//Scala

val numeros = List(1, 2, 3, 4)

val numero 1::2::3::4::NIL




numeros(1)
=>2
```

```
//Java

List<integer> numeros =
new ArrayList< integer> ();

numeros.add(1);
numeros.add(2);
numeros.add(3);
numeros.add(4);

Numeros.get(1)
=>2
```

Scala

Collections - List

Existem muitos tipos de operação em List , como:
isEmpty, tail, head, **concat**, **fill**, tabulate, reverse, etc

isEmpty: Verifica se está vazio. Exemplo:

```
object Test {
   def main(args: Array[String]) {
      val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
      val nums = Nil

      println( "Head of fruit : " + fruit.head )
      println( "Tail of fruit : " + fruit.tail )
      println( "Check if fruit is empty : " + fruit.isEmpty )
      println( "Check if nums is empty : " + nums.isEmpty )
   }
}
```

Collections - Map

```
//Scala

var m = Map(1 -> “Maça”)

m+=2 -> “Laranja”

m(1)
=>Maça
```

```
//Java

Map<Int, String> m =
   new HashMap<Int, String>();

m.put(1, “Maça”);
m.put(2, “Laranja”);

m.get(1);
=> Maça
```

Scala

Closure

Um closure é uma função, cujo valor de retorno depende do valor de uma ou mais variáveis declaradas fora desta função. Considere o seguinte trecho de código com a função anônima:

```
val multiplier = (i:Int) => i * 10
```

Aqui, a única variável utilizada no corpo da função, é i, o qual é definido como um parâmetro para a função. Agora vamos dar um outro pedaço de código:

```
var factor = 3
val multiplier = (i:Int) => i * factor
```

```
object Test { def main(args: Array[String]) {
    println( "muliplier(1) value = " + multiplier(1) )
    println( "muliplier(2) value = " + multiplier(2) ) }
    var factor = 3
    val multiplier = (i:Int) => i * factor }
```

Agora existe a variável factor , com uma referencia para fora da função ...

Temos a seguinte saída:

```
C:/>scalac Test.scala
C:/>scala Test
muliplier(1) value = 3
muliplier(2) value = 6
```

Valores Preguiçosos (Lazy)

Valores preguiçosos oferecem uma maneira de postergar a inicialização de um valor até o primeiro momento em que seja acessado. Isto pode ser útil quando estiver lidando com valores que podem não ser necessários durante a execução e cujo custo computacional seja significativo.

Exemplo acesso a um banco de dados:

```
case class Employee(id: Int,
name: String,
managerId: Int) {
    lazy val manager: Employee = Db.get(managerId)
    lazy val team: List[Employee] = Db.team(id)
}
```

Desta forma, atrasamos o acesso ao banco de dados até o momento em que ele seja realmente necessário, se isto ocorrer.

Serialização

```
import java.io.FileOutputStream
import java.io.ObjectOutputStream

@SerialVersionUID(15L)
class Animal(name: String, age: Int) extends Serializable {
override def toString = s"Animal($name, $age)"
}

case class Person(name: String) extends Serializable

object MySerialize extends App {
val fos = new FileOutputStream("../a.tmp")
val oos = new ObjectOutputStream(fos)

oos.writeObject(new Animal("Dvorak", 12))
oos.writeObject(Person("Dijkstra"))
oos.close
}
```

Scala

Desserialização

```
import java.io._

@SerialVersionUID(15L)
class Animal(name: String, age: Int) extends Serializable {
override def toString = s"Animal($name, $age)"
}

 case class Person(name: String)

class ObjectInputStreamWithCustomClassLoader(fileInputStream: FileInputStream) extends
ObjectInputStream(fileInputStream) {
  override def resolveClass(desc: java.io.ObjectStreamClass): Class[_] = {
   try { Class.forName(desc.getName, false, getClass.getClassLoader) }
   catch { case ex: ClassNotFoundException => super.resolveClass(desc) }
  }
}
```

```
object MyDeserialize extends App {
val fis = new FileInputStream("../a.tmp")
val ois = new ObjectInputStreamWithCustomClassLoader(fis)

val animal = ois.readObject
val person = ois.readObject
ois.close
println(animal)
println(person)
}
```

XML

- ✓ A linguagem possibilita a inserção de códigos XML diretamente em programas Scala
- ✓ O inverso também vale, inserir códigos scala em arquivos XML

```
val labPhoneBook =
    <phonebook>
    <descr>Phone numbers of<b>XML</b> hackers.</descr>
    <entry>
        <name>Burak</name>
        <phone where="work"> +41 21 693 68 67 </phone>
        <date> { df.format(new java.util.Date()) } </date>
    </entry>
    </phonebook>;
```

XML

Mais um exemplo...

```
val xml = <item>
            <title>Lift Version 1.0 Released</title>
            <link>http://www.scala-lang.org/node/1011</link>
            <description>Twelve is &gt; six</description>
            <comments>http://www.scala-lang.org/node/1011#comments</comments>
            <category domain="http://www.scala-lang.org/taxonomy/term/15">Featured</category>
            <pubDate>Fri, 27 Feb 2009 10:02:55 +0000</pubDate>
            <dc:creator>bagwell</dc:creator>
            <guid isPermaLink="false">1011 at http://www.scala-lang.org</guid>
</item>
```

Scala

Casamentos de Padrões

Casamento de padrões sobre hierarquias de classe unificam o acesso a dados funcional e orientado a objetos.

A casamento de padrões é o segundo recurso mais utilizado do Scala, a LP proporciona Grande suporte para casamentos de padrões para o processamento das mensagens.

```
object Test{
def main (args: Array[String]){
        println(matchTest(3))
}
def matchTest(x:int):String = x match {
        case 1 => “one”
        case 2 => “two”
        case _ => “many”
}
```

```
//Saída...

C:/>scalac Test.scala
C:/>scala Test

many
```

O casamento de padrões é uma generalização do comando switch do C ou Java para hierarquias de classes.

Concorrência

Monitores:

Um monitor provê mecanismos básicos para processos mutuamente exclusivos em Scala.

```
def synchronized[A] (e: => A): A
def wait()
def wait(msec: Long)
def notify()
def notifyAll()
```

O método synchronized computa e em modo mutuamente exclusivo – em um dado momento qualquer, somente uma thread pode executar um argumento synchronized em um dado monitor.

Concorrência

Semáforos

Um mecanismo comum para sincronização de processos é o uso de travas (lock) ou semáforo. Uma trava oferece duas operações atômicas: acquire e release. Aqui está a implementação de uma trava en Scala:

```
package scala.concurrent

class Lock {
   var available = true
   def acquire = synchronized {
   while (!available) wait()
   available = false
}
def release = synchronized {
   available = true
   notify()
 }
}
```

Concorrência

Actors

A resposta da Scala para facilitar a programação concorrente são os Actors (ou atores), que são analogos aos Threads do Java, mas muitos recursos para facilitar é uma grande diferença, Actors em scala possuem um protocolo definido para se comunicarem, eles enviam mensagens uns para os outros em vez de acessar
as mesmas variáveis que outras threads.

```
package scala.concurrent

abstract class Actor extends Thread with  MailBox {
def act(): Unit
  override def run(): Unit = act()
   def !(msg: Any) = send(msg)
}
```

Avaliação da Linguagem

- ✓ Presa muito a redigibilidade e reusabilidade;
- ✓ A legibilidade em algumas ocasiões é deficiente;
- ✓ Tenta manter o máximo de Confiabilidade;
- ✓ A eficiência é alta;
- ✓ A facilidade de aprendizado é moderada;
- ✓ A portabilidade é alta.

Scala

Dúvidas

Scala
Microsoft
.net

Scala
Microsoft
.net

OBRIGADO!

Speed
Slow
Fast
Java
Scala

Scala

Referências:

- ✓ http://blog.caelum.com.br/scala-sua-proxima-linguagem/
- ✓ http://www.scala-lang.org/
- ✓ http://sobrecodigo.com/o-que-eu-gostei-na-linguagem-scala/
- ✓ http://www.kylehailey.com/popularity-vs-productivity-vs-performance/
- ✓ http://blog.caelum.com.br/como-anda-a-linguagem-scala/
- ✓ http://programacaobr.com/
- ✓ http://www.tutorialspoint.com/scala

Scala