Gabriel Santa Clara Ucelli

Sumário

Histórico

- Scala foi desenvolvida em 2001 por Martin Odersky e pelo grupo dele na École Polytechnique Fédérale de Lausanne (EPFL), Lausana na Suíça.
- Scala é a sucessora de Funnel
- O design do Scala começou em 2001. Um primeiro lançamento ao público foi em 2003. Em 2006, uma segunda, versão remodelada foi lançada como Scala v 2.0. Desde então a linguagem tem ganhado popularidade. Atualmente a linguagem está na versão 2.11.

Introdução

- Scala (Scalable language) é uma linguagem moderna de programação multiparadigma
- Projetada para expressar padrões de programação comuns de uma forma concisa, elegante e *type-safe* (é uma medida em que a linguagem de programação desestimula ou impede erros de tipo).
- Ela incorpora de forma suave recursos de linguagens orientadas a objetos e funcionais.
- Grandes empresas como Twitter, Foursquare, Intel e LinkedIn usam Scala.

Visão Geral

- Scala é orientada a objetos
- Scala é funcional
- Códigos Reduzidos
- Compilada
- Scala roda na JVM (Java Virtual Machine) e CLR (plataforma .NET)
- Scala pode executar código Java puro
- Scala possui frameworks para web

Visão Geral

- Todos os tipos são objetos (não possui tipos primitivos)
- Possui inferência estática de tipos
- Possui funções e métodos aninhados
- Funções também são objetos
- Possui traits
- Possui closures
- Possui herança múltipla (mixin classes)
- Suporte à concorrência inspirado em Earlang

Tudo é Objeto

```
1 + 2 * 3 / x
```

```
1.+(2.*(3./(x)))
```

```
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)
  }
}
```

21	SAS	0.896%
22	Fortran	0.830%
23	Ada	0.817%
24	Lisp	0.792%
25	ABAP	0.709%
26	D	0.696%
27	Dart	0.678%
28	Scala	0.678%
29	Lua	0.649%
30	Scratch	0.598%

Ranking segundo TIOBE

Primeiro exemplo:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Compilando:

```
> scalac HelloWorld.scala
```

Executando:

```
> scala -classpath . HelloWorld

Hello, world!
```

Conceitos Básicos

Palavras reservadas

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

Tipos

Tipos	Descrição
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
Unit	Corresponds to no value
Nulll	null or empty reference
Nothing	The subtype of every other type; includes no values
Any	The supertype of any type; any object is of type *Any*
AnyRef	The supertype of any reference type

Amarrações

- Qualquer nome em Scala identifica um método, tipo, valor ou uma classe.
- Amarrações possuem diferentes tipos de precedências. Por exemplo declarações tem precedências sobre imports.
- Uma amarração em um escopo mais interno faz sombra em amarrações de precedência menor no mesmo escopo ou em escopos mais externos.
- Existe dois espaços de nomes distintos: Para tipos e nomes.

Verificação de Tipos

- Scala é fortemente tipada.
- Possui verificação de tipos estática e dinâmica.
- Maior parte das verificações é em tempo de compilação.
- Para poder dar suporte à Orientação a Objeto faz algumas verificações em tempo de execução.
- O compilador é capaz de realizar inferência de tipos para variáveis e funções. Já para os parâmetros é necessário declarar os tipos explicitamente.

Tipos

- Classes genéricas
- Anotações de variância
- Tipos de limite superior e inferior
- Classes e enumeração (tipo de dado)
- Tipo composto
- Auto referencia
- Métodos polimórficos.

Identificadores

Devem começar com uma letra ou underscore e pode ser seguido por letras, números ou underscores. Os símbolos $, # e @ são palavras reservadas em Scala e não devem ser usados em identificadores.

var, _myvar, __myvar_, myvar123_, _1_myvar
(identificadores válidos)

var$, 123abc, -myvar
(identificadores inválidos)

Variáveis e constantes

- Variáveis em Scala não necessitam de declaração por tipo
- Variável pode ter seu valor alterado posteriormente (mesmo tipo)

```
var a = 5
var b = “String variável”
```

- Constante não pode ter seu valor alterado posteriormente.

```
val a = 5
val b = “String constante”
```

- Pode definir o tipo na inicialização

```
var a : String = “Eu sou uma string”
```

Variáveis e constantes

- Alocação em pilha e/ou monte;
- Utiliza a estratégia utilizada por JVM para alocar e desalocar recursos (stack, heap, garbage collector);
- A JVM possui otimizações para decidir onde alocar cada variável;

Sintaxe

- Por convenção utiliza-se os nomes das classes com letras maiúsculas:

```
class MinhaClasse {...}
```

- E para métodos utiliza-se letras minúsculas no começo:

```
def meuMetodo () {...}
```

- Ponto e virgula (;) é opcional:

```
var a = 2; println (a)
```

Sintaxe

Funções e/ou métodos

```
def retornaDobro(i : Int) = 2*i

def printaDobro(i : Int) : Unit = println(retornaDobro(i))
```

Sintaxe

- Modificadores de acesso:

  public: visibilidade aberta.
  protected: visibilidade dentro do package, subclasses.
  private: visibilidade apenas dentro da própria classe.

Operadores

- Os operadores aritméticos e lógicos são os mesmo de Java
- Não tem a operação ++ e --
- Tem curto-circuito nos operadores lógicos

Condicionais e iterações

- **If/else**, **while** e **do while** são exatamente como são em Java
- O match é o equivalente de switch em Java

```
x match {
case 0 => …
case 1 => …
case 2 =>

}
```

- Existem várias formas de fazer **for**

Iterações

```
for(i <- 0 to 100){...}
for(i <- 0 until 100){...}
for(i <- Range(0, 100)){...}
for(i <- 0 to 100; j <- 0 to 100){...}
```

[0..100]
[0..99]
[0..99]
[0..100]

```
for(i <- 0 to 10 if (i != 3); if(i != 6)){..}
```

0 1 2 4 5 7 8 9

Iterações

```
var lista = List(0,1,2,3,4,5)
for(i <- lista){
println (i)
}
```

0 1 2 3 4 5

```
var valor = for(i <- 0 to 5 if(i < 3)) yield i
println (valor)
```

vector(0,1,2)

Parâmetros - varargs

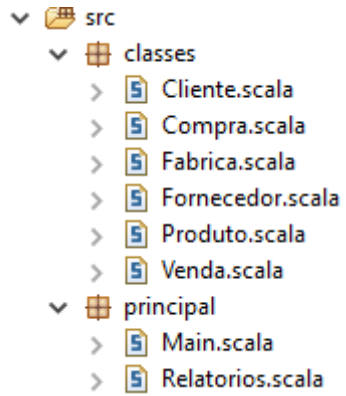
- A declaração deve estar na última posição do método ou função.
- Utilizamos o operador *.

```
object ExemploVar
{
  def main(args: Array[String])
  {
    imprime(10, "um", "dois", "tres", "etc")
  }

  def imprime(x: Int, strings: String*)
  {
    println(x)
    strings.map(println)
  }
}
```

Modularização

- Modularização por pacotes e parâmetros
- Pacotes aninhados: é possível definir pacotes dentro de um pacote.



Integração com 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)
  }
}
```

OBSERVAÇÃO

```
df format now
```

```
df.format(now)
```

Classe

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

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

```
object ComplexNumbers {
  def main(args: Array[String]) {
    val c = new Complex(1.2, 3.4)
    println("imaginary part: " + c.im())
  }
}
```

Classes estáticas são chamadas object
Também existem um tipo de Classe chamada case class

Mixins

- Talvez a forma mais fácil para um programador Java entender o que são **mixins** é vê-los como **interfaces** na qual podem também **conter código**.
- Em Scala, quando uma classe é sub-classe de mixin, ela implementa aquela interface mixin e **herda todo o código contido no nele**

Mixins

```
trait imprimiOi
{
  def Oi = println("Oi")
}

case class Pessoa(nome : String) extends imprimiOi
{}


object main extends App {
  var a = new Pessoa("Gabriel")
  println(a)
  if( a.isInstanceOf[imprimiOi])
  {
    a.Oi
  }
}
```

```
Pessoa(Gabriel)
Oi
```

Polimorfismo

Coerção

- Tipos menores são convertidos para tipos maiores:

  Int -> **Float**

  Feito implicitamente.

- Não é feito o contrário. Ou seja, operações de estreitamento não são realizadas implicitamente.

  Float -> **Int** (Não é valido)

Polimorfismo

Sobrecarga

- Scala possui sobrecarga de métodos e operadores, sendo possível usar o mesmo nome em diferentes amarrações.
- Em herança, declaramos a sobrecarga de métodos com a palavra reservada override.
- Permite a definição de novos operadores.

Polimorfismo

Inclusão

- Sistema de herança com extends.
- A herança pode ser simples ou múltipla. Na herança múltipla utilizamos a palavra reservada with.
- Colisão de nomes pode ser tratada com as palavras chaves abstract override, assim

a classes extensoras são forçadas a implementarem seus próprios métodos.

Polimorfismo

Sobrecarga

```
class deAula
{
  def chamada(qtd:Int)
  {...}

  def chamada()
  {...}
}
```

Inclusão

```
public class Pneu {...}

public class Motor {...}

public class Freios {...}

public class Carro extends Pneu with Motor with Freios{...}
```

Paramétrico

```
var vetor : Array[String] = Array();

var mapa : Map[Int, String] = Map();
```

Exceções

- Muito semelhante a Java. Com a criação de blocos try {...} catch {...}.

```
try {

  throwsException();

} catch {

  case e: IllegalArgumentException => println("illegal arg. exception");
  case e: IllegalStateException    => println("illegal state exception");
  case e: IOException              => println("IO exception");

} finally {

  println("this code is always executed");

}
```

- Utiliza **pattern matching** no catch para saber qual exceção foi capturada.

Concorrência

- Scala suporta os mesmos métodos de concorrência utilizados por Java.
- Possui um pacote de concorrência nativo, o **scala.concurrent**.
- Implementa alguns métodos que Haskell utiliza.
- A resposta da Scala para facilitar a programação concorrente são os **Actors** (ou atores), que são análogos 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.

Concorrência

- Sinais e monitores
- SyncVars
- Futuros
- Computação Paralela
- Semáforos
- Canais Assíncronos e síncronos
- Actors

Avaliação da Linguagem

- Aplicabilidade;

  Parcial

- Confiabilidade;

  Sim

- Aprendizado;

  Parcial, pode se torna muito complexa com o uso de ferramentas mais avançadas

- Eficiência;

  Parcial

Avaliação da Linguagem

- Portabilidade:

Sim, roda sobre a JVM e CLR

- Método de projeto:

OO e Funcional

- Evolutibilidade:

Sim

- Reusabilidade:

Sim

Avaliação da Linguagem

- Integração

Sim

- Custo

Depende

Referencia

1. http://www.scala-lang.org
2. https://wiki.scala-lang.org/pages/viewpage.action?pageId=294934
3. http://www.scala-lang.org/api/2.10.1/index.html#package
4. http://www.scala-lang.org/docu/files/ScalaTutorial-pt_BR.pdf
5. http://www.scala-lang.org/old/sites/default/files/linuxsoft_archives/docu/files/ScalaByExample-pt_BR.pdf
6. http://stackoverflow.com/
7. http://tutorialspoint.com/scala/
8. https://pt.wikipedia.org/wiki/Scala_%28linguagem_de_programa%C3%A7%C3%A3o%29
9. http://scalatutorials.com/tour/
10. http://twitter.github.io/scala_school/
11. https://www.youtube.com/watch?v=gtREsSfseyk&list=PLAIXw9RKLY_1bBn_u-i_qjmRpZw2ogeKV