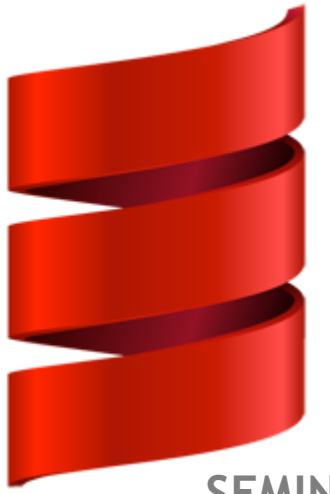
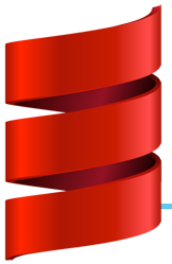


LINGUAGENS DE PROGRAMAÇÃO

SEMINÁRIO SOBRE A LINGUAGEM DE PROGRAMAÇÃO SCALA

LUIZ EDUARDO FAVALESSA PERUCH
RAFFAEL CALEGARI BOZZI
RENAN MIRANDA LUCAS

ORIENTADOR: PROF. DR. VÍTOR E. S. SOUZA



O QUE É SCALA?

- **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**, **Intel** e **LinkedIn** usam Scala[1].

[1] - http://www.scala-lang.org/what-is-scala.html

VISÃO GERAL

- Scala é orientada à objetos;
- Scala é funcional;
- Scala usa amarração estática;
- Scala roda na JVM;
- Scala pode executar código Java puro;
- Scala possui frameworks para Web.

SCALA vs JAVA

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

SINTAXE BÁSICA

- **Case-sensitive:**
  - Os identificadores hello e Hello possuem significado diferente para a linguagem.
- **Nomes de classes:**
  - Nomes de classes devem começar com letra maíuscula;
  - Nomes de classes compostos devem começar cada palavra com letra maiúscula.
  - Exemplo:

```
class MinhaPrimeiraClasse { }
```

- **Nomes de métodos:**
  - Nomes de métodos devem começar com letra minúscula. Caso sejam nomes compostos, as inicias das próximas palavras devem ser maiúsculas.
  - Exemplo:

```
def meuPrimeiroMetodo () = { }
```

SINTAXE BÁSICA

- **Nome do arquivo do programa:**
  - O nome do arquivo deve corresponder exatamente ao nome do objeto ou classe seguido da extensão .scala. Caso não seja, seu programa não irá compilar.
  - Exemplo:

    Caso o nome do objeto seja *HelloWorld*, o arquivo deverá ser nomeado *HelloWorld.scala*.
- **Função main:**
  - Um programa em Scala começa pela função *main*, portanto é obrigatório que a mesma esteja definida no código.

```
def main (args : Array[String] ) { }
```

IDENTIFICADORES

- **Identificadores alfanuméricos:**
  - 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.
  - Exemplo:

var, _myvar, __myvar_, myvar123_, _1_myvar

(identificadores válidos)

var$, 123abc, -myvar

(identificadores inválidos)

  - Para criar variáveis em Scala, usamos as palavras chave **var** ou **val**.

```
var x = "Eu sou uma variável!"
val y = "Eu sou uma constante!"
```

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	
-	:	=	=>
<-	<:	<%	>:
#	@		

AMARRAÇÕES

- Nomes em Scala identificam tipos, valores, métodos e classes, coletivamente chamados **entidades**.
- Amarrações de diferentes tipos têm diferentes precedências associadas:
  - Definições e declarações (mesma unidade de compilação);
  - imports explícitos;
  - imports usando curingas;
  - Definições (disponíveis em um pacote fora da unidade de compilação).
- Existem dois espaços de nomes distintos: um para **tipos** e um para **nomes**.

AMARRAÇÕES

- Uma amarração possui um escopo onde uma entidade pode ser acessada;
- Escopos são **estáticos** e **aninhados**;
- Uma amarração em um escopo mais interno *faz sombra* em amarrações de precedência menor no mesmo escopo ou em escopos mais externos;
- Uma referência para um identificador não-qualificado (**tipo** ou **termo**) ***x*** possui uma amarração única, que:
  - Define uma entidade com o nome ***x*** no mesmo espaço de nome do identificador;
  - Faz sombra em todas as outras amarrações que definem entidades com o nome ***x*** naquele espaço de nome.

TIPOS DE DADOS

Tipo de dado	Descrição
Byte	Valor de 8 bits com sinal. Range de -128 à 127
Short	Valor de 16 bits com sinal. Range de -32768 à 32767
Int	Valor de 32 bits com sinal. Range de -2147483648 à 2147483647
Long	Valor de 64 bits com sinal. De -9223372036854775808 à 9223372036854775807
Float	Ponto flutuante de 32 bits com precisão simples IEEE 754
Double	Ponto flutuante de 64 bits com precisão dupla IEEE 754
Char	Caracter Unicode de 16 bits sem sinal. Range de U+0000 à U+FFFF
String	Uma cadeia de Chars
Boolean	Assume os valores literais true ou false
Unit	Corresponde à sem valor
Null	Referência vazia ou nula
Nothing	O subtipo de todos os outros tipos; inclui Unit
Any	O supertipo de qualquer tipo; qualquer objeto é do tipo Any
AnyRef	O supertipo de qualquer referência

(DES)ALOCAÇÃO DE VARIÁVEIS

- A alocação de variáveis em Scala pode ser feita na pilha e/ou no monte, dependendo do caso.
- Segue a estatégia usada pela JVM para alocar e desalocar recursos (*stack*, *heap*, *garbage collector*).
- A JVM possui otimizações para decidir onde alocar cada variável.

MECANISMOS DE PERSISTÊNCIA

- Possui o trait Serializable que pode ser extendido pela classe que se deseja serializar (similar ao de Java).
- Método de serialização com ***pickling***[1].
- Funciona com JPA[2].

- [1] http://lampwww.epfl.ch/~hmiller/pickling/
- [2] http://hibernate.org/

OPERADORES

- Aritméticos
- Relacionais
- Lógicos
- Bit a bit
- Atribuição

EXPRESSÕES E OPERADORES

- Podem ser pré-fixas:
  - Pode ser um dos identificadores ‘+’, ‘-’, ‘!’ ou ‘~’. A expressão é equivalente à aplicação do método pós-fixo ***e.unary_op***.
- Podem ser pós-fixas:
  - Uma operação pós-fixa pode ser qualquer indentificador arbitrário. A operação pós-fixa ***e;op*** é interpretada como *e.op*.

EXPRESSÕES E OPERADORES

- Podem ser in-fixas:
  - Um operador in-fixo pode ser um identificador arbitrário.
  - Operadores in-fixos possuem precedência e associatividade definidas como:
    - A precedência de um operador in-fixo é determinada pelo primeiro caracter do operador.

      (todas as letras) (|) (^) (&) (= !) (< >) (:) (+ -) (* / %) (todos os outros caracteres especiais)

    - A associatividade de um operador in-fixo é determinada pelo último caracter do operador.

      opeadores terminados com : possuem associatividade da direita para a esquerda.
      demais operadores possuem associatividade da esquerda para a direita.

CONDICIONAIS

- São expressões condicionais em Scala: if, else, case *.
  - Case deve ser usado com pattern matching

```
object MatchTest1 extends App {
    def matchTest(x: Int): String = x match {
        case 1 => "one"
        case 2 => "two"
        case _ => "many"
    }
    println(matchTest(3))
}
```

```
object MatchTest2 extends App {
    def matchTest(x: Any): Any = x match {
        case 1 => "one"
        case "two" => 2
        case y: Int => "scala.Int"
    }
    println(matchTest("two"))
}
```

ITERAÇÕES

- while, do {...} while, for.
  - while e do { } while funcionam exatamente como em Java.
  - Já o for ...
    - É possível varrer **ranges** definidos pelo programador;
    - Possibilidade de usar **filtros** dentro do loop;
    - Possibilidade de varrer **coleções**;
    - Possui valor de retorno com **yield**.

LOOP FOR

- Usando **ranges:**

```
for(i <- 1 to 10) {
    println(i)
}

for(i <- 1 until 10) {
    println(i)
}
```

```
for(i <- Range(1,10)) {
    println(i)
}

for(i <- 1 to 10; j <- 1 to 10) {
    println(i + " " + j)
}
```

LOOP FOR

- Usando **filtros:**

```
var listaNumeros = List(1,2,3,4,5,6,7,8,9,10)
for(i <- listaNumeros if(i != 3); if(i < 9)) {
    println(i);
}
```

1 2 4 5 6 7 8

LOOP FOR

- Sobre **coleções:**

```
var listaNumeros = List(1,2,3,4,5,6,7,8,9,10)
for(i <- listaNumeros) {
    println(i);
}
```

LOOP FOR

- Usando **yield:**

```
var listaNumeros = List(1,2,3,4,5,6,7,8,9,10)
var retorno = for(i <- listaNumeros if(i != 3); if(i < 9)) { } yield i

for(i <- retorno) println(i)
```

1 2 4 5 6 7 8

MODULARIZAÇÃO: PACOTES

- Pacotes aninhados: é possivel definir pacotes dentro de um pacote.

```
package pacote1 {
    package pacote2 {
        ClasseDoPacote2
        OutraClasseDoPacote2
    }

    ClasseDoPacote1
}

 …

import pacote1._
import pacote1.pacote2.{ClasseDoPacote2,
                        Outra ClasseDoPacote2}
```

```
package society {
    package professional {
        class Executive {
            private[professional] var workDetails = null
            private[society] var friends = null
            private[this] var secrets = null

            def help(another : Executive) {
                println(another.workDetails)
                println(another.secrets) //ERROR
            }
        }
    }
}
```

MODULARIZAÇÃO: PARÂMETROS

- Valores default:

```
class HashMap[K,V](initialCapacity:Int = 16, loadFactor:Float = 0.75) { }

//Usa os valores padrões
val m1 = new HashMap[String,Int]
// initialCapacity 20, loadFactor padrão
val m2= new HashMap[String,Int](20)
// sobrecarregando os dois
val m3 = new HashMap[String,Int](20,0.8)
// sobrecarregando apenas loadFactory com parâmetros nomeados
val m4 = new HashMap[String,Int](loadFactor = 0.8)
// usando parâmetros nomeados
val m5 = new HashMap[String,Int](loadFactor = 0.8, initialCapacity = 20)
```

MODULARIZAÇÃO: PARÂMETROS

- Momento de passagem: ***eager***

```
object Teste{
    def caso(x:Int, b:Int) : Int = {
        println(s"FUNCAO CASO $x"); x
    }

    def incrementa(x:Int) : Int = {
        println(s"FUNCAO INCREMENTA $x");
        return x + 5
    }

    def main (args : Array[String]) {
        println(caso(incrementa(10), incrementa(2)))
    }
}
```

```
FUNCAO INCREMENTA 10
FUNCAO INCREMENTA 2
FUNCAO CASO 15
15
```

PARÂMETROS: VARARGS

- Scala oferece suporte para parâmetros com tamanho variável através do operador ‘*’.
  - A declaração do parâmetro deve estar na última posição do método/função.

```
object ScalaVarargsTests {
    def main(args: Array[String]) {
        printAll(true, "foo", "bar", "baz")
    }

    def printAll(x: Boolean, strings: String*) {
        println(x)
        strings.map(println)
    }
}
```

VERIFICAÇÃO DE TIPOS

- Possui verificação de tipos mista (estática e dinâmica);
  - Faz a maior parte das verificações em tempo de compilação.
  - Algumas verificações em tempo de execução (para suportar OO).
- Scala é fortemente tipada;

VERIFICAÇÃO DE TIPOS

- O compilador de Scala é capaz de realizar inferência de tipos para variáveis e funções. Parâmetros devem ser tipados explicitamente.

```
object ScalaTest {
  def main(args: Array[String]) {
   println(retorno(1))
   println(retorno(2))
   println(retorno(0))
  }

  def retorno (i : Int) =
      if(i == 1) "asd" else if(i == 2) 10 else 1.1
}
```

```
asd
10
1.1
```

POLIMORFISMO: COERÇÃO

- Suporta polimorfismo de coerção.
- Ampliação:
  - Tipos de menores conjuntos são convertidos para tipos de conjuntos maiores (Long para Float);
  - Ampliação é feita implicitamente.
- Estreitamento:
  - Não realiza operações de estreitamento implícitas.
  - Pode ser forçado com o operador **asInstanceOf[T]**

POLIMORFISMO: SOBRECARGA

- É possível usar o mesmo nome em amarrações diferentes;
- Scala possui sobrecarga de métodos e operadores:

```
class Rational (n : Int,  d : Int) {
     private def gcd (x: Int, y: Int): Int = /* … */
     private val g = gcd(n, d)
     val numer: Int = n/g
     val denom: Int = d/g
     def + (that : Rational) = new Rational(numer * that.denom + that.numer * denom,
                                            denom * that.denom)
     def - (that : Rational) = new Rational(numer * that.denom - that.numer * denom,
                                            denom * that.denom)
     def * (that : Rational) = new Rational(numer * that.numer, denom * that.denom)
     def / (that : Rational) = new Rational(numer * that.denom, denom * that.numer)

}
```

POLIMORFISMO: SOBRECARGA

- Em herança, é necessário declarar a sobrecarga de métodos com a palavra-chave ***override***.
- Scala permite a definição de novos operadores.
- Precedência dos operadores pode ser alterada.
- Bem como sua associatividade (*right to left* ou *left to right*).
- Scala possui sobrecarga dependente e independente de contexto.

POLIMORFISMO: PARAMÉTRICO

- Suporta polimorfismo paramétrico com o uso de classes **genéricas**;
- É possível definir traits genéricos em Scala;
- Compilador faz a checagem de tipos;

POLIMORFISMO: INCLUSÃO

- Possui sistema de herança com **extends**;
- Herança pode ser simples ou múltipla com ***mixin-classes***, usando a palavra chave ***with***;
- ***with*** só pode ser usada para um trait;
- Colisão de nomes pode ser tratada com as palavras chave **abstract override**, forçando as classes extensoras a darem a própria implementação do método.

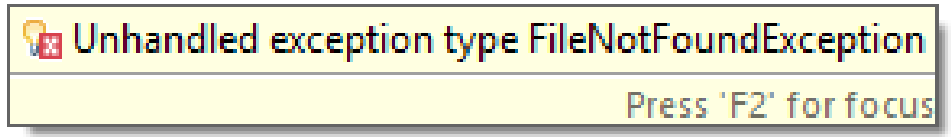
EXCEÇÕES

- Tratamento de exceções semelhante ao de Java com blocos **try { } catch { }**;
- Ao invés de retornar um valor, qualquer método pode terminar a execução lançando uma nova exceção.
- Usa *pattern matching* no **catch** para decidir qual exceção foi capturada;

EXCEÇÕES CHECADAS

- Scala não possui exceções checadas. Fica à critério do programador tratá-las ou não.

```
java.util.Scanner s = new java.util.Scanner(new java.io.File("teste"));
```



- Enquanto isso, em Scala...

```
val s = new java.util.Scanner(new java.io.File("teste"));
```

CONCORRÊNCIA

- Suporta os métodos de concorrência usados por Java;
- Implementa métodos de concorrência usados por Haskell, Earlang;
- Scala possui um pacote de concorrência nativo: scala.concurrent.
- Métodos mais populares:
  - thread-based, Actor-based, Software Transactional Memory, Parallel Collections, Futures.

AVALIAÇÃO DA LINGUAGEM

- Aplicabilidade;
  - Há poucos tipos de problemas que não têm solução simples em Scala.
- Confiabilidade;
  - É type-safe e opta por ser mais flexível sem abrir mão da confiabilidade.
- Aprendizado;
  - Depende do objetivo do programador: pode ser muito simples ou pode ser um inferno.
- Eficiência;
  - Comparada à Java, é tão boa quanto. Mas ...

AVALIAÇÃO DA LINGUAGEM

- Portabilidade:
  - Roda sobre a JVM, roda em .NET (em desenvolvimento).
- Método de projeto:
  - OO e Funcional.
- Evolutibilidade:
  - Scala stands for scalable language.
- Reusabilidade:
  - Polimorfismo universal (OO).

EXTRAS

- Currying;
- Parâmetros implicitos;
- Construção automática de closures dependentes de tipos;
- Funções de ordem superior;
- Parâmetros compostos;
- Processamento de XML;
- Expressões regulares;
- Interpolação de strings;
- Anotações de classes;
- Extratores com objetos;
- ...

REFERÊNCIAS

- http://docs.scala-lang.org/tutorials/tour/tour-of-scala.html
- http://scala-lang.org/
- http://stackoverflow.com/
- http://tutorialspoint.com/scala/
- https://wiki.scala-lang.org/display/SW/Welcome+to+the+Scala+Wiki!?_ga=1.151487984.836857857.1418139827
- http://scalatutorials.com/tour/
- http://twitter.github.io/scala_school/