Scala

Lucas Tabelini Torres
Matheus Vicente Delunardo de Lucena
Otavio Ferreira Cozer
Pedro Anselmo Santana De Angeli

Histórico

- Scala foi desenvolvida por **Martin Odersky** e pelo grupo dele na École Polytechnique Fédérale de Lausanne(EPFL), Lausana na Suíça. Em 1999, depois de se unir a EPFL, a meta era combinar programação orientada a objetos e programação funcional. O primeiro passo para este caminho era o Funnel, no entanto, foi descoberto que linguagem não era agradável para o uso na prática. O segundo, e atual, é o Scala, que trouxe algumas das ideias do Funnel. O design do Scala começou em 2001 e 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.

Índice TIOBE

Position	Programming Language	Ratings
21	Perl	0.831%
22	SAS	0.825%
23	PL/SQL	0.641%
24	Dart	0.615%
25	Rust	0.506%
26	Scratch	0.463%
27	Lisp	0.407%
28	COBOL	0.391%
29	Fortran	0.390%
30	Scala	0.387%
31	RPG	0.385%
32	Transact-SQL	0.380%
33	Logo	0.324%
34	ABAP	0.306%
35	Kotlin	0.300%
36	Ada	0.296%

Conceitos básicos

- Linguagem multiparadigma : orientada a objeto e funcional
- Implementação híbrida
- Executa na JVM
- Compatível com Java

Como compilar e executar

- Para compilar: **scalac Teste.scala**
- Para executar: **scala Teste**

Sintaxe

- Case-sensitive: **var x ≠ var X**
- Ponto e vírgula é opcional. Pode ser usado caso queira tudo em uma única linha. **var x = 2; println(x)**

Sintaxe

- Hello world em scala

```
object Teste {
 def main(args:Array[String]){
   println("Hello World")
 }
}
```

Sintaxe

- Declarando classes

```
class Ponto(var x: Int,var y: Int){}

 def main(args:Array[String]){
   var ponto = new Ponto(5,4);
   println(ponto.x) //5
 }
```

Sintaxe

- Em scala, um getter e um setter são criados implicitamente para atributos não privados. Podemos definir nossos próprios getters e setters.

```
class Ponto(var x: Int,var y: Int){}

 def main(args:Array[String]){
   var ponto = new Ponto(5,4);
   println(ponto.x) //5
 }
```

Sintaxe

- Definindo getter e setter

```
class Ponto(private var _x: Int,private var _y:
Int){
   def x :Int = _x
   def x_=(x: Int) {
     if (x < 0) this._x = x * (-1)
   }
 }

 def main(args:Array[String]){
   var ponto = new Ponto(5,4);
   println(ponto.x) //5
 }
```

Sintaxe

- Definindo nossos próprios construtores
- Utilizamos a palavra-chave this.

```
class Pessoa {
   var nome = "Fulano"
   var idade = 18
   def this(nome: String){
     this()
     this.nome = nome
   }
   def this(nome: String, idade: Int){
     this(nome)
     this.idade = idade
   }
   override def toString = {
     nome + " tem " + idade + " anos"
   }
 }

 def main(args:Array[String]){
   println(new Pessoa) //Fulano tem 18 anos
   println(new Pessoa("Pedro")) //Pedro tem 18 anos
   println(new Pessoa("Pedro", 20)) //Pedro tem 20 anos
 }
```

Sintaxe

- Modificadores de acesso

```
private var nome = "Fulano"

protected var idade = 18

var tamanho = 182 //public
```

- **protected:** visibilidade dentro do package, subclasses
- **private:** visibilidade apenas dentro da própria classe

Sintaxe

- Declarando e chamando funções
- Tipo do parâmetro é obrigatório
- Retorno pode ser omitido

```
def addInt(a:Int, b:Int):Int = {
   var sum:Int = 0
   sum = a + b
   return sum
 }

 def hello() : Unit = {
   println("Hello World")
 }

 def main(args:Array[String]){
   println(addInt(4,5)) //9
   hello //Hello World
 }
```

- **def** functionName ([list of parameters]) : [return type]

Identificadores

- Identificadores podem ter duas formas. Eles ou iniciam por uma letra, que é seguida por uma sequência (possivelmente vazia) de letras ou símbolos, ou podem iniciar com um caractere operador, seguido por uma sequência (possivelmente vazia) de caracteres operadores. Ambas as formas podem conter caracteres “underscore” ‘_’. Além disso, um caracter underscore pode ser seguido por quaisquer identificadores.

Identificadores válidos

x

Room10a

+

--

foldl_:

+_vector

Palavras reservadas

- Os seguintes são palavras reservadas, e não devem ser usadas como identificadores

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

Amarrações

- Amarrações são estáticas
- Inferência de tipo
- Uma amarração em um escopo mais interno faz sombra em amarrações em escopos mais externos.

```
var nome = "Fulano" //nome: String =
Fulano

var idade = 20

def printaIdade(idade:Int) = {
   println(idade)
 }

def main(args:Array[String]){
  printaIdade(18)
 }
```

Tipos de dados

- Todos os valores em Scala são objetos. Dado que Scala é baseada em classes, todos os valores são instâncias de uma classe.

Tipos de dados

Variáveis E Constantes

Variáveis

- São definidas da seguinte forma:

  var <nomeDaVariavel> : <Tipo> = <valor>

- É opcional declarar explicitamente o tipo da variável, pois a linguagem possui mecanismo de inferência de tipo.

Variáveis

```
// Variáveis com tipo explícito
var nome: String = "Matheus"
var idade: Int = 20
var harrypotter: Livro = new Livro("Harry Potter")

// Variáveis com tipo inferido
var professor = "Vitor Souza"
// Resulta em -> professor: String = Vitor Souza
```

Variáveis

- Além de valores, variáveis e constantes também podem armazenar funções anônimas:

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

Variáveis

- Variáveis (var) podem ser atualizadas a outros valores do mesmo tipo:

```
var nome: String = "Matheus"
var idade: Int = 21
nome = "Pedro"
idade = 20
```

Variáveis

- Mas não podem ser atualizadas por valores de outro tipo

```
var nome: String = "Matheus"
nome = 42
// error: type mismatch;
//  found    : Int(42)
//  required: String
```

Atualização de tipos compostos

- Uma variável de tipo composto pode ser atualizada por algo de mesmo tipo:

```
case class Livro(var nome: String, var autor: String)
var livro = Livro(nome = "A Game of Thrones", autor="George Martin")
livro = Livro(nome="Pai Rico, Pai Pobre", autor="Robert e Sharon")
```

Atualização de tipos compostos

- As propriedades de uma váriavel de tipo composto podem ser atualizadas, se essas também forem variáveis.
- No caso desse tipos serem classes, ao atribuir um valor novo a uma propriedade, o setter é chamado.

```
livro.nome = "A Origem das Espécies"
```

Atualização de tipos compostos

- Porém, isso não é possível no caso de alguma propriedade ser constante (indicada por "val")

```
case class Cor(val hex: String)
var cor = Cor("#ffbbaa")
cor.hex = "#aabbff"
//error: reassignment to val
//       cor.hex = "#aabbff"
```

Constantes

- Scala também pode possuir dados contantes. Dados que não podem ser alterados depois de definidos

```
val imutavel = List(6,4,54,5)
imutavel = imutavel.sorted
// error: reassignment to val
//  imutavel = imutavel.sorted
```

Constantes

- Constantes também podem armazenar funções anônimas.

```
val martingale = (x: Int) => x*2
val galemartin: Int => String = (x: Int) => s"Numero: $x"

galemartin(3) // Numero: 3
martingale(127) // 254
```

Atribuição múltipla

- Constantes e váriaveis permitem atribuição múltipla.

```
val (x, y) = (10.6, 7.5)

println(x) // 10.6
println(y) // 7.5
```

Atribuição múltipla

- Porém, esse mecanismo não podem ser utilizado para variáveis já definidas.

```
var m = 10
var n = 22
(m,n) = (22,10)
// error: ';' expected but '=' found.
// Acontece porque (m,n) é interpretado como "new
Tuple(m,n)" para variáveis já existentes.
```

Constantes em classes

- Quando um atributo é definido como uma constante em uma classe, é criado automaticamente um getter, mas não um setter.

```
case class Pessoa(var nome: Int, val nacionalidade: String)
var matheus = Pessoa("Matheus", "Brasileiro")
matheus.nome = "Matthew" // Permitido
matheus.nacionalidade = "Canadense" // Não permitido
```

Armazenamento de variáveis e constantes

Em Memória Principal

- Scala não possui primitivos

  Scala não possui tipos primitivos no mesmo sentido de Java, ela possui os tipos "pseudo-primitivos" encapsulados que herdam de AnyVal. São esses:

  Short, Int, Long, Double, Float, Boolean, Char and Byte. Todos os outros herdam de AnyRef.

Em Memória Principal

- Scala otimiza em tempo de compilação para tornar os tipos descendentes de AnyVal em tipos primários como os de Java, e estes serão armazenados na Pilha.
- Já os tipos que herdam AnyRef vão ser armazenados no Monte, e sua referência será armazenada na Pilha.

Serialização

- Scala possui suporte a serialização
- Isso permite o armazenamento de instâncias de objetos em memória secundária.
- Sintaxe ligeiramente diferente de Java.
- Precisamos explicitar que uma determinada classe possui suporte a serialização.

Serialização

Caso a classe não utilize herança, podemos simplemente herdar a trait Serializable

```
@SerialVersionUID(100L) // Identificador da versao da classe
class Produto(var nome: String, var preco: BigDecimal) extends
Serializable {
     override def toString() : String =  s"Nome: $nome / Preco:
$preco"
}
```

Serialização

Caso a classe já utilize alguma herança, utilizamos a trait Serializable com palavra-chave **with** (semelhante ao **implements** de Java)

```
@SerialVersionUID(120L) // Identificador da versão da classe
class Produto(var nome: String, var preco: BigDecimal) extends
Coisa with Serializable {
    // ....
}
```

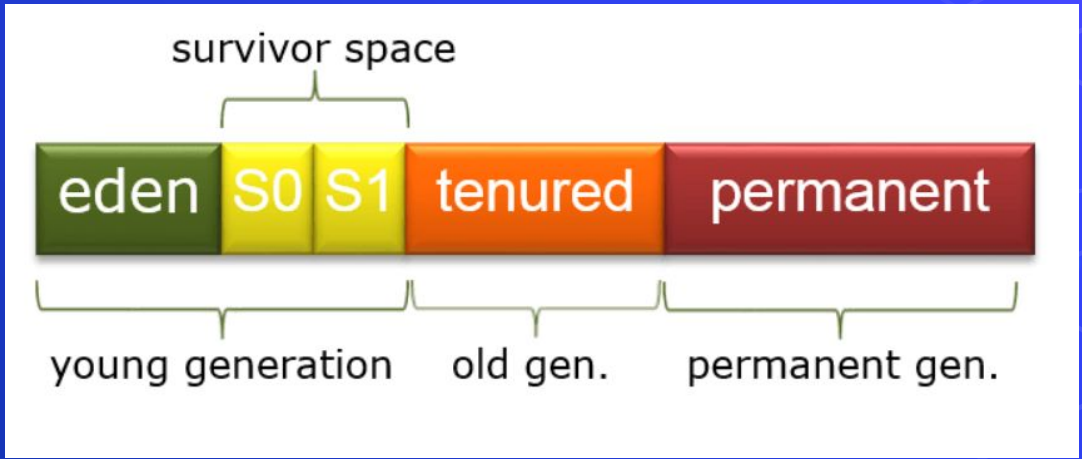
```
object MainSerializacao extends App {
  // (1) cria nova instância
  val cocacola = new Produto("Cola Cola 2L", BigDecimal(95.00))
  // (2) escreve a instância para um arquivo
  val outputstream =
        new ObjectOutputStream(new FileOutputStream("./cocacola"))
  outputstream.writeObject(cocacola)
  outputstream.close

 // continua...
```

```
object MainSerializacao extends App {
    //...
    // (3) Carrega a instancia para a memoria
    val inputstream = new ObjectInputStream(new
FileInputStream("./cocacola"))
    val cocaCarregada =
            inputstream.readObject.asInstanceOf[Produto]
    inputstream.close
    // (4) Imprime o objeto que foi carregado pra memória
principal
    println(cocaCarregada)
    // Nome: Cola Cola 2L / Preco: 95.0
```

Garbage Collector

- Mesmo mecanismo de coleta de lixo do Java.
- Coleta de Lixo Generacional
- Minor GC vs Major GC



Expressões e Comandos

Literais

- Int
- Long
- Double
- Float

```
var x = 23
>> x: Int = 23
var y = 0x17
>> y: Int = 23
var z = 23L
>> z: Long = 23
var w = 0x17L
>> w: Long = 23
var h = 23.0
>> h: Double = 23.0
var j = 23.0F
>> h: Float = 23.0
```

Literais

- Char
- String

```
var w = 'S'
>> w: Char = S
var x = '\u0051'
>> x: Char = Q
var y = "Scala"
>> y: String = Scala
var z =
 """Scala
   é
   muito
   legal
   """
>> z: String =
"Scala //1
é //2
muito //3
legal //4
"
```

Artimética

- Operadores +, -, *, / e %

```
var x = 4 + 2
>> x: Int = 6
var y = 4 - 2
>> y: Int = 2
var z = 4 * 2
>> z: Int = 8
var w = 5 / 2
>> w: Int = 2
var k = 4 % 2
>> k: Int = 0
```

Relacionais

- Operadores >, >=, !=, ==
- Por padrão compara as referências
- Pode ser sobreescrito

```
class Scala{ }

var s1 = new Scala
var s2 = new Scala

s1 == s2

>>defined class Scala

>> s1: Scala = Scala@109c42d8
>> s2: Scala = Scala@1a5a5d61

>>res0: Boolean = false

var x = List(1,2,3,4)
var y = List(1,2,3,4)

x == y
>> res0: Boolean = true
```

Relacionais

- Operadores >, >=, !=, ==
- Por padrão compara as referências
- Pode ser sobreescrito

```
class Scala{ }

var s1 = new Scala
var s2 = new Scala

s1 == s2

>>defined class Scala

>> s1: Scala = Scala@109c42d8
>> s2: Scala = Scala@1a5a5d61

>>res0: Boolean = false

var x = List(1,2,3,4)
var y = List(1,2,3,4)

x == y
>> res0: Boolean = true
```

Booleanas e Binárias

- Operadores booleanos:
  &&, &, ||, | e !
- Operadores binários: &, |,
  >>, >>>, <<, <<< e ~

Condicionais

- If, else e case
- Case: parecido com switch case com adição de pattern matching

```
"s" match {
 case 2 => println("Não imprime")
 case _ => println("Scala é legal")
 case "s" => println("Imprime?")
}

>> case 2 => println("Não imprime") //1
^
On line 2: error: type mismatch;
found   : Int(2)
required: String

"s" match {
 case "2" => println("Não imprime")
 case _ => println("Scala é legal")
 case "s" => println("Imprime?")
}

>> case _ => println("Scala é legal") //2
^
On line 3: warning: patterns after a variable
pattern cannot match (SLS 8.1.1)
case "s" => println("Imprime?") //3
```

Referenciamento

- Não possui os operadores: ->, * e &
- [ ] usado ao definir tipos genéricos

Categóricas

- “to” usado para converter
- getClass retorna objeto java.lang.Class

Operadores

- Em Scala os operadores são na verdade funções
- Primeira linguagem a usar esse conceito foi Smalltalk
- Todo operador entre expressões são chamadas de funções

```
var x = 1
var y = 2

x.+(y)

>> res0: Int = 3


/** Returns the sum of this value and `x`. */
def +(x: Int): Int


class Scala{
 def +(any: Any) = println(any)
}

var s = new Scala
s.+("Sobreescrevendo operador")

>> Sobreescrevendo operador
```

Comandos Iterativos

- “To” usado para definir o objeto Range.Inclusive usado para iterar um for
- Operador <- chama uma função

```
for(i <- 1 to 5){
 print(i)
}

>> 1 2 3 4 5
```

```
final override def foreach[@specialized(Unit) U](f: Int => U):
Unit = {
 if (!isEmpty) {
   var i = start
   while (true) {
     f(i)
     if (i == lastElement) return
     i += step
   }
 }
}
```

Comandos Iterativos

- Podemos inserir uma condição no for

```
for(i <- 1 to 5 if i < 3){
 print(i + " ")
}

>> 1 2
```

Comandos Iterativos

- break simulado com funções e try catch

```
import util.control.Breaks._

breakable(
 for( i <- 1 to 10){
   print(i + " ")
   if (i > 5) break
 }
)
>> 1 2 3 4 5 6

def breakable(op: => Unit): Unit = {
 try {
   op
 } catch {
   case ex: BreakControl =>
     if (ex ne breakException) throw ex
 }
}

def break(): Nothing = { throw breakException }
}
```

Precedência

() []	Esquerda para direita
! ~	Direita para esquerda
* / %	Esquerda para direita
+ -	Esquerda para direita
>> >>> <<	Esquerda para direita
> >= < <=	Esquerda para direita
== !=	Esquerda para direita

Precedência

&	Esquerda para direita
^	Esquerda para direita
|	Esquerda para direita
&&	Esquerda para direita
||	Esquerda para direita
= += -= /= %= >>= <<= &= ^= |=	Esquerda para direita
,	Esquerda para direita

Modularização

Pacotes

- Pode inserir arquivos em um pacote colocando o nome do pacote no topo do arquivo
- Não necessita corresponder aos diretórios mas é uma boa prática

```
package principal

class Principal {

}
```

Pacotes

- Permite pacotes aninhados
- A sintaxe mais parecida com Java na verdade é somente synctatic sugar

```
package com.principal{

//pacote com.principal.usecases
package usecases{
  class Incluir{}

}

//pacote com.principal.tests
package tests{
  class IncluirTest{}

}
}
```

Pacotes

- Devido ao aninhamento de pacotes é possível que pacotes mais externos sejam sombreados

```
package launch {
 class Booster3
}

// In file bobsrockets.scala
package bobsrockets {
 package navigation {
  package launch {
   class Booster1
  }
  class MissionControl {
   val booster1 = new launch.Booster1
   val booster2 = new bobsrockets.launch.Booster2
   val booster3 = new _root_.launch.Booster3
  }
 }
 package launch {
  class Booster2
 }
}
```

Importes

- Similar a java
- Podemos importar somente uma classe utilizando seu nome ou todas utilizando
- “_”

```
// easy access to Fruit
import bobsdelights.Fruit

// easy access to all members of bobsdelights
import bobsdelights._

// easy access to all members of Fruits
import bobsdelights.Fruits._
```

Importes

- O importe não necessita estar no início do arquivo
- Equivalente a fruit.name e fruit.color

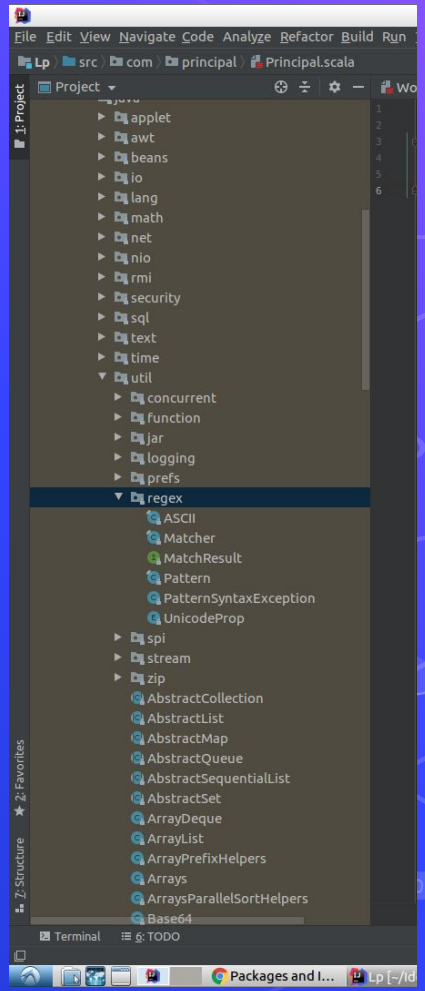
```
def showFruit(fruit: Fruit) {
 import fruit._
 println(name +"s are "+ color)
}
```

Importes

- Permite realmente importar o pacote e não necessariamente o seu conteúdo

```
import java.util.regex

class AStarB {
 // Accesses java.util.regex.Pattern
 val pat = regex.Pattern.compile("a*b")
}
```



Importes

- Permite importar objetos específicos e alterar o seu nome
- Permite importar todo o pacote exceto uma classe específica

```
import Fruits.{Apple => McIntosh, Orange}


import Fruits.{Pear => _, _}
```

Importes

- 3 importes implícitos
- Importar em Scala é mais flexível
- A própria linguagem resolve conflitos de imports ao contrário de C

```
import java.lang._ // everything in the java.lang package
import scala._     // everything in the scala package
import Predef._    // everything in the Predef object
```

Modificadores de Acesso

- Private e Protected
- Similar a Java
- Protected somente permite a visualização às subclasses e não a classes do mesmo pacote
- Não existe um modificador Public

```
import java.lang._ // everything in the java.lang package
import scala._     // everything in the scala package
import Predef._    // everything in the Predef object


package p {
 class Super {
   protected def f() { println("f") }
 }
 class Sub extends Super {
   f()
 }
 class Other {
   (new Super).f()  // error: f is not accessible
 }
}
```

Modificadores de Acesso

- Podem ser utilizados com a sintaxe private[X] ou protected[X]. Significa dar visibilidade ao escopo em que X se encontra

Modificadores de Acesso

- private[this]

```
class A {
 private[this] def func = println("func")
 def func2(x: A) = println(x.func)
}
/**Da erro por func só é visto pelo mesmo objeto */
```

Funções e Closures

- Parâmetros passados como “val”
- Unidirecional de entrada
- Retorno da função é a sua última linha. Deve-se evitar o uso de return

```
def sum(x: Int) = x = x+1

error: reassignment to val
```

Funções e Closures

- Avaliação feita na hora da chamada da função

```
def retorna(x: Int, z: Int, w: Int): Int ={
 if(true){
   if(true){
    x
   }else{
     z
   }
 }else{
   w
 }
}

def printAndReturn(n : Int): Int ={
 print(n + " ")
 n
}

retorna(
 printAndReturn(1),
 printAndReturn(2),
 printAndReturn(3)
)

>> 1 2 3
>> res0: Int = 1
```

Funções e Closures

- É possível usar avaliação lazy com : =>

```
def retorna(x: => Int, z: => Int, w: => Int): Int ={
 if(true){
   if(true){
    x
   }else{
     z
   }
 }else{
   w
 }
}

def printAndReturn(n :  Int): Int ={
 print(n + " ")
 n
}

retorna(
 printAndReturn(1),
 printAndReturn(2),
 printAndReturn(3)
)

>> 1
>> res0 : Int = 1
```

Funções e Closures

- Métodos: funções como parte de um objeto
- Local functions: funções definidas dentro de outras funçõe. Somente são visíveis no escopo em que foram definidas

```
class A {
 def metodo: Unit = {
   println("Eu sou um metodo")

   def localFunction ={
     println("Eu sou uma local function")
   }
 }
}
```

Funções e Closures

- First-class functions: podem ser definidas e passadas como valores
- Operador =>

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

>> x: Int => Int = <function>


var x = (x: Int) => {
 println("Multiline function")
 x + 1
}
```

Funções e Closures

- Partially applied functions
- x é um objeto criado automaticamente. x(1) equivale a x.apply(1)
-

```
def sum(x:Int, y:Int, z:Int) = x + y + z

var x = sum(1, _: Int, 3)
>> x: Int => Int = <function>

x(1)
>> res0: Int = 5
```

Funções e Closures

- more é uma variável livre diferente de x que é amarrada ao parâmetro
- O objeto criado para func é chamado de closure

```
(x: Int) => x + more
>> error: not found: value more

var more = 2
var func = (x: Int) => x + more
func.apply(1)

>> res0: Int = 3

more = 5
func.apply(1)
>> res1: Int = 6
```

Funções e Closures

- Funções com argumentos variáveis
- Tail recursion: compilador Scala reconhece chamadas recursivas e realiza otimizações

```
def echo(args: String*) =
 for (arg <- args) print(arg + " ")

echo("Scala", "Java", "C", "C++")

>> Scala Java C C++
```

```
def approximate(guess: Double): Double =
 if (isGoodEnough(guess)) guess
 else approximate(improve(guess))
```

Classes, Objects Traits e Case Classes

Classes

- As classes em Scala são muito semelhantes as classes em Java. E elas são definidas da seguinte forma:

```
class Animal {
    // Atributos e métodos
}
```

Classes

- Uma classe pode possuir um contrutor principal:

```
class Animal(var especie: String, idade: Int) {
    // Atributos e métodos
}
```

Classes

- E também outros construtores auxiliares:

```
class Animal(var especie: String, var idade: Int) {
    var nome = ""
    def this(especie: String,idade: Int, nome: String) =  {
        this(especie, idade)
        this.nome = nome
    }
}
```

Classes

- E poderá criar as instâncias com qualquer um desses construtores:

```
var animal1 = new Animal("sapiens", 20, "homo")
var animal2 = new Animal("lupus", 8)
println((animal1.especie, animal1.idade, animal1.nome))
// (sapiens, 20, homo)
println((animal2.especie, animal2.idade, animal1.nome))
// (lupus,8,)
```

Classes

- Atributos variáveis: Getter e Setter
- Atributos constantes: Somente Getter

```
class Livro {
    var nome: String = "" // Possui getter e setter
    val autor: String = "" // Só possui getter
}
var livro = new Livro()
println(livro.nome) // acesso encapsulado pelo getter
livro.nome = "Harry Potter" // atribuição encapsulada pelo setter
```

Herança

- Classes podem herdar de outras classes e traits.
- Atributos da classe pai são automaticamente

```
class Pai { var atributo1 = "atributo1" }
class Filho extends Pai { var atributo2 = "atributo2" }
val filho = new Filho()
println((filho.atributo1, filho.atributo2))
//(atributo1,atributo2)
```

Herança

- Quando a classe possui atributos no construtor, é necessário declarar de maneira um pouco diferentes:

```
class Cachorro(especie: String, idade: Int, var raca: String)
extends Animal(especie, idade) {
}
var cao1 = new Cachorro("Canis Lupus", 20, "Bulldog");
println((cao1.especie, cao1.idade, cao1.raca))
// (Canis Lupus, 20, Bull Dog)
```

Herança

- Quando a classe possui atributos no construtor, é necessário declarar de maneira um pouco diferentes:

OBS: Um detalhe a se perceber é que "**especie**" e "**idade**" não possuem a palavra-chave **var**, isso acontece porque esses são atributos do pai, onde ele já foi declarado, e portanto, o seu getter e setter já foram criados.

Objetos

- Scala possui um tipo de classe especializado chamado **Object**. Eles são classes que só pode ter uma instância, um singleton. Ele é similar a uma classe em java que só possua atributos constantes e métodos estáticos.

Objetos

Um objeto é declarado da seguinte forma:

```
object <identificador> [extends <identificador>] {
   // Atributos e métodos
}
```

E ele é instanciado pela primeira e única vez quando algum de seus membros é acessado.

Objetos

- Utilidade:
  - Coleção de funções puras (sem efeitos colaterais)
  - Coleção de funções que trabalham com I/O

```
object HtmlUtils {
    def removeMarkup(input: String) = {
        input
        .replaceAll("""</?\w[^>]*>""","")
        .replaceAll("<.*>","")
    }
}
val html = "<html><body><h1>Introduction</h1></body></html>"
val text = HtmlUtils.removeMarkup(html)
// text: String = Introduction
```

Companion Objects e Factory

Outra forma com que objetos são utilizados normalmente, é implementar o padrão de **fábrica**, onde o objeto é responsável por gerar instâncias de uma classe. Para isso, ele se utiliza do conceito de Companion Object e do método apply()

Companion Objects

- Companion Objects são objetos que possuem o mesmo nome de uma classe no mesmo projeto.

- Companion Objects podem acessar todos os atributos de uma classe, inclusive os privados, e vice versa.

```scala
class Pessoa (var nome: String) {
    override def toString(): String = {
        Pessoa.formatar(nome)
    }
}
object Pessoa {
    private def formatar(nome: String) = {
        val nomes = nome.split(" ")
        s"${nomes.head} ${nomes.last}"
    }
}
var pessoa = new Pessoa("Matheus Vicente Delunardo")
println(pessoa) // Matheus Delunardo
```

Método apply()

- É o método padrão que é chamado quando se passa parâmetros para um objeto:

```
object CriadorDeTupla {
    def apply(x: Int, y: Int, z : int) {
        (x,y,z)
    }
}
val tupla = CriadorDeTupla(1,2,3)
// (1,2,3)
```

Padrão de Fábrica

- Podemos então implementar o padrão de projeto de **fábrica** utilizando desses dois conceitos, da seguinte forma de forma conjunta.

```scala
class Filme(var nome: String, var genero: String) {
    override def toString (): String = {
        s"Nome: $nome / Genero: $genero"
    }
}
object Filme {
    val todosFilmes = List(("Split", "Suspense"), ("Glass", "Suspense"))
    def apply(idx: Int): Filme = {
        if(idx < 0 || idx > todosFilmes.length - 1) {
           new Filme("Nao disponível", "Indefinido")
        } else {
            new Filme(todosFilmes(idx)._1, todosFilmes(idx)._2)
        }
    }
}
```

Padrão de Fábrica

- Depois de definirmos a classe e o companion object com o método apply, podemos utilizar da seguinte forma:

```
val split = Filme(1)
println(split)
// Nome: Glass / Genero: Suspense
val naoexiste = Filme(2)
println(naoexiste)
// Nome: Nao disponível / Genero: Indefinido
```

Case Classes

- As Case Classes são um tipo de classe que já possui vários métodos úteis implementados, além de possuir seu próprio companion object automático.

```
case class NomeDaClasse (var atributo: String) extends
OutraClasse {   // Atributos e Métodos  }
// Sao instanciadas sem a palavra-chave new, porque na
realidade está chamando o método apply do companion object
val instancia = NomeDaClasse("atributo")
```

Case Classes

Os métodos criados automaticamente nas case classes são os seguintes:

- apply
- copy
- equals
- hashCode
- toString

São úteis para armazenamento de dados como se fossem tipos estruturados.

Traits

São tipos especializados de classes semelhante a interfaces em Java, porém com algumas características extras:

- Permite a implementação parcial de métodos
- Permite a herança múltipla

Traits não podem ser instanciadas, e portanto, também não podem possuir construtores.

Traits

- As traits são declaradas da seguinte forma:

```
trait Caracteristica extends OutraCaracteristica with
AlgumComportamento {
    // Atributos e métodos
}
```

Traits

Uma vantagem das traits é o fato de poderem ser utilizada para adicionar comportamentos ou características a outras traits ou classes, incentivando a reusabilidade de código. Isso é chamado de **mixin**.

É possível adicionar múltiplos mixins a uma outra classe/trait usando a keyword **with**.

Isso é semelhante ao *implements* de Java, porém, traits podem ter seus métodos já implementados.

```
trait HtmlUtils {
    def removeMarkup(input: String) = {
          input
            .replaceAll("""</?\w[^>]*>""","")
            .replaceAll("<.*>","")
    }
}
class Page(val s: String) extends HtmlUtils {
    def asPlainText = removeMarkup(s)
}
new
Page("<html><body><h1>Introduction</h1></body></html>").asPlainText
// Introduction
```

```scala
trait Carnivoro {
    def mastigar(): Unit = println("*Mastigando Carnivoramente*")
    def comerCarne() =  println("*Comendo Carne*")
}
trait Herbivoro {
    def mastigar() : Unit = println("*Mastigando Herbovoramente*")
    def comerVegetal() = println("*Comendo Vegetal*")
}
class Humano (var nome: String) extends Carnivoro with Herbivoro {
    override def mastigar(): Unit = println("*Mastigando Onivoramente*")
}
val humano = new Humano("Otavio")
Humano.mastigar // *Mastigando Onivoramente*
humano.comerCarne //*Comendo Carne*
humano.comerVegetal //*Comendo Vegetal*
```

Polimorfismo

Polimorfismo

Scala possui suporte aos 4 tipos de polimorfismo:

- Ad-hoc
  - Polimorfismo de Coerção
  - Polimorfismo de Sobrecarga
- Universal
  - Polimorfismo Paramétrico
  - Polimorfismo de Inclusão

Polimorfismo de Coerção

- Scala suporta a coerção na atribuição, em seus operadores e também em funções
- Assim como Java, Scala somente suporta a coeção implicita de tipos "menores" para tipos "maiores". Ou seja, a ampliação.

```
var valorByte : Byte = 12
var valorInt : Int = 123444

var maior : Int = valorByte
// maior: Int = 12
var menor : Byte = valorInt
// error: type mismatch; found: Int; required: Byte

var novoInteiro : Int = valorByte + valorInt
// inteiro: Int = 123456
var novoByte : Byte = valorByte + valorInt
// error: type mismatch; found: Int; required: Byte
```

```scala
// Polimorfismo nos parâmetros das funções
// *Funciona*
def area (a: Double, b: Double) : Double = a * b
var altura : Int = 12
var largura : Int = 24
val minhaArea = area(altura,largura)
// minhaArea: Double = 288.0

// *Não funciona*
def area (a: Int, b: Int) : Int = a * b
val minhaArea = area(12.0, 24.0)
// error: type mismatch; found: Double(24.0); required: Int
```

Polimorfismo de Sobrecarga

- A sobrecarga de métodos é Independente de Contexto:
  - Métodos diferenciados pelos parâmetros, não pelo retorno.

OBS: Scala suporta sobrecarga de operadores.

```scala
object Util {
    def sum (a: Int, b: Int) = {
        a + b
    }
    def sum (a: List[Int], b: List[Int]) = {
         (a zip b).map((x) => x._1 + x._2)
    }
}
var somaDeInteiros = Util.sum(13,23)
// somaDeInteiros: Int = 36
var somaDeVetores = Util.sum(List(1,2,3), List(3,4,5))
// somaDeVetores: List[Int] = List(4, 6, 8)
```

```
class Complexo(val real: Double, val imaginario: Double ) {
    def +(that: Complexo) = new Complexo(this.real + that.real,
this.imaginario + that.imaginario)

    def -(that: Complexo) = new Complexo(this.real - that.real,
this.imaginario - that.imaginario)

    override def toString = real + " + " + imaginario + "i"
}
var a = new Complexo(4.0,5.0)
var b = new Complexo(2.0,3.0)
println(a)  // 4.0 + 5.0i
println(a + b)  // 6.0 + 8.0i
println(a - b)  // 2.0 + 2.0i
```

Polimorfismo Paramétrico

- Scala permite o polimorfismo paramétrico por meio de tipos genéricos.

```
class Pilha[T] {
    var elementos : List[T] = List[T]()
    def push (x: T) { elementos = x :: elementos}
    def top(): T = elementos.head
    def pop() { elementos = elementos.tail }
}
var p = new Pilha[Int]()
p.push(1)
p.push(2)
p.top // 2
p.pop
p.top // 1
```

Polimorfismo de Inclusão

- Como uma linguagem orientada a objeto, Scala implementa o conceito de herança. O que permite o polimorfismo de inclusão.

```
class Poligono(var arestas: Int) {
    // ...
}
class Retangulo(arestas: Int, var largura : Int, var altura :
Int ) extends Forma(arestas) {
    // ...
}
class Hexagono(arestas : Int, var lado: Int) extends Forma {
    // ...
}

var hexagono : Poligono = new Hexagono(6, 10)
```

Polimorfismo de Inclusão

- É possível fazer herança múltipla utilizando-se das traits, como já foi demonstrado anteriormente.
- Isso introduz o problema do "diamante", que scala resolve por meio da linearização da herança.

```scala
trait SerVivo {
    def info () = println("Eu sou vivo")
}
trait Carnivoro extends SerVivo{
    override def info () = println("Eu como carne")
}
trait Herbivoro extends SerVivo{
    override def info () = println("Eu como vegetais")
}
object Humano extends Carnivoro with Herbivoro
object Humano2 extends Herbivoro with Carnivoro

var humano = Humano
var humano2 = Humano2

humano.info // Eu como vegetais
humano2.info // Eu como carne
```

A linearização da herança de "Humano" é:
Humano -> Herbivoro -> Carnivoro -> SerVivo

E de "Humano2" é:
Humano -> Carnivoro -> Herbivoro -> SerVivo

Amarração Tardia

- Scala utiliza o conceito de amarração tardia, onde o tipo é descoberto em tempo de execução.

```
class Pessoa (var nome: String) {
    def introduzir () = println(s"Meu nome eh $nome")
}
class Doutor(nome: String) extends Pessoa(nome) {
    override def introduzir () = println(s"Eu sou o Dr. $nome")
}
var listaDePessoas : List[Pessoa] =
   List(new Pessoa("Matheus"), new Doutor("Vitor"), new Doutor("Verejao"), new Pessoa("Pedro"))
listaDePessoas.foreach((pessoa) => pessoa.introduzir)
```

```
// Meu nome eh Matheus
// Eu sou o Dr. Vitor
// Eu sou o Dr. Verejao
// Meu nome eh Pedro
```

Exceções

Uso básico

- Exceções são objetos
- Possui todas as exceções de Java
- Scala não possui exceções checadas
- Dois meios principais:
  - try / catch / finally
  - scala.util.Try

try / catch / finally

- Mecanismo básico de exceções
- Também são expressões

```
try {
    metodoQueJogaExcecao()
} catch {
    case e: FileNotFoundException => println("Arquivo não encontrado")
    case e: IOException => println("Erro de escrita/leitura")
    case _: Throwable => println("Alguma outra exceção")
} finally {
    // SEMPRE executa depois do bloco try/catch
}
```

scala.util.Try

- Menos comum que o try / catch
- Padrão funcional
- Exceções não fatais não são capturadas

```
def funcao(s: String): Try[Int] = {
    Try(s.toInt)
}

scala> funcao("30")
res0: scala.util.Try[Int] =
Success(30)

scala> funcao("foo")
res1: scala.util.Try[Int] =
Failure(java.lang.Number(...))
```

```scala
import scala.util.{Try,Success,Failure}
def stringToInt(s: String): Try[Int] = {
    Try(s.toInt)
}
var possivelNumero : Try[Int] = stringToInt("12dds34")
possivelNumero match {
    case Success(i) =>
        println(s"O numero eh ${possivelNumero.get}")
    case Failure(e) =>
        println("O valor dado nao pode ser convertido para um
inteiro")
        println("Informações do erro: " + e.getMessage)
}
```

Concorrência

Concorrência

- Nativa na linguagem
- Facilitada pelo fato de a maioria das estruturas de dados serem imutáveis
- Future
- Promises
- Pode-se também usar as funcionalidades já existentes em Java (e.g., Thread)

Future

- Mecanismo assíncrono não bloqueante
- Utiliza callbacks

```
val f: Future[List[String]] = Future {
    session.getRecentPosts
}

f onComplete {
    case Success(posts) => println(posts)
    case Failure(e) => println(e.getMessage)
}
```

Promise

- Completam Futures
- success só pode ser chamado uma vez por promise

```
val p = Promise[T]()
val f = p.future

val producer = Future {
  val r = produceSomething()
  p success r
  continueDoingSomethingUnrelated()
}

val consumer = Future {
  startDoingSomething()
  f foreach { r =>
    doSomethingWithResult()
  }
}
```

Avaliação

Critérios gerais	C	C++	Java	Scala
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estruturado e OO	OO	Funcional e OO
Evolutibilidade	Não	Parcial	Sim	Sim
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Critérios gerais	C	C++	Java	Scala
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	
Aprendizado	Não	Não	Não	
Eficiência	Sim	Sim	Parcial	
Portabilidade	Não	Não	Sim	
Método de projeto	Estruturado	Estruturado e OO	OO	Funcional e OO
Evolutibilidade	Não	Parcial	Sim	Sim
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Não oferece ao programador o controle direto ao hardware

Critérios gerais	C	C++	Java	Scala
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	
Eficiência	Sim	Sim	Parcial	
Portabilidade	Não	Não	Sim	
Método de projeto	Estruturado	Estruturado e OO	OO	Funcional e OO
Evolutibilidade	Não	Parcial	Sim	Sim
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Fortemente tipada, possui exceções, coleta de lixo, etc.

Critérios gerais	C	C++	Java	Scala
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	**Não**	**Não**	**Não**	**Não**
Eficiência	Sim	Sim	Parcial	
Portabilidade	Não	Não	Sim	
Método de projeto	Estruturado	Estruturado e OO	OO	O
Evolutibilidade	Não	Parcial	Sim	Sim
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Sintaxe e conceitos incomuns e complexos, e nem sempre é ortogonal

Critérios gerais	C	C++	Java	Scala
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	**Sim**	**Sim**	**Parcial**	**Parcial**
Portabilidade	Não	Não	Sim	
Método de projeto	Estruturado	Estruturado e OO		O
Evolutibilidade	Não	Parcial		
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Para uma maior confiabilidade, controla alguns aspectos que diminuem a eficiência

Critérios gerais	C	C++	Java	Scala
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	**Não**	**Não**	**Sim**	**Sim**
Método de projeto	Estruturado	Estruturado e OO		
Evolutibilidade	Não	Parcial		
Reusabilidade	Parcial	Sim		
Integração	Sim	Sim	Parcial	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Roda tanto na JVM quanto na CLR (.NET), plataformas portáveis

Critérios gerais	C	C++	Java	Scala
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	**Estruturado**	**Estruturado e OO**	**OO**	**Funcional e OO**
Evolutibilidade	Não	Parcial	Sim	Sim
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Critérios gerais	C	C++	Java	Scala
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estruturado e OO	OO	Funcional e OO
Evolutibilidade	**Não**	**Parcial**	**Sim**	**Sim**
Reusabilidade	Parcial	Sim	Si	
Integração	Sim	Sim	Pa	
Custo	Depende da ferramenta	Depende da ferramenta	Depe ferra	

Os métodos de projeto estimulam códigos legíveis

Critérios gerais	C	C++	Java	Scala
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não		
Método de projeto	Estruturado	Estruturado e OO		
Evolutibilidade	Não	Parcial	Sim	
Reusabilidade	**Parcial**	**Sim**	**Sim**	**Sim**
Integração	Sim	Sim	Parcial	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Classes e pacotes que auxiliam na reusabilidade

Critérios gerais	C	C++	Java	Scala
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estruturado e OO		O
Evolutibilidade	Não	Parcial		
Reusabilidade	Parcial	Sim	Sim	
Integração	**Sim**	**Sim**	**Parcial**	**Parcial**
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Interoperável com Java, mas é necessário o JNI (Java Native Interface) para outras linguagens

Critérios gerais	C	C++	Java	Scala
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estruturado e OO	OO	Funcional e OO
Evolutibilidade	Não	Parcial	S	
Reusabilidade	Parcial	Sim	S	
Integração	Sim	Sim	Parcial	Parcial
Custo	**Depende da ferramenta**	**Depende da ferramenta**	**Depende da ferramenta**	**Depende da ferramenta**

Sem custo, mas pode haver ferramentas pagas

Critérios específicos	C	C++	Java	Scala
Escopo	Sim	Sim	Sim	Sim
Expressões e comandos	Sim	Sim	Sim	Sim
TIpos primitivos e compostos	Sim	Sim	Sim	Sim
Gerenciamento de memória	Programador	Programador	Sistema	Sistema
Persistência dos dados	Biblioteca de funções	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	Biblioteca de classes e serialização
Passagem de parâmetros	Lista variável e por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência	Lista variável, por nome e por cópia de referência

Critérios específicos	C	C++	Java	Scala
Encapsulamento e proteção	Parcial	Sim	Sim	Sim
Sistema de tipos	Não	Parcial	Sim	Sim
Verificação de tipos	Estática	Estática / Dinâmica	Estática / Dinâmica	Estática / Dinâmica
Polimorfismo	Coerção e sobrecarga	Todos	Todos	Todos
Exceções	Não	Parcial	Sim	Sim
Concorrência	Não (biblioteca de funções)	Não (biblioteca de funções)	Sim	Sim

Tutorial

- Acesse: https://www.jetbrains.com/idea/download/#section=linux
- Escolha entre a versão Ultimate e a Community
- Embora a versão Ultimate seja paga, os alunos da UFES podem obter uma licença para alunos através do email: <seu-login-portal-aluno>@ufes.com.br
- Siga as instruções para instalar a IDE

Download IntelliJ IDEA

Windows Mac Linux

Version: 2019.2.4
Build: 192.7142.36
10/28/2019
Release notes

System requirements
Installation Instructions
Other versions

Ultimate

For web and enterprise development

Download .tar.gz

Free trial

Community

For JVM and Android development

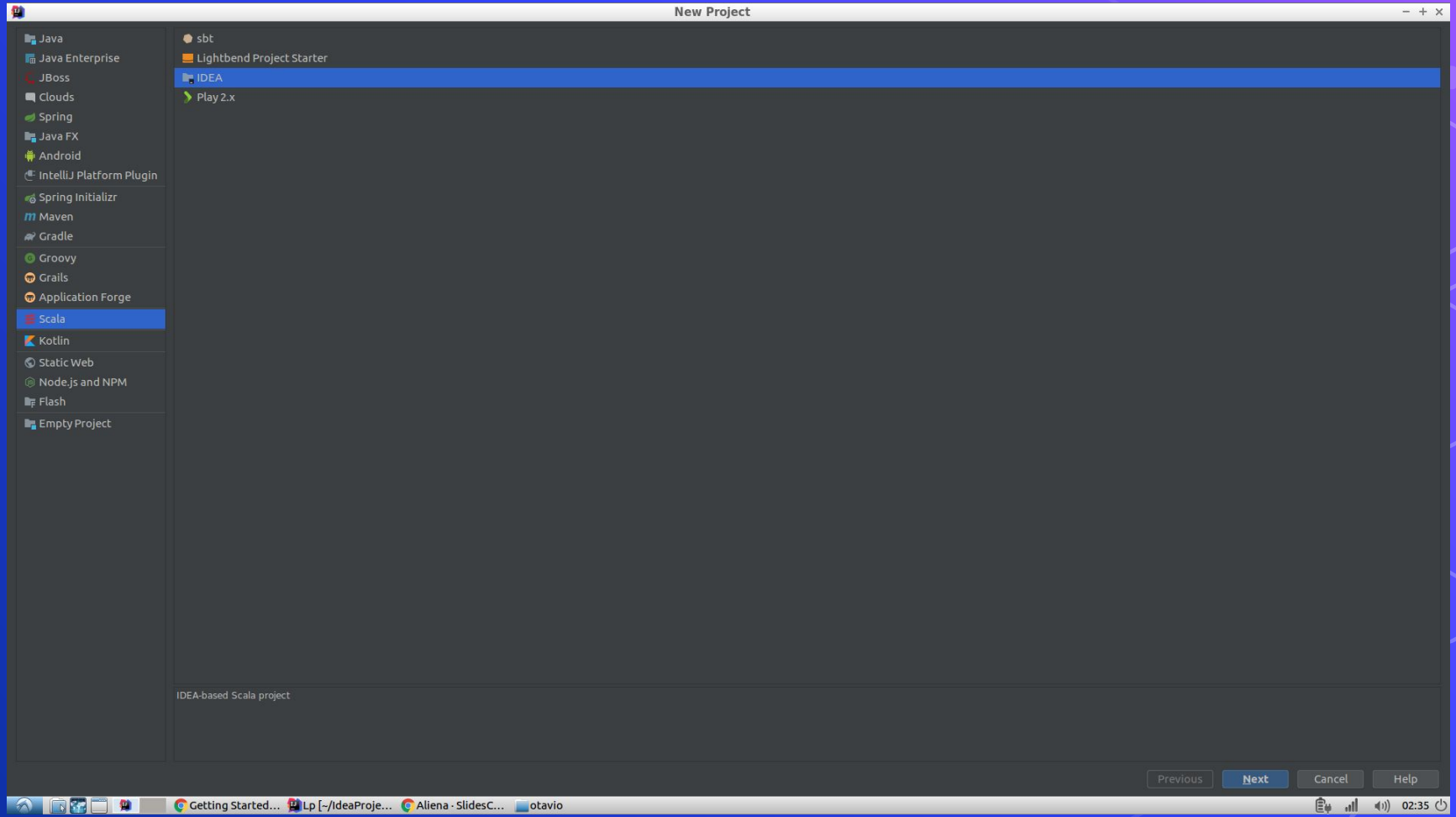
Download .tar.gz

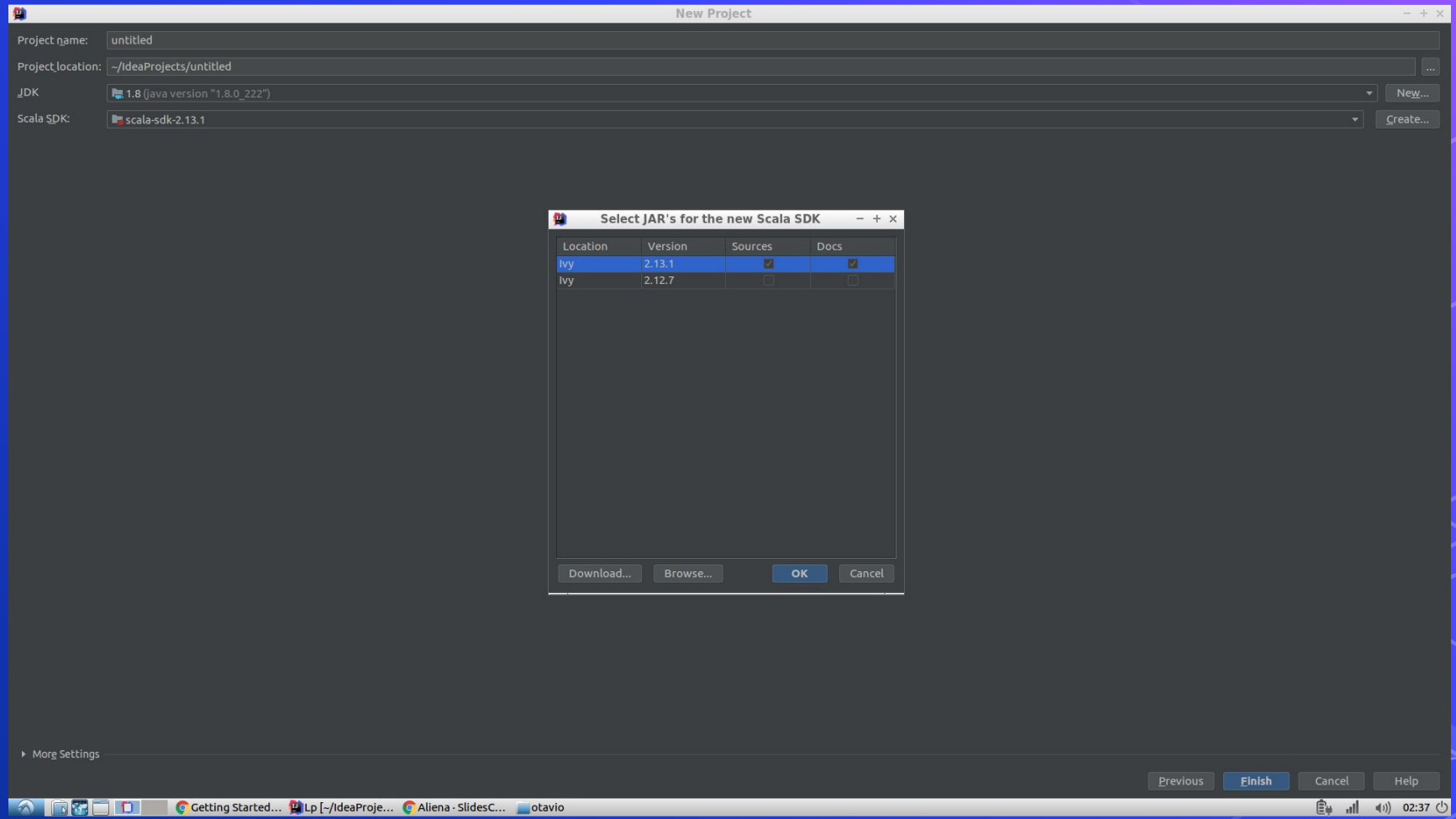
Free, open-source

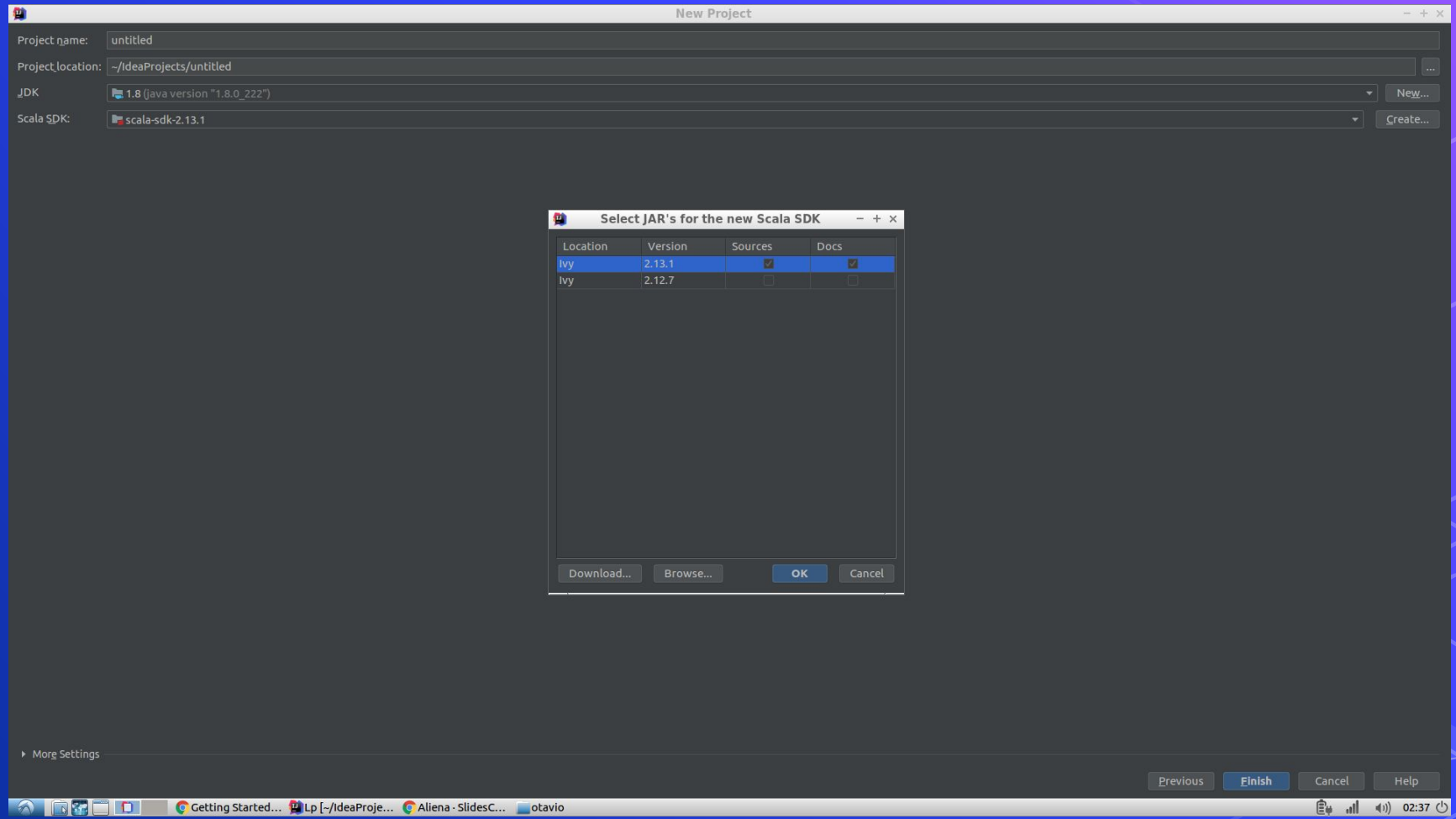
License	Commercial	Open-source, Apache 2.0
Java, Kotlin, Groovy, Scala	✓	✓
Android	✓	✓
Maven, Gradle, SBT	✓	✓
Git, SVN, Mercurial	✓	✓
Perforce	✓	×
JavaScript,TypeScript	✓	×
Java EE, Spring, Play, Grails, OtherFrameworks	✓	×

Download Intel... Lp [~/IdeaProje... Aliena · SlidesC... otavio 02:28

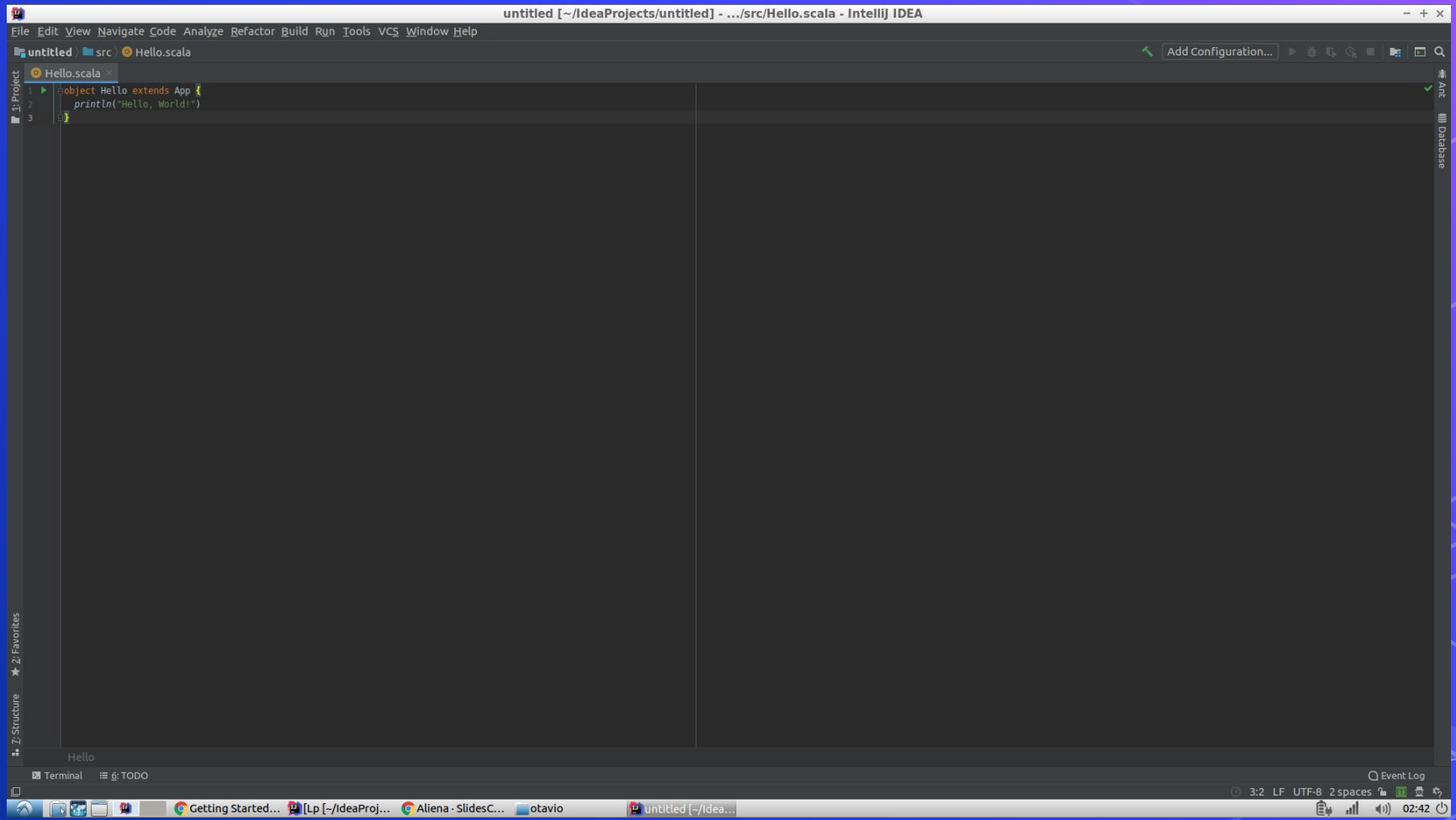
- Crie um novo projeto
- No painel esquerdo clique em Scala
- No painel direito clique em IDEA
- Dê um nome ao projeto
- Assumindo que seja a primeira vez que você cria um projeto usando Scala e o IntelliJ você precisará instalar o Scala SDK. Clique em create e baixe a versão mais recente


New Project
Java
Java Enterprise
JBoss
Clouds
Spring
Java FX
Android
IntelliJ Platform Plugin
Spring Initializr
Maven
Gradle
Groovy
Grails
Application Forge
Scala
Kotlin
Static Web
Node.js and NPM
Flash
Empty Project
sbt
Lightbend Project Starter
IDEA
Play 2.x
IDEA-based Scala project
Previous
Next
Cancel
Help
Getting Started...
Lp [~/IdeaProje...
Aliena · SlidesC...
otavio
02:35


New Project
Project name:
untitled
Project location:
~/IdeaProjects/untitled
JDK
1.8 (java version "1.8.0_222")
New...
Scala SDK:
scala-sdk-2.13.1
Create...
Select JAR's for the new Scala SDK
Location
Version
Sources
Docs
Ivy
2.13.1
Ivy
2.12.7
Download...
Browse...
OK
Cancel
More Settings
Previous
Finish
Cancel
Help
Getting Started...
Lp [~/IdeaProje...
Aliena · SlidesC...
otavio
02:37


New Project
Project name:
untitled
Project location:
~/IdeaProjects/untitled
JDK
1.8 (java version "1.8.0_222")
New...
Scala SDK:
scala-sdk-2.13.1
Create...
Select JAR's for the new Scala SDK
Location
Version
Sources
Docs
Ivy
2.13.1
Ivy
2.12.7
Download...
Browse...
OK
Cancel
More Settings
Previous
Finish
Cancel
Help
Getting Started...
Lp [~/IdeaProje...
Aliena · SlidesC...
otavio
02:37

- Clique com o botão direito em src e escolha New => Scala Class
- Nomeie sua classe e troque o tipo para Object
- Digite o código abaixo e clique na seta à direita


untitled [~/IdeaProjects/untitled] - .../src/Hello.scala - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
untitled src Hello.scala
Add Configuration...
Hello.scala
object Hello extends App {
println("Hello, World!")
}
1: Project
Ant
Database
2: Favorites
7: Structure
Hello
Terminal
6: TODO
Event Log
3:2 LF UTF-8 2 spaces
Getting Started...
[Lp [~/IdeaProj...
Aliena · SlidesC...
otavio
untitled [~/idea...
02:42

Obrigado!

Perguntas?