Universidade Federal de Itajubá

Itajubá, Junho de 2018

Trabalho Linguagens de Programação

Breno Salgado de Oliveira, Tales Henrique Carvalho, Érika Mayumi Saito Tagima, Leonardo Said da Costa, Andrei Alves Cardoso, Caíque Cléber Dias de Rezende

[1]Instituto de Engenharia de Sistemas e Tecnologia da Informação – Universidade Federal de Itajubá (UNIFEI)

Resumo. *Este documento foi desenvolvido como trabalho avaliativo da disciplina de Linguagem de Programação - ECOM04. Neste guia será abordado a instalação da linguagem, configuração, características e exemplos de aplicações reais. Na apresentação na sala de aula será demonstrado um programa simples implementado na linguagem para resolver um problema cotidiano.*

1. História

Scala, abreviação de *Scalable Language* começou a ser desenvolvida em 2001 na École Polytechnique Fédérale de Lausanne (EPFL), na Suíça por Martin Odersky. Scala teve sua aparição publicamente em 2004 ao ser publicada na plataforma Java, tendo grande motivação em seu potencial funcional e respaldo em Java, uma linguagem amplamente utilizada, madura e robusta. Scala foi desenvolvida como uma linguagem de programação funcional híbrida, integrando de forma uniforme os recursos de orientado a objetos e funcionais, esta linguagem foi compilada para rodar na JVM (Java Virtual Machine), além disso ela é utilizada por empresas que desejam aumentar a eficiência de desenvolvimento, lamento do software ou simplesmente deixar o código mais otimizado.

2. Instalação

As ferramentas de compilação e execução do Scala podem ser instaladas em qualquer sistema UNIX ou Windows. Como requisitos para execução de programas Scala estão o ambiente de execução Java (Java Runtime Environment) e os binários Scala. Para compilação de código em binários Java, é necessária a instalação do Java 8 JDK ou superior. Depois, basta uma IDE apropriada, sendo IntelliJ a recomendada para iniciantes. Alternativamente, é possível instalar o sbt (Scala's build tool), uma ferramenta de programação inteiramente baseada em comandos pelo terminal.

2.1. Requisitos

- Java Runtime Environment (JRE): necessário para rodar os binários gerados pelo Scala e uma variade de ferramentas para essa linguagem. Consulte a lista de links úteis (1) para o caminho de download. A instalação é simples e consiste em concordar com os termos e uso e prosseguir até a conclusão.
- Java Development Kit (JDK): utilitários para desenvolvimento em Java. Assim como java, a instalação é simples e intuitiva. Consulte o link (2) na seção de links úteis para download. Certifique-se de selecionar a opção adequada para seu sistema operacional. Recomenda-se alterar o diretório de instalação do JDK para um caminho fácil de ser encontrado e sem necessidade de permissões adicionais, já que isso facilita a configuração futura da IDE.

- Binários Scala: binários *scala* e *scalac*, para permitir a interpretação de códigos Scala, bem como a execução de binários *standalone* que dependam da saída padrão (terminal ou console). Consulte o link (3) na seção de links úteis para download. Baixe a versão mais recente. A instalação também consiste na simples sequência de passos do instalador.

2.2. IDE

A IDE recomendada para se programar em Scala é a IntelliJ IDEA, que permite programar em Scala de forma confortável, fácil e rápida. Com os requisitos instalados, basta baixar o IntelliJ Community Edition, seguindo o link (4) na seção de links úteis. A instalação consiste em seguir os passos do instalador, marcando as opções de se instalar o Launcher.

A IDE por padrão não vem com ferramentas para se desenvolver em Scala, sendo necessária a instalação de plugins. Ao abrir pela primeira primeira vez a IDE, ela dará a opção de customizar interface e configurar plugins. Pularemos essa etapa e faremos a configuração pela interface mais comum.

Para instalar o plugin Scala siga para a tela principal da IDE. No canto inferior direito, acesse o menu *Configure* e então a opção *Plugins*. Na janela de plugins, clique na opção *Browse repositories...*. Na janela de navegação pelos repositórios, role ou pesquise pela opção *Scala*, do tipo *LANGUAGES*. Selecione a opção para instalar e aguarde. Quando estiver concluída a instalação, reinicie a IDE.

Se o procedimento tiver sido executado com sucesso, ao selecionar a opção *Create New Project* na tela principal da IDE, deve aparecer a opção Scala à esquerda.

Prosseguimos para a criação de um novo projeto: seleciona-se a opção Scala à esquerda da tela *New Project* e então a opção *sbt*, à direita, e clicamos em *next*. Em seguida é pedido o nome do projeto e sua localização (nesse caso, pela conveniência do tutorial, colocaremos "HelloWorld"no nome e não mexeremos no diretório). Também escolha o caminho para a pasta do JDK pelo navegador da IDE (lembre-se de como é útil escolher um caminho fácil no momento da instalação!) e então selecione a versão do JDK da lista que sugir. Escolha as últimas versões do *sbt* e do *Scala* também nessa tela e dê *Finish*. Aguarde.

Se o procedimento tiver sido executado adequadamente, a IDE deve montar a árvore de diretórios e arquivos necessária para compilação. Ela é composta pelo menos pelo caminho <*source/main/scala/*> e pelo arquivo *build.sbt*. Siga o caminho <*source/main/scala/*> pela IDE e peça para criar um novo *Scala Script* com o nome *main* (caso essa opção não apareça, peça para simplesmente criar um novo arquivo com o nome *main.scala*. Ao abrir esse arquivo a IDE vai sugerir, na parte superior do editor de código, a instalação do Scala SDK. Aceite, clique em *Download* e aguarde). Mais adiante, compilaremos o primeiro Hello World.

2.3. Linha de comando

O uso da ferramenta de linha de comando, o *Scala's Build Tool* (*sbt*), segue os mesmos requisitos da IDE. Siga o link (5) na seção de links úteis para download do *sbt*. A instalação em sistemas Windows é simplificada e não exige configurações adicionais. Se a instalação for bem sucedida, ao abrir o console (ou terminar) e executar o comando:

`sbt version`.

o sistema deve responder com a versão do *sbt*.

Crie uma pasta vazia e navegue até ela usando o console. Rode o comando *sbt* sozinho. No próprio console do sbt, rode o comando:

`new scala/hello-world.g8`.

Isso serve para baixo um Hello-World padrão de um repositório online. Será criada outra pasta com o nome solicitado logo depois da execução do comando. Quando concluído, navegue para a nova pasta, e novamente rode o comando *sbt*. No sbt, rode o comando run. Se tudo tiver sido feito adequadamente, a frase "Hello, World!"deve ser impressa no console logo antes do programa terminar. Para editar o código, acesse o caminho <*source/main/scala/main.scala*>.

2.4. Interpretador

Uma outra forma de se rodar códigos Scala é interpretando o código fonte diretamente. Para isso, basta navegar até o diretório onde se encontra o código usando o console e rodar o comando:

`scala NomeDoArquivo.scala`.

Observe que se o código tiver dependências de outras bibliotecas, esse método pode ser insuficiente.

2.5. Compilando um Fat JAR

Por vezes, é conveniente distribuir os programas como um único arquivo que executa em qualquer sistema. Isso pode ser feito compilando um pacote Fat Jar.

Usando o sbt, isso pode ser feito da seguinte forma:

- No diretório do projeto, navegue para a pasta *project* e crie um arquivo com o nome <*assembly.sbt*>. Abra o arquivo com algum editor de texto puro, e escreva a seguinte linha:
  `addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.7")`.
- Salve o arquivo e execute o sbt na pasta do projeto.
- Para compilar o Fat Jar, execute o seguinte comando no sbt:
  `assembly`.
- Se tudo tiver sido feito corretamente, um grande arquivo .jar deve surgir em no diretório <*target/scala-x.yz*> (sendo x, y e z no nome dependente da versão do Scala compilada), relativa à raiz do projeto. Esse arquivo está pronto para ser executado em qualquer sistema Java.

2.6. Um Hello, World!

Segue abaixo um simples código de *Hello, World!*, ou Ola, Mundo!, em Scala:

```
object OlaMundo{
  def main(args:Array[String]): Unit={
    println("Ola, mundo!");
  }
}
```

Para compilar, siga as instruções da ferramenta que estiver usando (*sbt* ou *interpretador*). Na IDE, basta seguir a opção *Run* no menu superior, e a saída do programa será no console da própria IDE.

3. Características

Scala é uma linguagem de múltiplos paradigmas, orientada objeto, assim como também é uma linguagem funcional com suporte a identificação de padrões sendo estendido até XML, dando ampla aplicação de Scala para Web Services. Ela também é considerada fortemente e estaticamente tipada, de modo que a linguagem garante que abstrações sejam usadas de forma segura e coerente.Deste modo Scala tem suporte aos seguintes modelos:

- Classes Genéricas
- Anotações de variância
- Classes internas
- Tipos Compostos
- Referências próprias explicitamente tipadas
- Parâmetros implícitos
- Métodos polimórficos
- Suporte à todas as classes da Java SDK e classes Java customizadas

3.1. Sintaxe básica

A maior diferença de sintaxe entre Java e Scala é que o ";"no final da linha de código é opcional, o término da linha também pode ser representada por uma nova linha, no geral o uso do ponto e vírgula fica reservado para quando há vários comandos na mesma linha, neste caso o uso é obrigatório. Todos os componentes precisam de nomes, os nomes de: objetos, classes, variáveis e métodos são chamados **identificadores**. Uma palavra reservada não pode ser usada como identificador. Outras propriedades da sintaxe da linguagem Scala são:

- *Case-sensitive* : Os identificadores Hello e hello podem ter significados diferentes em Scala;
- Nomenclatura padrão de classes: a primeira letra deve ser maiúscula assim como outras palavras inclusas. Ex.: *class MyFirstScalaClass*;
- Nome de métodos devem começar com letra minúscula, em casos de mais de uma palavra devem iniciar com letra maiúscula. Ex.: *def myMethodName()*;
- Arquivo do programa deve ter o nome exato do nome do objeto incluindo a extensão ".scala";
- **def main(args: Array[String])** todo programa Scala começa pelo método *main()*.

3.2. 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	
-	:	=	=>
<-	<:	<%	>:
#	@		

4. Componentes Base

4.1. Pacotes Scala

Um pacote é um módulo de código que agrupa classes, interfaces e exceções relacionadas entre si. Ela possui um nome e deve ser inserido na primeira linha do arquivo. A declaração de um pacote possui a seguinte sintaxe:

```
package com.liftcode.stuff
```

Uma das grandes vantagens do Scala é sua flexibilidade ao importar pacotes. Um pacote externo pode ser importado para o programa, com no exemplo:

```
import scala.xml._
```

Onde o símbolo underline (_) indica que todas as classes do pacote são importadas. Caso não seja necessário importar todos os recursos do pacote, é possível importar seletivamente os membros de um determinado pacote:

```
import scala.collection.{Vector, Sequence}
```

O Scala ainda oferece ainda outros recursos, como importar todo um pacote exceto um determinado membro e renomear uma importação.

4.2. Expressões

Valores

Valor é um tipo de dado em Scala que tem seu valor imutável, caracterizado pela palavra **var**:

```
val x = 1 + 1
println(x) // 2
```

Caso o programador alterar seu valor durante o programa, haverá um erro de compilação. Os valores são implicitamente tipados, ou seja, o compilador infere os tipos no momento de uso delas. Porém, seu tipo pode ser explícito, como demonstrado abaixo:

```
val x: Int = 1 + 1
```

Variáveis

Variável é um tipo de dado em Scala que possui valor mutável, caracterizado pela palavra **val**. Assim como no Valor, um objeto Variável pode ser declarado definindo seu valor inicial, e o compilador infere seu tipo ao ser utilizado.

```
var x = 1 + 1
x = 3 // "x"  declarado inicialmente com a palavra-chave "var"
println(x * x) // 9
```

Para uma Variável com o tipo explícito, sua declaração é demonstrada abaixo:

```
var x: Int = 1 + 1
```

4.3. Blocos

Em Scala, assim como em outras linguagens, é possível combinar diferentes códigos agrupados em apenas um bloco usando chaves, como demonstrado abaixo:

```
println({
  val x = 1 + 1
  x + 1
}) // 3
```

4.4. Estrutura Condicional If-else

A estrutura **if-else** do Scala é igual a do Java. O termo **if** pode ser usado isoladamente, ou seguido de vários termos **else-if** em cascata, ou seguido de apenas um **else**. É possível também cascatear **if**s dentro de ifs. O exemplo demonstra seu uso:

```
object Exemplo {
  def main(args: Array[String]) {
    var x = 30;

    if( x <= 10 ){
      if(x == 10){
        println("Valor de X eh 10");
      }
      if(x == 5){
        println("Valor de X eh 5");
      }
    } else if( x == 20 ){
      println("Valor de X eh 20");
    } else if( x == 30 ){
      println("Valor de X eh 30");
    } else{
      println("Resultado para o else.");
    }
  }
}
```

O Scala permite a mesma sintaxe para escolha entre duas expressões dentro de atribuições, funcionando como substituto para o operador ternário (... ? ... : ...).

```
def abs(x: Double) = if (x >= 0) x else -x
```

4.5. Estruturas de repetição

4.5.1. For

A estrutura de um laço **for** no Scala possui uma estrutura diferente do Java, sendo um *range* o critério de contagem de *loops*. O operador seta (<-) é chamado de **gerador**, pois gera valores individuais de um *range*. Existem duas formas de criar um laço **for** simples, usando as tags **to** e **until**. A diferença entre os dois é demonstrada abaixo:

```
for( var x <- 1 to 10 ){
  // O loop repete 10 vezes
}
```

```
for( var x <- 1 until 10 ){
  // O loop repete 9 vezes
}
```

O Scala também permite ao programador utilizar um objeto List ao invés de um range, sendo este uma coleção pelo qual o laço fará a iteração de acordo com o número de elementos nele contido.

É possível utilizar múltiplos *ranges* dentro de um *loop*. Separando os valores por ponto e vírgula (;), o laço fará a iteração entre todas as possibilidades dentro do *range*:

```
var a = 0;
var b = 0;

for( a <- 1 to 2; b <- 1 to 3){
  print("(" + a + ", " + b + ") ");
  // Saida: (1, 1) (1, 2) (1, 3) (2, 1) (2, 2) (2, 3)
}
```

O Scala ainda oferece recursos para laços **for**, como filtros condicionais e o **Yield**, que armazena os valores retornados do *loop* **for** em uma variável.

4.5.2. While e Do-While

As estruturas dos laços **while** e **do-while** são as mesmas do Java e não possuem grandes diferenças quanto ao seu uso. Abaixo, são apresentados dois exemplos de como essas estruturas são utilizadas.

```
while(condicao){
  // codigo
}
```

```
do {
  //codigo
}
```

```
while( condicao );
```

4.6. Funções

Em Scala, as funções são objetos e por este motivo é possível utilizá-las de diversas formas, como passar funções como argumentos de outras funções. Essas funções recebem o nome *função de ordem superior*. Sua estrutura básica é:

```
def nomeFuncao ([lista de parametros]) : [return tipo] = {
  // corpo da funcao
  return [expr]
}
```

Em Scala, a palavra-chave **return** pode ser omitida em algumas funções. Neste caso, a função retorna um valor do tipo **Unit**, similar ao *void* do Java. Por padrão, se nenhuma expressão de retorno é fornecida, o compilador assume que o tipo retornado é **Unit**, o que pode causar problemas em alguns casos.

Função anônima

Uma função anônima é definida sem declaração.

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

Ela pode ser atribuída à uma variável:

```
val adicionar = (x: Int, y: Int) => x + y
println(adicionar(1, 2)) // 3
```

Função como parâmetro

Como dito acima, as funções de ordem superior permitem a passagem de funções como argumento. Isso dá uma maior flexibilidade à linguagem, pois é possível modificar todo funcionamento de uma função com apenas uma linha de código. Esse processo pode ser feito diretamente, chamando uma função pelo seu nome, ou através das funções anônimas.

```
def soma(a:Int, b:Int, f:Int => Int):Int = a + b + f(b)
def sqr(x:Int) = x * x

println(soma(1,3,(x:Int) => x*x)) // Resultado: 1+3+(3*3)=13
println(soma(1,3,(x:Int) => x+x)) // Resultado: 1+3+(3+3)=10
println(soma(1,3,sqr))            // Resultado: 1+3+(3*3)=13
```

Currying

Currying é um tipo específico de função que retorna uma outra função. Para uma aplicação de soma entre duas variáveis, por exemplo, pode ser necessário que um dos parâmetros deva ser fixo, enquanto o outro varia em seu valor, gerando resultados diferentes. Utilizando a técnica de Currying:

```
def add(x:Int)(y:Int)=x+y
val op1 = add(1)(5) //1+5
val op2 = add(4)_ //Retorna funo Int=>Int
val op3 = op2(2) //4+2
```

O Currying possui principal aplicação em problemas matemáticos complexos.

4.7. Classes

Classes podem ser definidas pela palavra class seguida por seu nome e parâmetros construtores

```
class Greeter(prefix: String, suffix: String) {
  def greet(name: String): Unit =
    println(prefix + name + suffix)
}
```

O tipo de retorno ***greet*** é ***Unit***, o qual significa que não haverá retorno da função assim como ***void*** em Java e C.

Obs.: A diferença é que já que toda expressão em Scala precisa de um valor, existe um tipo Unit, porém não contém nenhuma informação

4.8. *Case Classes*

Scala tem um tipo especial de classe chamada de *"case"class*, na qual são imutáveis e comparáveis à um valor.

```
case class Ponto(x: Int, y: Int)
```

As ***case classes*** podem ser instânciadas sem a palavra chave ***new***

```
val ponto = Ponto(1, 2)
val outroPonto = Ponto(1, 2)
val outroOutroPonto = Ponto(2, 2)
```

Elas são comparadas por valor

```
if (ponto == outroPonto) {
  println(ponto + " e " + outroPonto + " sao o mesmo.")
} else {
  println(ponto + " e " + outroPonto + " sao diferentes.")
} // Ponto(1,2) e Ponto(1,2) sao o mesmo.

if (ponto == outroOutroPonto) {
  println(ponto + " e " + outroOutroPonto + " sao o mesmo.")
} else {
  println(ponto + " e " + outroOutroPonto + " sao diferentes.")
} // Ponto(1,2) e Ponto(2,2) sao diferentes.
```

4.9. Objetos

Objetos são instâncias únicas com suas definições próprias

```
object IdFabrica {
  private var contador = 0
  def create(): Int = {
    contador += 1
    contador
  }
}
```

Você pode acessar o objeto referenciando seu nome

```
val novoId: Int = IdFabrica.create()
println(novoId) // 1
val novissimoId: Int = IdFabrica.create()
println(novissimoId) // 2
```

4.10. Vetores

Vetor unidimensional

```
object ExemploVetor {
  def main(args: Array[String]) {
    var vetor1 = Array(1.9, 2.9, 3.4, 3.5)

    // Mostrando todos os elementos:
    for ( x <- vetor1 ) {
      println( x )
    }

    // Somando todos os elementos:
    var total = 0.0;

    for ( i <- 0 to (vetor1.length - 1)) {
      total += vetor1(i);
    }
    println("Total eh " + total);

    // Encontrando o maior elemento:
    var max = vetor1(0);

    for ( i <- 1 to (vetor1.length - 1) ) {
      if (vetor1(i) > max) max = vetor1(i);
    }

    println("Max eh " + max);
  }
}
```

Vetor bidimensional

```
import Array._

object ExemploMatriz {
  def main(args: Array[String]) {
    var matriz1 = ofDim[Int](3,3)

    // Constroi a matriz
    for (i <- 0 to 2) {
      for ( j <- 0 to 2) {
        matriz1(i)(j) = j;
      }
    }

    // Mostra os valores da matriz
    for (i <- 0 to 2) {
      for ( j <- 0 to 2) {
        print(" " + matriz1(i)(j));
      }
      println();
    }
  }
}
```

No exemplo acima, a função ofDim cria uma matriz com as dimensões dadas. Notação: *def ofDim[T](n1: Int): Array[T].*

4.11. Traços(*Traits*)

Traços são tipos contendo certos campos e métodos, se comportando como Interfaces como no Java

```
trait Recepcionista {
  def saudar(name: String): Unit
}
```

Traços também podem ter implementações padrões

```
trait Recepcionista {
  def saudar(nome: String): Unit =
    println("Ola, " + nome + "!")
}
```

Traços podem ser estendidos com ***extends*** e podem ser sobrescritos com a palavra ***override***

```
class RecepcionistaPadrao extends Recepcionista

class RecepcionistaPersonalizado(prefixo: String, posfixo:
   String) extends Recepcionista {
```

```
  override def saudar(nome: String): Unit = {
    println(prefixo + name + posfixo)
  }
}

val recepcionista = new RecepcionistaPadrao()
recepcionista.saudar("Scala developer") // Hello, Scala
    developer!

val recepcionistaPersonalizado = new
    RecepcionistaPersonalizado("Como esta, ", "?")
recepcionistaPersonalizado.saudar("Scala developer") // How are
    you, Scala developer?
```

4.12. Método *Main*

Esse método é o ponto de entrada do programa, em que a JVM exige um método chamado de ***main*** que recebe um argumento e um vetor de ***Strings***

```
object Main {
  def main(args: Array[String]): Unit =
    println("Hello, Scala developer!")
}
```

5. Estruturas da biblioteca padrão

5.1. Lista

É uma estrutura fundamental de dados e pode ser definida por *List(x1, . . . , xn)*:

```
val fruta = List("macas", "laranjas", "peras")
val nums = List(1, 2, 3, 4)
val dialog3 = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val vazio = List()
```

Listas são imutáveis, recursivas e homogenias. O tipo da lista pode ser explicitamente definido também:

```
val fruta: List[String] = List("macas", "laranjas", "peras")
val nums: List[Int] = List(1, 2, 3, 4)
val dialog3: List[List[Int]] = List(List(1, 0, 0), List(0, 1,
    0), List(0, 0, 1))
val vazio: List[Nothing] = List()
```

Junto das listas tem-se operações como *map* e *filter*:

```
List(1, 2, 3).map(x => x + 1) == List(2, 3, 4)
List(1, 2, 3).filter(x => x % 2 == 0) == List(2)
```

5.2. Valores Opcionais

É possível representar um valor opcional do tipo ***A*** com o tipo ***Option[A]***, esse recurso é util para funções parcialmente definidas:

```
def sqrt(x: Double): Option[Double] =
  if (x < 0) None else Some(...)
```

Com valores opcionais é possível manipular seus valores de saída:

```
def foo(x: Double): String =
  sqrt(x) match {
    case None => "sem resultado"
    case Some(y) => y.toString
  }
```

5.3. Tratamento de Erros

5.3.1. Try

O Bloco **Try[A]** representa uma tentativa de executar ***A***, que pode ser um sucesso ou uma falha:

```
def sqrt(x: Double): Try[Double] =
  if (x < 0) Failure(new IllegalArgumentException("x deve ser
     positivo."))
  else Success(...)
```

5.3.2. Either

Either é utilizado também para tratamento de erros. Ele tem como função fazer uma escolha em duas opções(decomposto em **Left** e **Right**):

```
def sqrt(x: Double): Either[String, Double] =
  if (x < 0) Left("x deve ser positivo")
  else Right(...)
```

6. Usos de Scala

Atualmente Scala é usado largamente em aplicações de desempenho em Big Data sendo linguagem base para o desenvolvimento da plataforma Spark. Além de suas aplicações para alto desempenho em big data, Scala também vem sendo utilizada em sistemas de processamento distribuído. Essa influência em mercado vem sendo demonstrada a partir da aceitação da linguagem em empresas grandes como LinkedIn, Twitter, Sony, Foursquare, Netflix, AirBnB, Apple e outras aplicações por Frameworks.

7. Exemplos de código

7.1. Fórmula de Bhaskara

```
object Bhaskara{
  def main(args: Array[String]): Unit={
    println("Calculadora de raizes de equacao ax^2+bx+c = 0")
    println("Insira os parametros:")
    print("a = ")
    val a = readDouble()
    if(a==0.0){
      println("Nao consigo resolver com a = 0.0!")
      return
    }
    print("b = ")
    val b = readDouble()
    print("c = ")
    val c = readDouble()

    var delta:Double = 0.0
    var x1:Double = 0.0
    var x2:Double = 0.0

    delta = b*b - 4.0 * a * c

    if(delta<0){
      println("Essa equacao tem raizes complexas!")
    }else{
      x1 = (-b + Math.sqrt(delta))/(2*a)
      x2 = (-b - Math.sqrt(delta))/(2*a)

      println(s"x1 = $x1")
      println(s"x2 = $x2")
    }
  }
}
```

7.2. Herança de classes

```
object HerancaCaseClass {
  abstract class Pessoa {
    def nome: String
    def idade: Int
    // endereco e outras propriedades
    // metodos
    def getNome(): String = {
      return nome
    }
  }
```

```
  case class Gerente(val nome: String, val idade: Int, val taxa:
     Int)
     extends Person

  case class Empregado(val nome: String, val idade: Int, val
     salario: Int)
     extends Person

  case class PessoaA(val nome: String, val idade: Int, val
     salario: Int) extends Pessoa

  def main(args: Array[String]) {
       val p = PessoaA("Fulano",10,42)
       println(p.getNome)
  }
}
```

8. Links Úteis

- (1) Java Runtime Environment (JRE) - Download - <www.java.com/download>
- (2) Java Development Kit (JDK) - Download - <oracle.com/technetwork/java/javase/downloads/jdk10-downloads-4416644.html>
- (3) Scala - Download - <www.scala-lang.org/download/all.html>
- (4) IntelliJ IDEA - Download - <www.jetbrains.com/idea/download>
- (5) Scala's Build Tool - Download - <www.scala-sbt.org/download.html?_ga=2.188632609.888614847.1537414292-1432949400.1536796134>

9. Referências

WAMPLER, Dean; PAYNE, Alex. **Programming Scala**. 2a. ed. [S.l.]: O'Reilly Media, 2014. 583 p.

ODERSKY, Martin; SPOON, Lex; VENNERS, Bill. **Programming in Scala**. 3a. ed. [S.l.]: Artima, 2016. 859 p.

SCALA Exercises. Disponível em: <https://www.scala-exercises.org/>. Acesso em: 17 set. 2018.

Who's using Scala?. Disponível em:<https://alvinalexander.com/scala/whos-using-scala-akka-play-framework>

Programming in Scala, Third Edition. Disponível em: <https://booksites.artima.com/programming_in_scala_3ed/examples/html/ch02.html>

Scalacheat Scala Documentation. Disponível em: <https://docs.scala-lang.org/cheatsheets/index.html>

Basics Scala Documentation. Disponível: <https://docs.scala-lang.org/tour/basics.html>

Scala(programming language)-Wikipedia. Disponível em: <https://en.wikipedia.org/wiki/Scala_(programming_language)>