Scala Fundamentals

Uma linguagem de programação escalável

Michael Silva

Classes e Objetos

- A sintaxe para criação de classes

```
class Team {
  // membros
}
```

- Instanciação de objeto referente a uma determinada classe

```
new Team
```

Membros de Classes e Objetos

- Os modificadores de acesso são: **private** e **protected**, para acesso público, não é necessário incluir o modificador de acesso
- Os parâmetros de métodos são constantes (val), não é possível alterá-los
- Não é recomendado que um método tenha múltiplos retornos, isso encoraja a criação de métodos pequenos e com responsabilidade definida
- Não é necessário incluir o tipo de retorno de uma função, por causa da inferência de tipo do Scala

Exemplo de classe com membros

```
// In file Team.scala
class Team {
  private var trophy = 0
  def add(b: Int): Unit = { trophy += b }
  def checksum(): Int = ~(trophy & 0xFF) + 1
}
```

Objetos *Singleton*

- Uma classe não tem membros estáticos (por isso é "mais orientada a objetos"). Scala tem **objetos singleton**
- Os "membros estáticos" de uma classe devem ser criados no seu **companion object**
- Um *companion object* deve ter o mesmo nome e ser implementado no mesmo arquivo de sua classe
- Uma *companion class* e um *companion object* podem acessar os membros privados um do outro

Exemplo de *Companinon Object*

```scala
class Team(val name: String, val city: String)

object Team {
  def fromString(teamString: String): Option[Team] = {
    teamString.split(':') match {
      case Array(a, b) => Some(new Team(a, b))
      case _ => None
    }
  }
}

// Usar
val team = Team.fromString("Flamengo.RJ")
```

Tipos de dados

- **Byte** 8-bit signed two's complement integer (-27 to 27 - 1, inclusive)
- **Short** 16-bit signed two's complement integer (-215 to 215 - 1, inclusive)
- **Int** 32-bit signed two's complement integer (-231 to 231 - 1, inclusive)
- **Long** 64-bit signed two's complement integer (-263 to 263 - 1, inclusive)

Tipos de dados

- **Char** 16-bit unsigned Unicode character (0 to 216 - 1, inclusive)
- **String** a sequence of Chars
- **Float** 32-bit IEEE 754 single-precision float
- **Double** 64-bit IEEE 754 double-precision float
- **Boolean** true or false

Classe Imutável

```
class Rational(n: Int, d: Int) {
  require(d != 0)

  private val g = gcd(n.abs, d.abs)

  val numer = n / g

  //overload do construtor
  def this(n: Int) = this(n, 1)

  override def toString = "The numer is: " + numer

  private def gcd(a: Int, b: Int): Int =
  if (b == 0) a else gcd(b, a % b)
}
```

Classe Imutável

Pensando Juntos

Por que classe/objeto imutável é importante?

Estruturas de Controle

Comando IF

```
val filename =
  if (!args.isEmpty) args(0)
  else "default.txt"
```

Estruturas de Controle

Comando WHILE

```
// While
var a = 5
var b = 10
while (a != 0) {
  val temp = a
  a = b % a
  b = temp
}

// Do While
var line = ""
do {
  line = readLine()
  println("Read: " + line)
} while (line != "")
```

Estruturas de Controle

Comando MATCH

```
val firstArg = if (args.length > 0) args(0) else ""
firstArg match {
  case "salt" => println("pepper")
  case "chips" => println("salsa")
  case "eggs" => println("bacon")
  case _ => println("huh?")
}
```

Função como valor

First-Class Function

```
val increase = (x: Int) => x + 1
increase(10)

// Passando a funcao para filtrar uma collection
someNumbers.filter((x: Int) => x > 0)

// Mais enxuto
someNumbers.filter(x => x > 0)

// Usando o placeholder
someNumbers.filter(_ > 0)

// Placeholder como parametro
someNumbers.foreach(println _)
```

Argumento com valor default

```
def printTime2(out: java.io.PrintStream = Console.out,
               divisor: Int = 1) =
    out.println("time = " + System.currentTimeMillis()/
        divisor)

// Especificando o valor a ser passado como argumento
printTime2(divisor = 1000)
```

Tail Recursive

- Função recursiva pode degradar a perfomance
- **Tail Recursive**: Quando uma função recursiva é criada a qual a última execução é a chamada para ela mesma
  - Nesse caso, o Scala faz uma otimização: Para evitar o gasto de memória/processamento enpilhando os contextos de execução, ocorre um *jump* para o inicio da execução com os valores atualizados

Tail Recursive

```
@tailrec
private def sumWithAccumulator(list: List[Int], accumulator:
    Int): Int = {
    list match {
        case Nil => accumulator
        case x :: xs => sumWithAccumulator(xs, accumulator +
            x)
    }
}
```

Composição e Herança

Composição

Quando uma classe tem, entre seus atributos, referência para outra classe

Herança

Relacionamento entre *subclass* ou *superclass*

Composição e Herança

Herança

```
class Element(height: Double) {
  def doubleHeigth: Double = height * height
  final def hiMethod: String = "Hi"
}

class SpecificElement(width: Double) extends Element {
  def elementSize: Double = doubleHeight + (width * width)
}

class NotSpecificElement(width: Double) extends Element {
  override def elementSize = 1.0
}
```

Traits

- Encapsula definição de métodos e atributos
- Uma classe pode extender múltiplas Traits
- Pode conter implementação de métodos

```
trait Philosophical {
  def philosophize() = {
    println("I consume memory, therefore I am!")
  }
}

class Frog extends Philosophical {
  override def toString = "green"
}
```

Traits

```
trait Philosophical {
  def philosophize() = {
    println("I consume memory, therefore I am!")
  }
}

class Animal

trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
  override def toString = "green"
}
```

Contato

Email

michael@michaelsilva.io

Bibliografia

[1] Odersky, Martin and Spoon, Lex and Venners, Bill. *Programming in Scala*. Artima Incorporation, 2016.