CONGRESSO CATARINENSE DE CIÊNCIA DA COMPUTAÇÃO
2022. Ano 9, Vol. 9. ISSN: 2319-0655

LINGUAGEM FUNCIONAL SCALA

Carlos Henrique Pinheiro Cordeiro[1], Renan Shindi Okada[2], Daniel Gomes Soares[3]

[1] Estudante do curso de Bacharelado em Ciências da Computação do Instituto Federal Catarinense, IFC, Rio do Sul – SC
[2] Estudante do curso de Bacharelado em Ciências da Computação do Instituto Federal Catarinense, IFC, Rio do Sul – SC
[3] Professor do curso de Bacharelado em Ciências da Computação do Instituto Federal Catarinense, IFC, Rio do Sul – SC

carloshenpincord@gmail.com, renanokada2000@gmailcom, daniel.soares@ifc.edu.br

Abstract. *The purpose of this article is to present the main points of the functional language when introducing the Scala language. Organized in six parts and ways to understand the proposal, being introduction, methodology, history, characteristics of the Scala programming language and the development of your ideas in the functional aspect along with its coding and conclusion. The research is of the bibliographic type and languages with high market share were identified, such as Java and C#*

Key-words:*Functional; scale; Java.*

Resumo. *A proposta deste artigo é apresentar os principais pontos da linguagem funcional quando introduzido a linguagem Scala. Organizada em seis partes e caminhos para o entendimento da proposta, sendo introdução, metodologia, histórico, características da linguagem de programação Scala e o desenvolvimento de suas ideias no quesito funcional junto a sua codificação e a conclusão. A pesquisa é do tipo bibliográfico e identificou-se linguagens de grade participação no mercado como Java e C#*
Palavras-chave: *Funcional; Scala; Java.*

1. Introdução

Ao introduzir o desenvolvimento de sistemas é de se habituar na utilização da linguagem imperativa e orientada a objeto, todavia, é de se perceber a utilização e desenvolvimento com um paradigma de programação cada vez mais crescente, a Linguagem Funcional. Com ele o desenvolvimento é feito com expressões, como se as funções fossem os objetos, dessa forma o valor de saída depende apenas dos argumentos de entrada.

Destaca-se, dessa forma, a linguagem de programação Scala, uma linguagem que mistura os paradigmas Funcional e Orientado a Objeto, de tipagem estática e que roda sobre a JVM e interopera com a linguagem java (BAZILIO, 2019). Assim, vale lembrar a importância e a influência dos princípios de Linguagens Funcionais, dando ênfase na linguagem Scala, é oque pretende o presente artigo. Para tanto, levantar teorias e conceitos de sua importância e como afeta a programação em si é fundamental para o entendimento de sua funcionalidade e eficiência quando utilizada de forma correta, ou seja, com um estudo prévio.

De mesma fora, o foco será no entendimento da linguagem Scala e como isso se relaciona e com a linguagem funcional , realizando inicialmente a explicação de seu conceito, e após esclarecer sua funcionalidade e sua função, será focado na implementação da linguagem Scala ,dando ênfase em sua utilização e sua funcionalidade em exemplos.

Por fim, serão desenvolvidas as explicações da codificação da linguagem Scala com foco no programação orientada a aspectos, entendo seu funcionamento e desenvolvimento que se assemelham a linguagem Java, junto ao entendimento do funcionamento de uma linguagem Funcional junto a exemplos e implementações para o melhor entendimento.

2. Metodologia

A revisão de materiais que tratam as principais fontes históricas dos princípios da Linguagem Funcional Scala junto ao entendimento da importância de sua função para o meio da computação foram os principais recursos metodológicos adotados neste artigo, o que poderia classificá-lo como uma pesquisa bibliográfica ou estudo de revisão.

Em outras palavras, os apontamentos realizados a partir de leituras formaram os capítulos seguintes. Boa parte inspirada nas fontes localizadas em exercícios de levantamento de materiais, principalmente pelo recurso de buscas na internet, onde houve o acesso em sites de pesquisa científicas e acadêmicas e artigos existentes.

3. Histórico

A linguagem de programação Scala (TORRES, 2019) teve seu desenvolvimento iniciado, no ano de 2001, tendo sua primeira versão lançada em 2003, e em 2006 ganhando sua segunda versão (Scala v2.0), ganhando popularidade desde então. Desenvolvida por Martin Odersky e seu grupo na *École Polytechnique Fédérale de Lausanne* (EPFL) em Lausanne na Suíça, surgiu da meta, estabelecida em 1999, de combinar a programação Orientada a Objetos com a Funcional.

E para reforçar a ideia da linguagem de programação Scala, completa Rodrigues (p. 21), explicando que:

O advento da linguagem de programação Scala é algo relativamente recente no mundo da programação, se tivermos em conta algumas das linguagens ainda em uso. Embora tenha começado a ser concebida em 2001, por Martin Odersky, e a primeira versão surgido em 2003, foi a partir da segunda versão desta linguagem que começou realmente a ganhar popularidade, no ano de 2006 (Scala, 2013). As principais características desta linguagem são a escalabilidade e o facto de ser multiparadigma – assenta no paradigma orientado a objetos e funcional.

Figura 1 - Martin Odersky, desenvolvedor da linguagem Scala. Disponível em: <http://www2.ic.uff.br/~bazilio/cursos/lp/material/Scala.pdf>.

Como primeiro plano adotado para atingir tal meta, tentou-se utilizar a linguagem Funnel, mas esta não se mostrou agradável, na prática, para atingir o objetivo. Então, em alternativa ao Funnel, mas trazendo algumas de suas ideias, foi desenvolvida a Scala. Vale constar que (LAGO, 2015) esta linguagem, e suas principais ferramentas, são *software* livre, disponíveis sob as licenças Apache e BSD.

4. Características

A Scala (NETO, 2014) é uma linguagem multiparadigma, sendo totalmente orientada a objetos também comportando aspectos da programação funcional, assim combinando estes dois paradigmas de programação. Também é multiplataforma, podendo ser executada nas plataformas: Android, JVM (*Java Virtual Machine*) e .NET. Vale constar também que esta é uma linguagem compilada, gerando *bytecodes*, possibilitando a utilização de bibliotecas Java sem a necessidade de cópias de código ou interfaces.

É uma linguagem estaticamente tipada, se mostrando ótima para criar *scripts* que comportam componentes Java. E, demonstra sua maior força no campo do desenvolvimento de *frameworks* e grandes sistemas, onde componentes reutilizáveis são necessários.

Dessa forma a linguagem Scala se mostra bastante útil em sua utilização, principalmente em sua compatibilidade com a linguagem/ bibliotecas Java, assim, Rodrigues (p. 46) complementa que:

> Esta linguagem apresenta muitas vantagens para o trabalho que se pretende realizar. Uma delas é clara a escalabilidade: ela foi desenhada para dar suporte a grandes sistemas. Uma outra é que permite interoperabilidade com a plataforma Java. Ou seja, o compilador do Scala reconhece e compila código fonte Java (Scala, 2013), o que permite, por exemplo, usar a aplicação JEcoLi, totalmente programada em Java, sem ser necessário fazer qualquer alteração para ser reconhecida, se necessário. A linguagem Scala reconhece métodos Java, acede a campos Java, implementa interfaces em Java e pode herdar classes em
> Java, sem qualquer problema.

Em se tratando da combinação dos paradigmas nesta linguagem, antes mencionados, a parte de orientação a objetos possibilita a fácil adaptação a grandes novas demandas, e já a parte da programação funcional torna rápida e fácil a criação de coisas simples. Sendo assim, traz Neto (2014): “Essa combinação torna possível a criação de novos padrões de programação e abstração de componentes, possibilitando também um estilo legível e mais conciso de programação”. E também Lago (2015): “Todo valor é um objeto e toda operação é uma chamada de método”.

Ainda sobre a parte da programação funcional dentro desta linguagem, (LAGO, 2015) a Scala traz uma estrutura desta completa, comportando funções como entidades de primeira classe, biblioteca padrão com estruturas de dados imutáveis (a imutabilidade em si é bastante favorecida pelo próprio estilo da linguagem), e o suporte nativo para o “casamento” de padrões. Vale constar que estas características, próximas ao modelo funcional, são implementadas a fim de simplificar o desenvolvimento de aplicações paralelas.

4.1. Escalabilidade e Expansão

Vale abordar os destaques da escalabilidade e expansão da linguagem Scala, começando com o seu fator de escalabilidade. O livro (LEMOS, 2019) *Akka in Action*, trazido por *Raymond Roestenburg* traz a ferramenta *Akka*, escrita em Scala, cujo objetivo é facilitar o desenvolvimento de aplicações paralelas, utilizando o modelo de programação de atores.

Assim, apresentando no livro, como exemplo, uma aplicação de cadastros de venda de ingressos para eventos. O destaque é o fato de o código apresentado ter a capacidade de lidar tanto com pequenos, quanto grandes volumes de dados e requisições (cenário real de aplicação), demonstrando em si a escalabilidade da tecnologia. Conforme traz Lemos (2019): “Em um projeto feito em Scala é possível escalá-lo para trabalhar em situações mais complexas, como por exemplo, um serviço consegue trabalhar com um número maior de usuários simultaneamente.”

E, sobre o cenário de expansão em que a linguagem está inserida, esta é demonstrada pelo fato de a Scala estar sendo requisitada por empresas de alto impacto global, e também por grandes áreas de crescimento como a *Data Science*, treinamento de inteligências artificiais, e processamento de fluxos de dados em tempo real.

Figura 2 - Empresas que utilizam comercialmente Scala. Disponível em: <http://www2.ic.uff.br/~bazilio/cursos/lp/material/Scala.pdf>.

5. Código

A seguir serão demonstrados os aspectos da linguagem Scala referente ao código em si, abordando características como variáveis (imutabilidade e inferência de tipos), classes, funções, estrutura de seleção e laço de repetição.

5.1. Expressões

Antes de abordar sobre os itens citados, vale ressaltar um ponto interessante sobre a linguagem Scala. Que é o seu conceito de Expressão. Como traz Odersky (2013): “Scala vem com um interpretador que pode ser visto como uma calculadora sofisticada. O usuário interage com a calculadora digitando expressões. A calculadora retorna o resultado do cálculo e o seu tipo de dado”. Conforme imagem abaixo:

```
scala> 87 + 145
unnamed0: Int = 232

scala> 5 + 2 * 3
unnamed1: Int = 11

scala> "hello" + " world!"
unnamed2: java.lang.String = hello world!
```

Figura 3 - Exemplo de expressão pelo interpretador Scala. Disponível em: <http://lampwww.epfl.ch/~hmiller/scala_by_example/ScalaByExample.pdf>.

Também uma sub-expressão com um valor, e invocá-la pelo nome. Utilizando a palavra reservada *def* (será utilizada também para as funções, que são expressões com

parâmetros) seguida do nome da expressão, e seu valor. Como segue exemplo em figura abaixo:

```
scala> def scale = 5
scale: Int

scala> 7 * scale
unnamed3: Int = 35
```

Figura 4 - Exemplo de expressão definida com valor. Disponível em: <http://lampwww.epfl.ch/~hmiller/scala_by_example/ScalaByExample.pdf>.

Tendo fundamentado as expressões em Scala, fecha Fidelis (2016) sobre a programação funcional nesta linguagem:

> Programação Funcional é um paradigma onde o código é composto de várias expressões (pode se pensar em funções que recebem um valor e sempre retornam algum valor) e não declarações, evitando ao máximo a mudança de estado de objetos e variáveis mutáveis. Então, ao utilizar Scala, você deve programar "orientado a expressões".

.

5.2. Variáveis

A sintaxe de declaração de variáveis em Scala é bastante simples, sendo esta (TORRES, 2019) de tal forma: *var* <nome da variável> : <tipo> = <valor>. Por exemplo: var Inteiro : Int = 3. Vale constar que é opcional informar o tipo da variável, visto que a linguagem dá suporte à inferência de tipos.

```
// 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
```

Figura 5 - Variáveis em Scala. Disponível em: <http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-2019 2-seminario-scala.pdf>.

Também há as constantes, trazendo o conceito da imutabilidade de objetos, onde seu valor não pode ser mudado em nenhum momento do programa. Sendo sua

sintaxe idêntica à das variáveis, apenas mudando a palavra reservada para *val* e não possuindo tipagem na declaração.

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

Figura 6 - Constantes em Scala. Disponível em: <http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-2019 2-seminario-scala.pdf>.

Vale constar que (TORRES, 2019), tanto as variáveis, quanto constantes, podem armazenar funções anônimas dentro de si, e não somente valores. Também é permitida atribuição múltipla para estas, tendo por exemplo variáveis que representam pares ordenados. Porém, não se pode utilizar este recurso em variáveis já inicializadas.

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

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

Figura 7 - Armazenamento de funções anônimas. Disponível em: <http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-2019 2-seminario-scala.pdf>.

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

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

Figura 8 - Atribuição Múltipla. Disponível em: <http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-2019 2-seminario-scala.pdf>.

```
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.
```

Figura 9 - Atribuição múltipla com variáveis já inicializadas. Disponível em: <http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-2019 2-seminario-scala.pdf>.

Por último, (NETO, 2014) tem se as variáveis *lazy*, ou "preguiçosas". O interessante destas variáveis é o fato de elas somente serem inicializadas, quando acessadas pela primeira vez. Como sugere o exemplo na figura abaixo, a variável tem o propósito de abrir um arquivo de texto. Porém, se esta variável não for acessada em nenhum momento do programa, este arquivo de texto nunca será aberto.

```
lazy val words = scala.io.Source.fromFile("/usr/share/dicts/words").mkString
```

Figura 10 - Exemplo de variável *lazy*. Disponível em: <https://cepein.femanet.com.br/BDigital/arqTccs/1111330685.pdf>.

5.3. Listas

A linguagem Scala (ODERSKY, 2013) também oferece suporte à estrutura de dados Lista, bastante importante em muitos programas Scala. São semelhantes aos vetores de linguagens como C ou Java, mas aqui vale destacar três pontos importantes, conforme traz Odersky (2013):

> Primeiro, as listas são imutáveis. Ou seja, elementos de uma lista não podem ser mudados por meio de atribuição. Segundo, listas tem uma estrutura recursiva, enquanto vetores são triviais. Terceiro, em geral, listas suportam um conjunto muito mais rico de operações que vetores.

```
val fruit: List[String]    = List("apples", "oranges", "pears")
val nums : List[Int]       = List(1, 2, 3, 4)
val diag3: List[List[Int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty: List[Int]       = List()
```

Figura 11 - Criando listas em Scala (1/3). Disponível em: <http://lampwww.epfl.ch/~hmiller/scala_by_example/ScalaByExample.pdf>.

As listas (ODERSKY, 2013) também podem ser construídas a partir de dois construtores, o "Nil" e "::" (operador infixo). "Nil" por si só representa uma lista vazia. Já o operador infixo expressa extensão da lista construída. Sendo assim, x :: xs representa "x" como sendo o primeiro elemento da lista, e "xs" como sendo restante (corpo) dela.

```
val fruit  = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums   = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3  = (1 :: (0 :: (0 :: Nil))) ::
             (0 :: (1 :: (0 :: Nil))) ::
             (0 :: (0 :: (1 :: Nil))) :: Nil
val empty  = Nil
```

Figura 12 - Criando listas em Scala (2/3). Disponível em: <http://lampwww.epfl.ch/~hmiller/scala_by_example/ScalaByExample.pdf>.

Vale constar que, o operador infixo é associativo à direita. Sendo assim, a criação da lista "A :: B :: C" pode ser interpretada como "A :: (B :: C)". Então, pode se remover os parênteses dos exemplos de listas demonstrados acima, obtendo o resultado conforme figura abaixo.

```
val nums  =  1 :: 2 :: 3 :: 4 :: Nil
```

Figura 13 - Criando listas em Scala (3/3). Disponível em: <http://lampwww.epfl.ch/~hmiller/scala_by_example/ScalaByExample.pdf>.

Finalizado, linguagem Scala também traz as operações básicas para o objeto de lista, sendo elas o *head* (retorna o primeiro elemento de uma lista), *tail* (retorna uma lista constituída de todos os elementos, exceto o primeiro elemento) e *isEmpty* (retorna True se a lista está vazia). Vale constar que *head* e *tail* são definidos apenas para listas que não estão vazias, ou seja, se invocadas por uma lista vazia, é disparada uma exceção. Estes métodos são invocados como métodos dos objetos de lista, sendo assim, são chamados a partir de uma desta, conforme figura abaixo.

```
empty.isEmpty   = true
fruit.isEmpty   = false
fruit.head      = "apples"
fruit.tail.head = "oranges"
diag3.head      = List(1, 0, 0)
```

Figura 14 - Métodos de listas. Disponível em: <http://lampwww.epfl.ch/~hmiller/scala_by_example/ScalaByExample.pdf>.

5.4. Estrutura de Seleção

As Estruturas de Repetição Scala são bem parecidas e se assemelham bastante com as que encontramos na linguagem Java. Pode-se verificar isso na figura a seguir:

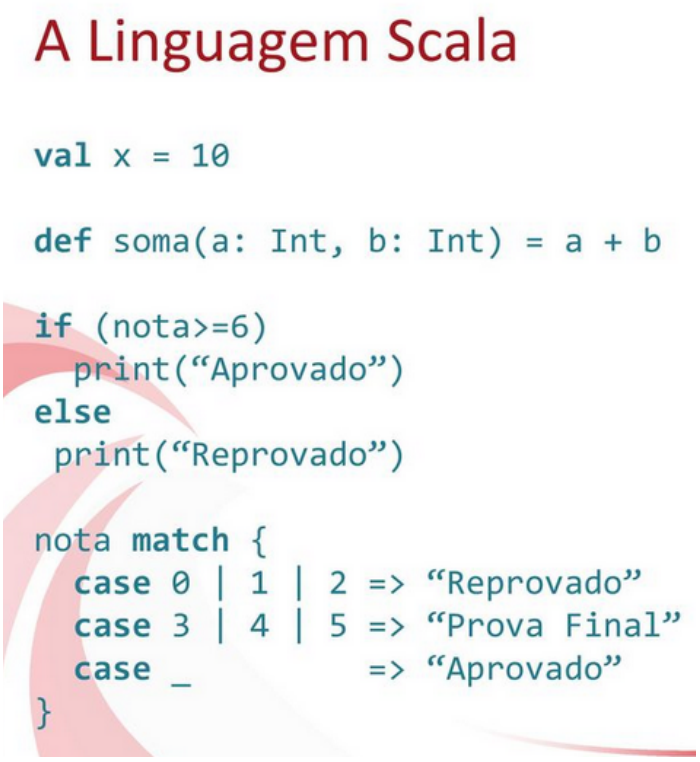


Figura 15 - Código em Scala (Seleção)

E no sentido mais estrito, programar sem variáveis mutáveis, atribuições, loops ou qualquer outra estrutura de controle imperativo, assim, programar sem efeitos colaterais. (LUCENA, 2012) Evitando que a linguagem seja corrompida e a sua codificação seja mais "Limpa".

5.5. Função

Scala é considerada uma linguagem funcional, no sentido que toda função é um valor. Além de fornecer um sintaxe leve para definir funções anônimas, que suporta funções de primeira ordem, permite funções aninhadas e suporta currying. As case classes da linguagem Scala e o suporte embutido para correspondência de padrões modelam tipos algébricos utilizados em muitas linguagens de programação funcional.

Assim reforça Rodrigues (p.52), com a definição de funções e suas utilidades na codificação, mostrando sua semelhança quando comparado a outras linguagens, especificamente, como Java:

> Em Scala existe distinção entre métodos e funções (Macbeath, 2014). Um método, tal como em Java, é parte de uma classe, tem nome, assinatura, possivelmente anotações. Uma função, em Scala, é um objecto completo. Em (Odersky, 2008) podemos ver que uma função é um objecto e, como tal, pode ser atribuída a uma variável, pode ser guardada como um valor. A linguagem apresenta vários traits que permitem implementar estas – Function0, Function1,..., Function22 – sendo que a diferença entre elas é o número de argumentos que permitem a uma função receber (Macbeath, 2014) (Odersky, 2008). Sendo uma função uma instância de uma classe, naturalmente vai ter métodos. Um desses métodos é o método apply(), sendo este o que contém o código que implementa o corpo de uma função (Odersky, 2008) (Macbeath, 2014). (RODRIGUE, 2014)

```
object teste {
  def method(x: Int) = x + 1                    > method: (x: Int)Int

  val function = (x: Int) => x +1               > function  : Int => Int = <function1>

  val function2 = (x:Int) =>  method(x)         > function2  : Int => Int = <function1>
}
```

Figura 16 - Exemplificação de um código em Scala

É de se perceber que foi definido um método e duas outras funções que possuem seus respectivos processos dentro do Object teste. Na primeira função ele retorna a variável x somado com um, e a função dois realiza a mesma operação, porém chamando o método neste caso.

5.6. Estrutura de Repetição

Em se tratando das estruturas de repetição, (NETO, 2014) a linguagem scala possui os tradicionais laços de interatividade *while* e *for*, semelhantes ao C++. Porém com certas diferenças quanto à sintaxe. Serão demonstradas as suas sintaxes e exemplos através de figuras para ambas as estruturas.

Para o *for*, tem se a variável contadora e o valor limite do qual ela deve alcançar para cessar o loop. Na linguagem Scala, pode se, opcionalmente, adicionar uma condição de teste ao *for* também. Segue sintaxe: *for* (<variável contadora> *to* <limite> <condição de teste/expressão booleana opcional>) e figura de exemplo abaixo.

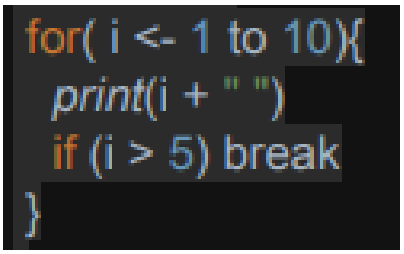


Figura 17 - Exemplo de laço *for*. Disponível em: <http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-2019 2-seminario-scala.pdf>.

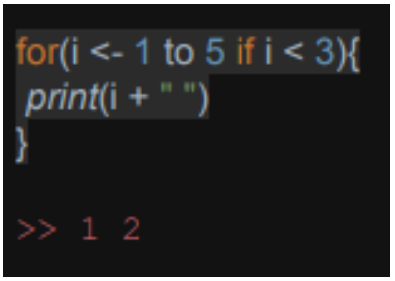


Figura 18 - Exemplo de laço *for* com condicional de teste. Disponível em: <http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-2019 2-seminario-scala.pdf>.

Já o *while* se dá de forma tradicional (TORRES, 2019), bastante semelhante ao C++. Com a sintaxe: *while (*<expressão booleana>*)*. Conforme exemplo em figura abaixo:

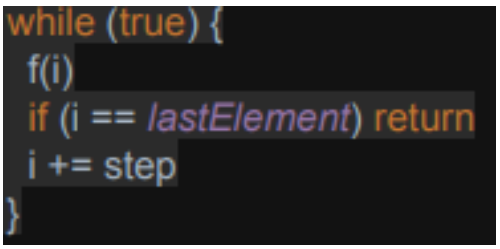


Figura 19 - Exemplo de laço *while*. Disponível em: <http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-2019 2-seminario-scala.pdf>.

Vale constar o suporte à recursividade, demonstrada como a principal/tradicional alternativa para a programação funcional em se tratando de repetição, através de funções. Seguem exemplos abaixo de recursividade e utilização do *mapping*.

```
def fac(n: Int): Int = if (n <= 0) 1 else n * fac(n - 1)
```

Figura 20 - Exemplo de função recursiva (Fatorial) em Scala. Disponível em: <https://cepein.femanet.com.br/BDigital/arqTccs/1111330685.pdf>.

```
List(1,2).map( (x : Int) => x + 1 )
```

Figura 21 - Map em Scala. Disponível em: <http://www2.ic.uff.br/~bazilio/cursos/lp/material/Scala.pdf>.

5.7. Classe

As definições de classes em Scala (NETO, 2014) são bastante semelhantes a Java e C++, como por exemplo permitindo a utilização de construtores customizados (em Scala utilizando *this*). Porém, vale o destaque para certas características da linguagem: os campos criados em classes da Scala já possuem seus *getters* e *setters* e todas as classes possuem construtor primário (entrelaçado em sua definição).

A linguagem também suporta a Herança, com os seus conceitos e funcionalidades tradicionais (herdar atributos, métodos, possibilidade de sobrescrever métodos, entre outras). Sobre esta característica, as classes em Scala (ODERSKY, 2013) sempre estendem alguma outra.

Quando esta herança não é definida explicitamente pelo programador (*extends* <nome-da-classe>), é implicitamente estendida a classe *anyRef*, como o *java.lang.Object* do Java. E, a ponto de curiosidade, quando se deseja sobrescrever um método herdado, este deve ser precedido da palavra *override*.

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // como antes
  override def toString = "" + numer + "/" + denom
}
```

Figura 22 - Sobrescrever método herdado em Scala. Disponível em: <http://lampwww.epfl.ch/~hmiller/scala_by_example/ScalaByExample.pdf>.

Vale constar que (NETO, 2014) uma classe, nesta linguagem, não é declarada como criada pública na sua criação. Sendo assim, um código-fonte de Scala possui todas as suas classes com visibilidade pública. A linguagem também suporta o conceito de classes abstratas (*abstract class*).

E também, a declaração de seus métodos se dá da mesma forma da criação de funções, com a diferença de se tipificar o seu retorno, como sugere a sintaxe: *def* <nome da função>(<parâmetros>) : <tipo do retorno> = <corpo da função>. Segue

abaixo figura para exemplificar, tendo a classe *NonEmptySet* com os métodos *contains* e *incl*.

```
class NonEmptySet(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: Int): IntSet =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
}
```

Figura 23 - Exemplo de método dentro de classe. Disponível em: <http://lampwww.epfl.ch/~hmiller/scala_by_example/ScalaByExample.pdf>

Os seus *getters* e *setters* trabalham de forma idêntica ao Java, e, como mencionado anteriormente, são criados de forma automática para cada atributo da classe. O desenvolvedor deve se atentar para o fato de todos os atributos da classe serem inicializados com um valor inicial atribuído para tais. Também os atributos podem ser privados (*private*) ou públicos.

```
class Pessoa {
  var idade = 0
  var nome = null
}
```

Figura 24 - Classe com atributos em Scala. Disponível em: <https://cepein.femanet.com.br/BDigital/arqTccs/1111330685.pdf>.

Contudo, a linguagem Scala traz um ponto interessante no quesito de escopo privado de variáveis. Uma classe pode acessar um atributo privado de outro objeto, a menos que seja definido, explicitamente, este pode somente ser acessado dentro do próprio objeto, estando "bloqueado" definitivamente para acessos externos.

Sendo assim, traz Neto (2014) um exemplo prático, através de duas figuras consecutivas abaixo. Tem se a classe Counter, com seu objeto *value*. Observe que a primeira imagem demonstra um método da classe Counter acessando o *value* de outra classe Counter enviada como parâmetro para esse. Agora a segunda figura, demonstra como o atributo *value* pode ser criado de modo que este acesso não seja permitido.

```
class Counter {
  private var value = 0
  def increment() { value += 1 }
  def isLess(other: Counter) = value < other.value
  // Pode acessar o atributo privado do outro objeto
}
```

Figura 25 - Exemplo de Neto (1/2), uma classe acessando um atributo *private* diretamente. Disponível em: <https://cepein.femanet.com.br/BDigital/arqTccs/1111330685.pdf>.

```
private[this] var value = 0 // O acesso ao compo 'value' de um outro
                            //objeto dessa classe não é permitido
```

Figura 26 - Exemplo de Neto (2/2), definindo um atributo privado que pode ser acessado somente, e somente, dentro do mesmo objeto instanciado. Disponível em: <https://cepein.femanet.com.br/BDigital/arqTccs/1111330685.pdf>.

5.7.1 Construtores

Sobre os construtores de classes em Scala, primeiramente (NETO, 2014) tem se que é possível criar classes sem construtores. Uma vez feito isso, é estabelecido o chamado construtor primário (sem parâmetros) para ela, implicitamente. Em se tratando disso, há dois tipos de construtores, o primário citado neste momento, e os construtores auxiliares. Ressaltando que o número de construtores que uma classe em Scala pode ter, é indefinido.

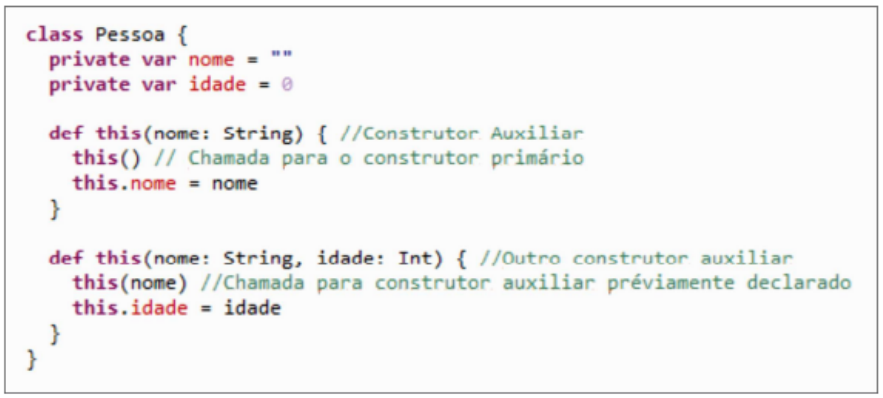
Sobre o Construtor Primário, pode se dizer que todas as suas declarações são executadas no momento de definição desta mesma classe. Os parâmetros do construtor primário se tornam os atributos dela, e é possível ligar métodos ao construtor primário, que assim serão executados no mesmo momento de instanciação do objeto, citado anteriormente.

Observe a imagem abaixo como forma de exemplo, tendo a classe Pessoa, recebendo parâmetros “nome” e “idade” (construtor primário recebendo estes). Note também os métodos *println* e *descricao* ligados a ele. O *println* é executado logo após a instanciação do objeto desta classe.

```
class Pessoa(val nome: String, val idade: Int) {
  println("Mais uma pessoa construída...")
  def descricao = nome + " têm " + idade + " anos de idade"
}
```

Figura 27 - Classe Pessoa com construtor primário, com parâmetros e métodos ligados a ele. Disponível em: <https://cepein.femanet.com.br/BDigital/arqTccs/1111330685.pdf>.

Já os construtores auxiliares, (NETO, 2014) são outros construtores definidos explicitamente para a classe. Podem existir vários destes para uma mesma classe, assim como na linguagem Java. Estes construtores devem ter a palavra reservada *this* os precedendo, para referenciar a classe em si, definindo seu construtor auxiliar. Cada um pode receber os parâmetros que bem desejar. Vale constar que estes podem invocar o construtor primário da classe, se for o caso.



```
class Pessoa {
  private var nome = ""
  private var idade = 0

  def this(nome: String) { //Construtor Auxiliar
    this() // Chamada para o construtor primário
    this.nome = nome
  }

  def this(nome: String, idade: Int) { //Outro construtor auxiliar
    this(nome) //Chamada para construtor auxiliar préviamente declarado
    this.idade = idade
  }
}
```

Figura 28 - Classe com construtores auxiliares, cada um com seus parâmetros e comandos diferentes. Disponível em: <https://cepein.femanet.com.br/BDigital/arqTccs/1111330685.pdf>.

5.8. Traits

Uma trait seria um conceito usado em programação orientada a objetos, que representa um conjunto de métodos que podem ser usados para estender a funcionalidade de uma classe. Dessa forma fornecem um conjunto de métodos que implementam o comportamento de uma classe e exigem que a classe implemente um conjunto de métodos que parametrizam o comportamento fornecido.

Dessa forma, o comportamento de um tipo consiste nos métodos que podemos chamar para aquele tipo. Tipos diferentes dividem o mesmo comportamento se podemos chamar os mesmos métodos em todos esses tipos. Definições de traits são um modo de agrupar métodos de assinaturas juntos a fim de definir um conjunto de comportamentos para atingir algum propósito.

E para entender melhor Rodrigues (p. 47) afirma e acrescente essas informações referente aos conceitos de traits e suas funcionalidades:

> Um Trait é um módulo como uma interface, uma classe, e que, tal com estes, permite composição. Possui uma sintaxe em tudo parecida ao da classe (ver Figura 5. 1), no entanto apresenta diferenças significativas que torna este mecanismo uma poderosa ferramenta, no que toca a reutilização de código em Scala (Odersky et al, 2008). Encapsula métodos e membros privados, tal e qual uma classe, podendo ser depois reutilizada por uma classe, através de uma técnica chamada mixin (Odersky et al, 2008).

Comparando com o Java é equivalente a uma interface mas muito mais poderoso, uma vez que pode conter estado e comportamento (para modificar o estado e/ou para dar a conhecê-lo).

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

Figura 29 - Exemplo de um trait

Conforme a figura acima, foi instanciado uma trait do tipo Philosophical, onde possui uma função, e dentro dessa possui suas funcionalidades.

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

Figura 30 - Trait em uso

Agora podemos ver uma classe Frog que herdou os métodos da trait criada anteriormente , podendo realizar um override dos mesmo.

6. Conclusão

Com todo o conteúdo trazido até aqui, pode-se concluir que, a linguagem de programação Scala se mostra interessante para a aplicação da Programação Funcional, juntamente com a experimentação da mistura deste paradigma com a Orientação a Objetos. Vale constar a semelhança com linguagens populares como Java e C, tornando seu aprendizado mais fluido. Assim, conferindo novas possibilidades de lógicas e implementação para pôr em prática, trazendo para os desenvolvedores uma forma diferente de abstrair e programar.

7. Referências

BAZILIO, C. **Introdução à Linguagem Scala**. Disponível em: <http://www2.ic.uff.br/~bazilio/cursos/lp/material/Scala.pdf>. Acesso em: 09/05/2022.

FIDELIS, Bruno. **Conceitos de Linguagem Funcional com Scala.** Disponível em: <https://king.host/blog/2016/10/conceitos-de-programacao-funcional-com-scala/>. Acesso em: 11/05/2022.

LAGO, Nelson. **A linguagem de programação Scala e o arcabouço para concorrência Akka**. Disponível em: <https://www.ime.usp.br/~gold/cursos/2015/MAC5742/reports/scala.pdf>. Acesso em: 02/05/2022.

LEMOS, Sandra. **Por que você deve aprender Scala?** Disponível em: <https://insightlab.ufc.br/por-que-voce-deve-aprender-scala>. Acesso em: 09/05/2022.

ODERSKY, Martin. **Scala Através de Exemplos.** Disponível em; <http://lampwww.epfl.ch/~hmiller/scala_by_example/ScalaByExample.pdf>. Acesso em: 11/05/2022.

NETO, Daniel Munhoz Moreno. **Estudo descritivo sobre a linguagem de programação Scala**. Disponível em: <https://cepein.femanet.com.br/BDigital/arqTccs/1111330685.pdf>. Acesso em: 02/05/2022.

RODRIGUES, A, M, O, A. **Exploração da linguagem Scala para suporte a aplicações paralelas**. Disponível em: <https://run.unl.pt/bitstream/10362/14782/1/Rodrigues_2014.pdf>. Acesso em: 09/05/2022.

TORRES, Lucas Tabelini. et al. **Scala**. Disponível em: <http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-20192-seminario-scala.pdf>. Acesso em: 02/05/2022.