Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

DANIEL MUNHOZ MORENO NETO

ESTUDO DESCRITIVO SOBRE A LINGUAGEM DE PROGRAMAÇÃO SCALA

Assis

2014

DANIEL MUNHOZ MORENO NETO

ESTUDO DESCRITIVO SOBRE A LINGUAGEM DE PROGRAMAÇÃO SCALA

Monografia apresentada ao curso de Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e à Fundação Educacional do Município de Assis – FEMA como requisito parcial à obtenção do Certificado de Conclusão de Curso.

Orientador: Me. Douglas Sanches da Cunha

Área de Concentração: Informática

Assis

2014

FICHA CATALOGRÁFICA

MORENO, Daniel Munhoz Neto
Estudo Descritivo Sobre a Linguagem de Programação Scala / Daniel Munhoz Moreno Neto. Fundação Educacional do Município de Assis – FEMA – Assis, 2014.
57p.

Orientador (a): Me. Douglas Sanches da Cunha.
Trabalho de Conclusão de Curso – Instituto Municipal de Ensino Superior de Assis – IMESA.

1. Scala. **2**. Orientação a Objetos. **3**. Programação Funcional. **4**. Java

CDD: 001.6
Biblioteca FEMA

DANIEL MUNHOZ MORENO NETO

ESTUDO DESCRITIVO SOBRE A LINGUAGEM DE PROGRAMAÇÃO SCALA

Monografia apresentada ao curso de Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e à Fundação Educacional do Município de Assis – FEMA como requisito parcial à obtenção do Certificado de Conclusão de Curso.

Orientador: Me. Douglas Sanches da Cunha

Examinador: Esp. Célio Desiró

Assis

2014

RESUMO

Com a lenta evolução das linguagens Java e C++, os programadores precisam optar por linguagens que possam atendê-los nos mais amplos âmbitos de necessidade. Uma linguagem que não fique presa a uma determinada plataforma, ou a um único paradigma é uma boa escolha para estes programadores que também buscam soluções para os problemas atuais de concorrência e escalabilidade. Para atender a estes propósitos é que a linguagem Scala fora criada. Scala é uma linguagem que opera com a Java Virtual Machine, com a plataforma .NET e também em ambientes Android. Seu código pode ser escrito de forma imperativa ou funcional, não limitando assim os usuários a trabalharem com um único paradigma. Além destas características, a linguagem oferece total compatibilidade com bibliotecas, frameworks e códigos escritos nas linguagens Java e C#.

Palavras-chave: SCALA, ORIENTAÇÃO A OBJETOS, PROGRAMAÇÃO FUNCIONAL, JAVA.

ABSTRACT

With the slow evolution of Java and C ++ languages, programmers must choose languages that can serve them in broader areas of need. A language that does not get attached to a particular platform, or to a single paradigm is a good choice for those programmers who are also seeking solutions to current problems of competition and scalability. To serve these purposes is that Scala is made. Scala is a language that operates on the Java Virtual Machine, with NET platform and also on Android environments. Your code can be written in the imperative or functional form, thus not limiting users to work with a single paradigm. Besides these features, the language offers full compatibility with libraries, frameworks and code written in Java and C # languages.

Keywords: SCALA, OBJECT ORIENTATION, FUNCTIONAL PROGRAMMING, JAVA.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1. INTRODUÇÃO

Com um avanço tecnológico tão grande sobre os *hardwares* nos dias de hoje, é essencial que os aplicativos e sistemas criados estejam preparados para usufruir de todos os recursos oferecidos por essas novas tecnologias.

Com a implantação de mais núcleos na UCP (Unidade Central de Processamento) e o aumento na demanda de acessos a jogos online, redes sociais e as demais aplicações da World Wide Web (www) que requerem acesso simultâneo de milhares e até mesmo milhões de pessoas, uma abordagem concorrente se faz necessária no desenvolvimento destes sistemas.

Essa concorrência é disponibilizada por diversas linguagens e com diversos modelos. O modelo mais adotado atualmente é o baseado em *threads*, que faz uso de memória compartilhada. Segundo Salgado (2012), *thread* é uma forma do processo se dividir em tarefas menores para que possa ser executado paralelamente. Em processadores com um núcleo, as *threads* são processadas tão rapidamente que é passada ao usuário uma falsa sensação de paralelismo, enquanto em processadores multi-core, o processamento é sim feito de forma paralela, dividindo as *threads* entre os núcleos do processador onde são processadas de formas individuais.

Outro modelo de programação paralela também apresentado é o modelo baseado em troca de mensagens, adotado pela linguagem Scala, linguagem esta que será abordada durante este trabalho. Este modelo consiste em realizar comunicação entre as entidades computacionais que são denominadas atores por meio de troca de mensagens. Este modelo evita a necessidade do compartilhamento de memória exigido quando se trabalha com *threads*, porém, pode adicionar uma sobrecarga (overhead) no tempo de execução da aplicação devido as trocas de mensagens.

A linguagem Scala introduz várias inovações em termos de linguagem de programação, como:

- Abstração de tipos e conceitos unificados de objetos e módulos de sistema.
- Unificação dos paradigmas de orientação a objetos e de programação funcional.

• Sintaxe flexível de definições simples de tipos que contribuem para a criação de avançadas bibliotecas para a linguagem.

A completa integração com Java, uma das linguagens mais utilizadas atualmente, possibilita o uso de bibliotecas e frameworks desenvolvidos para a linguagem da ORACLE sem que haja a necessidade de declarações adicionais.

1.1 OBJETIVO

Este trabalho tem como objetivo geral abordar as características principais da linguagem Scala, buscando expor de forma simples as tecnologias e inovações implementadas na linguagem criada por Martin Odersky.

Para conclusão do objetivo geral, foi planejada primeiramente uma apresentação da linguagem, buscando mostrar seu diferencial perante as outras linguagens encontradas atualmente e uma breve explicação sobre os paradigmas de programação apresentados por ela. Após esta explicação as características da linguagem serão apresentadas de uma forma mais profunda, buscando expor de forma detalhada o funcionamento de seus métodos, classes e objetos. Por final, a implementação de um simples cadastro usando a integração de frameworks Java para demonstrar a simplicidade do uso de códigos escritos em Scala em conjunto com códigos escritos em Java.

1.2 JUSTIFICATIVA

O forte crescimento apresentado pela linguagem devido as suas inovações e a sua adoção por empresas de grande porte no mercado mundial como o Twitter, Linkedin, Sony e Siemens foram fatores que influenciaram a elaboração do projeto. A linguagem apresenta potencial para alcançar Java devido a todas as suas qualidades. As vantagens de a linguagem ser de propósito geral, se fazer presente nas plataformas JVM (Java Virtual Machine), Androide .NET e ainda oferecer os paradigmas de programação orientada a objetos e programação funcional são fatores indiscutíveis para o sucesso da linguagem no mercado atual, onde

processamento rápido e concorrente não pode mais ser deixado de lado pelas empresas.

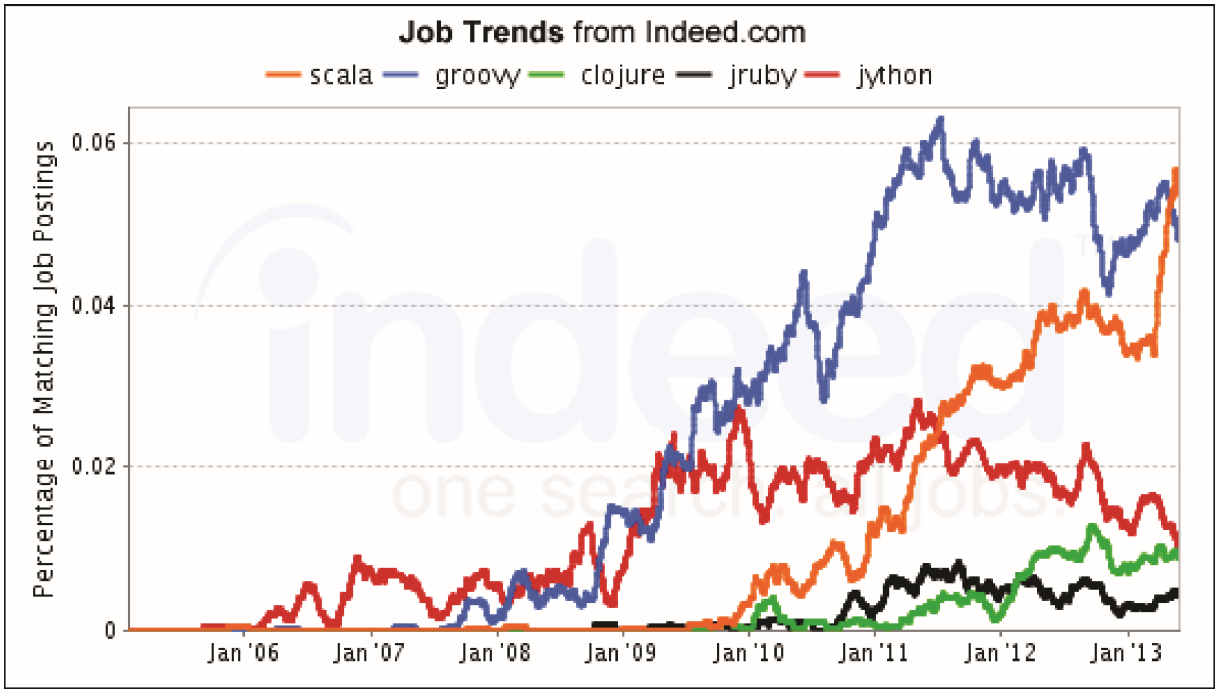


Figura 1 – Tendência de trabalho para as linguagens que trabalham com JVM (Fonte: Artigo sobre Scala no site randonom[1])

1.3 PROBLEMATICA

A ideia para o desenvolvimento deste projeto teve como base diversos aspectos que se fazem presentes nos dias atuais, como a modernização rápida dos dispositivos de *hardware* e uma forma de se tirar proveito de todas essas vantagens oferecidas por meio de concorrência e utilização constante do processador, evitado ociosidade entre os processos.

[1]Disponível em: < http://randonom.com/blog/category/scala/> Acesso em set. 2014.

1.4 ESTRUTURA DE DESENVOLVIMENTO

Este trabalho está estruturado nas seguintes partes:

- **Capítulo 1 – Introdução**
- **Capítulo 2 – Linguagem de Programação Scala**
- **Capítulo 3 – Paradigmas de Programação**
- **Capítulo 4 – Características da Linguagem**
- **Capítulo 5 – Implementação**
- **Capítulo 6 – Conclusão**
- **Referências Bibliográficas**

2. LINGUAGEM DE PROGRAMAÇÃO SCALA

O intuito deste capitulo é apresentar a linguagem de programação Scala por meio da apresentação de fatos e características sobre a mesma.

2.1 A LINGUAGEM

Scala é a mistura dos nomes “*scalable*” e “*language*”. Esse foi o nome escolhido para a linguagem considerada a mais suscetível a substituir o tão forte Java. Denominada assim por crescer junto com as necessidades do usuário, Scala pode ser utilizada desde a escrita de pequenos scripts até construção de grandes sistemas.

Scala é uma linguagem multiparadigma, pois emprega métodos de programação funcional e é totalmente orientada a objetos, além de ser multiplataforma, já que roda em três diferentes plataformas: Android, JVM (Java *Virtual Machine*), e .NET (ODERSKY; SPOON; VENNERS, 2008, p.39).

Figura 2 – Logotipo linguagem Scala

Quando compilado, o código Scala gera Java bytecodes, portanto, é possível a utilização de qualquer biblioteca Java sem a necessidade de interfaces ou cópia de código (ODERSKY; SPOON; VENNERS, 2008, p.39).

Ela é uma ótima linguagem para quando se deseja criar *scripts* que incorporam vários componentes Java, mas sua força é ainda mais notável quando empregada na construção de grandes sistemas e *frameworks* onde componentes reutilizáveis

são necessários. Tecnicamente, Scala é a mistura de orientação a objetos e conceitos de programação funcional em uma linguagem estaticamente tipada. A fusão desses conceitos pode ser observada em diferentes aspectos da linguagem, ainda mais quando fala-se em escalabilidade. A parte funcional da linguagem torna fácil e rápida a construção de coisas simples, enquanto a parte orientada a objeto possibilita fácil adaptação de grandes a novas demandas. 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, que de tão maleável, torna a programação em Scala divertida (ODERSKY; SPOON; VENNERS, 2008, p.39).

Eric Raymond (1999 apud ODERSKY; SPOON; VENNERS, 2008, p. 41) diz que podemos usar uma catedral e um bazar como duas metáforas para construção de softwares.

> *The cathedral is a near-perfect building that takes a long time to build. Once built, it stays unchanged for a long time. The bazaar, by contrast, is adapted and extended each day by the people working in it.*

Scala parece muito mais com um bazar do que com uma catedral se avaliarmos a citação de Raymond. A linguagem não oferece ao usuário tudo o que ele necessita, fazendo dela uma linguagem perfeita, ao invés disso, ela disponibiliza todas as ferramentas necessárias para que o usuário crie o que ele precisar (ODERSKY; SPOON; VENNERS, 2008, p.41).

2.2 HISTÓRIA

Desenvolvida em 2001 na École Polytechnique Fédérale de Lausanne (EPFL), Lausana na Suíça, por Martin Odersky (ex-integrante do Java *Generics*), a linguagem Scala foi liberada publicamente na plataforma Java em janeiro de 2004.Sua versão alternativa para a plataforma .NET foi publicada pouco tempo

depois, em junho do mesmo ano. A segunda versão da linguagem foi liberada em março de 2006 (ODERSKY, 2014, p.01).

Scala é o resultado do esforço coletivo de diversas pessoas. O *design* e as implementações da versão 1.0 foram feitas por Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, e Martin Odersky. Iulian Dragos, Gilles Dubochet, Philipp Haller, Sean McDirmid, Lex Spoon, e Geoffrey Washburn juntaram-se a equipe para ajudar no lançamento da segunda versão da linguagem e suas ferramentas. Gilad Bracha, Craig Chambers, Erik Ernst,Matthias Felleisen, Shriram Krishnamurti, Gary Leavens, Sebastian Maneth, Erik Meijer, Klaus Ostermann, Didier Rémy, Mads Torgersen, e Philip Wadler deram forma ao design da linguagem através de várias discussões e análises do documento de especificações da linguagem, atentando-se também aos inscritos no grupo de e-mails de linguagem, que contribuíram bastante para que ela chegasse ao estado em que se encontra hoje (ODERSKY, 2014, p. 01).

2.3 ARQUITETURA

Com o vasto crescimento dos processadores multi-core, o uso de paralelismo nas aplicações, torna-se cada vez mais necessários. Para fazer com que isso aconteça, é necessária a reescrita de código para que o processamento seja distribuído em diversos *threads*, no entanto, a criação dessas aplicações, provou ser um desafio (ODERSKY; SPOON; VENNERS, 2008, p.42-45).

O modelo de *threads* de Java baseado em distribuição de memória e *locking* oferecem um pouco de dificuldade quando falamos em sistemas grandes e complexos devido ao fato dos *deadlocks*, que nem sempre se apresentam nos testes, mas sim em produção. Uma alternativa muito mais segura que a proposta em Java é a arquitetura baseada em trocas de mensagens, bastante parecida com o modelo de atores usado pela linguagem Earlang. Java possui uma biblioteca muito rica baseada em *threads* que pode ser usada por programadores Scala assim como qualquer outra biblioteca Java, no entanto, Scala conta com uma biblioteca adicional

que essencialmente implementa o modelo de atores adotado na linguagem Earlang (ODERSKY; SPOON; VENNERS, 2008, p.42-45).

Os atores são uma abstração de concorrência que podem ser implementados por meio de *threads* que se comunicam entre si trocando mensagens. Um ator realiza duas operações básicas, enviar e receber mensagens. A operação de envio de mensagem é denotada por um ponto de exclamação (!). A operação de enviar mensagem é assíncrona, o que quer dizer que o ator que está enviando a mensagem não precisa esperar para que a mesma seja recebida e processada pelo destinatário. Cada ator possui uma caixa de mensagens, onde as mensagens que são recebidas ficam armazenadas na forma de uma fila. Os atores trabalham com as mensagens armazenadas por meio de um bloco de recebimento (ODERSKY; SPOON; VENNERS, 2008, p.42-45).

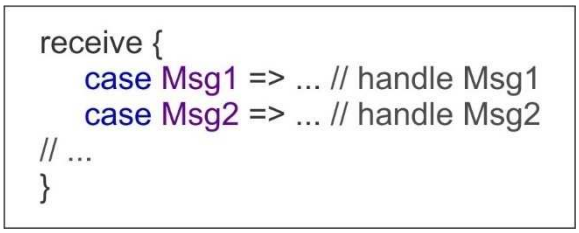
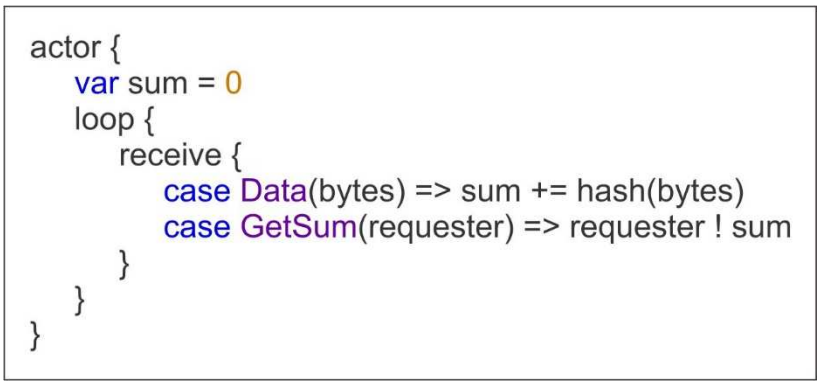


Figura 3 – Bloco de recebimento dos atores

(In: Odersky; Spoon; Venners, 2008, p.43)

O bloco de recebimento consiste em vários processos que consultam um padrão de mensagens dentro da caixa de mensagens. A primeira mensagem da caixa de mensagens que corresponder ao processo selecionado será executada. Se a caixa de mensagens não possuir nenhuma mensagem que corresponda ao processo exigido, o ator suspende e aguarda mensagens futuras (ODERSKY; SPOON; VENNERS, 2008, p.42-45).

A frase a seguir apresenta um exemplo simples da implementação de atores em Scala.



```
actor {
    var sum = 0
    loop {
        receive {
            case Data(bytes) => sum += hash(bytes)
            case GetSum(requester) => requester ! sum
        }
    }
}
```

Figura 4 – Exemplo de implementação de atores em Scala

(In: Odersky; Spoon; Venners, 2008, p.44)

Primeiramente, é definida uma variável local com o nome de *sum* com o valor inicial zero. Entra-se então em um *loop* a espera de mensagens. Essas mensagens, caso forem do tipo *Data message*, terá seus valores adicionados a variável *sum*, porém, se forem do tipo *GetSummessage*, o valor atual da variável sum será retornado ao requisitante. Esse método de atores não poderia ser implementado caso não houvesse a fusão dos conceitos de orientação a objetos e programação funcional na linguagem Scala. Sendo assim, pode-se destacar essa mistura como sendo o aspecto mais importante para que Scala seja uma linguagem escalável. Outras linguagens como Oz e Groovy tentaram fundir esses dois tipos de paradigmas, mas funções e objetos eram dois conceitos diferentes, diferentemente de Scala, onde o valor de uma função é um objeto e tipos de funções de classes que podem ser herdadas por subclasses (ODERSKY; SPOON; VENNERS, 2008, p.42-45).

2.4 CARACTERÍSTICAS

A seguir serão apresentadas algumas características que destacam a linguagem Scala.

2.4.1 COMPATIBILIDADE COM JAVA

Scala não exige que códigos escritos em Java sejam reescritos para Scala. A linguagem foi projetada para trabalhar em perfeita harmonia com Java. Sua compilação gera bytecodes que serão executados na JVM (Java *Virtual Machine*) e a performance em tempo de execução dos programas escritos em Scala são comparados aos escritos em Java (ODERSKY; SPOON; VENNERS, 2008, p.48-49).

Outro aspecto a ser destacado sobre a harmonia presente entre linguagens é o reuso que Scala faz dos tipos de Java. Em Scala, uma variável do tipo inteiro, é representada por uma variável primitiva inteira de Java, e isso acontece também com as variáveis de tipo *String* e booleanas. Um exemplo disto, uma variável *String* em Scala é do tipo java.*lang.String* (ODERSKY; SPOON; VENNERS, 2008, p.48-49).

Além do reuso dos tipos de Java, a linguagem Scala também os mascara, tornando-os melhores. Uma variável do tipo *String* em Scala oferece suporte a métodos como *toInt* e *toFloat*, métodos estes que convertem uma variável do tipo *String* para uma do tipo *Int* ou *Float* respectivamente. Esses métodos não se encontram disponíveis na classe *String* de Java, sendo assim, quem se encarrega dessa conversão é o compilador Scala, que quando encontra o trecho do código no qual a invocação do método é feita, procura na classe *String* o método, porém não o encontrará. O que será feita é uma conversão implícita, na qual a *String* Java será convertida para uma instancia da classe Scala *RichString*, onde se encontra a implementação do método (ODERSKY; SPOON; VENNERS, 2008, p.48-49).

2.4.2 CÓDIGOS CONCISOS

Alguns programadores Scala afirmam que um código escrito em Scala apresenta aproximadamente metade do número de linhas de um mesmo programa escrito em Java (ODERSKY; SPOON; VENNERS, 2008, p.49-51).

Essa redução acontece porque a linguagem Scala tenta evitar alguns clichês apresentados na linguagem Java, como por exemplo, o uso de ponto e vírgula, que em Scala é opcional, e por isso, muitas vezes deixado de lado pelos programadores.

Outro aspecto que ajuda essa compactação do código Scala é a não repetição de informações sobre tipos, deixando assim o código mais conciso.

Provavelmente, o fator mais importante para a diminuição no tamanho do código é o código que não precisa se escrever, porque já está escrito em bibliotecas. Scala oferece diversas ferramentas para criação de bibliotecas que expressam comportamento comum, e uma exemplo dessas ferramentas são as *traits* (ODERSKY; SPOON; VENNERS, 2008, p.49-51).

Como exemplo da diferença existente estre código Java e código Scala, segue exemplo de duas implementações de uma classe e seu construtor. A primeira mostra o exemplo de implementação de uma classe chamada *MyClass* em Java.

```
class MyClass {
  private int index;
  private String name;
    public MyClass(int index, String name) {
      this.index = index;
      this.name = name;
    }
}
```

Figura 5 – Exemplo de implementação de classe em Java

(In: Odersky; Spoon; Venners, 2008, p.50)

A imagem a seguir apresenta como seria implementada a mesma classe *MyClass* em Scala

```
class MyClass(index: Int, name: String)
```

Figura 6 – Exemplo de implementação de classe em Scala

(In: Odersky; Spoon; Venners, 2008, p.50)

Os códigos em Scala são mais rápidos para ser escritos, melhores para serem lidos e o mais importante, menos suscetíveis a erros.

2.4.3 TIPAGEM ESTÁTICA

Um sistema de tipagem estática classifica variáveis e expressões de acordo com o tipo de dados que elas armazenam. Scala é uma linguagem que apresenta como um de seus pontos fortes um sistema muito avançado de tipagem estática. Assim como Java, que apresenta muitos tipos aninhados de variáveis, em Scala também é possível se trabalhar com parametrização de tipos usando *generics,* com combinações usando *intersection* e também esconder os tipos de dados com que se trabalha usando *abstract*. Essa gama de opções possibilita ao usuário criar seus próprios tipos de dados, e a criarem interfaces que possam ser seguras e flexíveis as suas necessidades (ODERSKY; SPOON; VENNERS, 2008, p.52-55).

Há no entanto, quem não aprecie esta forma de tipagem, argumentando que ela tira do usuário a liberdade para se expressar como gostaria e que ela impede certos padrões de modificações dinâmicas em *softwares*, além de deixar o código gerado muito verboso. Em contra partida a esses argumentos, Scala evita a verbosidade através da inferência de tipos e o usuário ganha flexibilidade através da correspondência de tipos e das novas formas de se escrever e compor tipos. Sem estes impedimentos, pode-se aproveitar os benefícios da tipagem estática, que entre eles, vale ressaltar as propriedades verificáveis do programa, uma refatoração segura e uma melhor documentação (ODERSKY; SPOON; VENNERS, 2008, p.52-55).

Graças as verificações de propriedades, as linguagens com tipagem do tipo estático já não apresentam alguns erros de execução, ela também evita certos erros como, por exemplo, somar uma variável booleana a uma inteira ou permitir que uma variável privada seja acessada por uma classe externa. No entanto, outros tipos de erros ainda não são detectados pelas linguagens estáticas atuais, como por exemplo, terminação de funções e violações ao tamanho de *Arrays* e divisão de números por zero. Devidos à esses problemas, linguagens estáticas foram sendo deixadas de lado devido a argumentos de que somente erros simples eram encontrados pela linguagem, e que os testes unitários forneceriam uma ampla cobertura aos erros que escapavam das linguagem estáticas. Verdadeiramente, os testes unitários oferecem esta amplitude muito maior sobre as linguagens estáticas, porém, a redução no número de testes unitários em certas propriedades quando se

trabalha com essas linguagens é significante (ODERSKY; SPOON; VENNERS, 2008, p.52-55).

A segurança oferecida ao usuário que trabalha com linguagens estáticas quando se trata de refatoração de código é muito grande. Considerando uma alteração onde é adicionado um parâmetro a determinado método, ao se fazer isso, recompila-se o programa e simplesmente corrigem-se as linhas que apresentarem erros. O mesmo vale quando se deseja alterar nome de métodos ou transferi-los de uma classe para outra.

2.5 RAÍZES DA LINGUAGEM

O *design* da linguagem Scala foi influenciado por diversas linguagens de programação. A linguagem apresenta de fato apenas alguns aspectos que são realmente inéditos, porque os demais de certa forma já foram aplicados em outras linguagens. O diferencial de Scala é como todas essas ideias foram postas em uma única linguagem (ODERSKY; SPOON; VENNERS, 2008, p.55-57).

Em uma camada mais superficial, Scala adota grande parte da sintaxe de Java e de C#. Modelos de expressões, declarações, blocos, sintaxes de classes, pacotes, tipos básicos, bibliotecas e modelo de execução são como os de Java.

O modelo orientado a objetos foi inspirado no modelo adotado por Smaltalk. A ideia de aninhamento universal também pode ser encontrado nas linguagens Algol e Simula. O modelo de programação funcional é semelhante ao encontrado nas linguagens SML, Ocam1 e F#. Os parâmetros implícitos de Scala foram baseados nos tipos de classes de Herkell, enquanto a implementação dos atores foi fortemente baseada no modelo encontrado na linguagem Earlang (ODERSKY; SPOON; VENNERS, 2008, p.55-57).

Neste capitulo foram apresentados alguns dos aspectos principais da linguagem Scala como sua interoperabilidade total com Java, seu código que pode ser escrito de forma muito concisa e seu método de tipagem estática, características essas que a distingue das demais outras encontradas atualmente.

3. PARADIGMAS DE PROGRAMAÇÃO

Como a linguagem Scala apresenta dois diferentes paradigmas de programação, o orientado objetos e o funcional, torna-se necessária a apresentação destes dois modelos ao leitor.

3.1 PARADIGMA DE PROGRAMAÇÃO ORIENTADA A OBJETOS

Paradigma de programação que hoje é imensamente utilizado começou a ser introduzido nos anos 60 na linguagem Simula, e nos anos 70 com Smaltalk.Hoje está presente na maioria das linguagens de programação existentes como C++, C#, VB.NET, Java, Object Pascal, Objective-C, Python, SuperCollider, Ruby e Smalltalk. Linguagens como ActionScript, ColdFusion, Javascript, PHP (a partir da versão 4.0), Perl (a partir da versão 5) e Visual Basic (a partir da versão 4) são linguagens que oferecem suporte a orientação a objetos(ODERSKY; SPOON; VENNERS, 2008, p.45-46).

Segundo Odersky, embora o termo orientação a objetos não possua uma definição precisa, entende-se por ele como uma forma de programação com abstrações de objetos do mundo real em forma de containers, nos quais são armazenados dados e funções respectivas para aquele objeto. Quando falamos em orientação a objetos, o entendimento de alguns conceitos se torna necessário:

• Classe: Uma Classe é um conjunto de objetos com características afins. A classe é responsável por definir as ações dos objetos por meio de seus métodos e quais características eles podem possuir através de seus atributos.

• Objeto: Um objeto é uma instância da classe, o objeto possui características próprias de acordo com a classe a qual pertence.

• Atributos: Atributos são as características apresentadas pelos objetos de uma classe. Basicamente, atributos são as estruturas de dados responsáveis por representar a classe.

• Métodos: Métodos são as ações realizadas pelos objetos que são definidas pela classe à qual pertencem.

A motivação para a criação da orientação a objetos veio da necessidade de se armazenar dados em algum tipo de estrutura, e a melhor forma de se fazer isso era armazenar dados e operações em um único container. O criador da linguagem Smaltalk, Alan Kay, diz que a construção de um simples objeto e de um grande computador seguem os mesmos princípios, a combinação de dados e operações sobre uma interface formalizada. Scala é puramente orientada a objetos, tanto que quando dizemos *1+2* estamos chamando o método + da classe *Int*, além disso, a linguagem permite que o usuário defina métodos com nomes de operadores para que depois esses métodos possam ser chamados através da notação de operador. Outro exemplo onde podemos ver este tipo de notação é no caso dos atores, quando desejamos passar uma mensagem, usa-se a denotação !, que nada mais é que um método da classe Ator(ODERSKY; SPOON; VENNERS, 2008, p.45-46).

Scala é mais avançada que algumas linguagens quando falamos em composição de objetos, e um exemplo disso são seus *traits. Traits* são parecidas com Interfaces em Java, no entanto, elas podem conter implementações de métodos e até mesmo campos. Objetos em Scala são construídos misturando-se membros da classe e membros de um determinado número de *traits*, assim, diferentes aspectos de classes podem ser encapsuladas em um *trait*. Este processo se assemelha a herança múltipla, no entanto, uma *trait* é muito mais “plugável” que uma classe, já que ela adiciona novas funcionalidades a uma superclasse (ODERSKY; SPOON; VENNERS, 2008, p.45-46).

3.2 PARADIGMA DE PROGRAMAÇÃO FUNCIONAL

Além de ser puramente orientada a objetos, Scala é uma linguagem completamente funcional. O conceito de programação funcional data antes mesmo da criação dos computadores, foi concebido nos anos 30 e é baseado nos cálculos Lambda de Alonzo Church. A primeira linguagem funcional é Lisp, que foi criada nos anos 50, mas como exemplo de outras linguagens pode-se citar Scheme, SML, Erlang, Haskell, OCaml e F# (ODERSKY; SPOON; VENNERS, 2008, p.46-48).

A programação funcional há muito tempo faz parte do ambiente acadêmico, mas vinha sendo deixada um pouco de lado pelas empresas, mas recentemente, o

interesse pelas técnicas e linguagens de programação funcionais tem aumentado. Algumas linguagens como C# e Java incorporam funcionalidades da programação funcional, como por exemplo, c*losures* (ODERSKY; SPOON; VENNERS, 2008, p.46-48).

Closure ou clausura ocorre normalmente quando uma função é declarada dentro do corpo de outra, e a função interior referência variáveis locais da função exterior. Em tempo de execução, quando a função exterior é executada, então uma *closure* é formada, que consiste do código da função interior e referências para quaisquer variáveis no escopo da função exterior que a *closure* necessita (ODERSKY; SPOON; VENNERS, 2008, p.46-48).

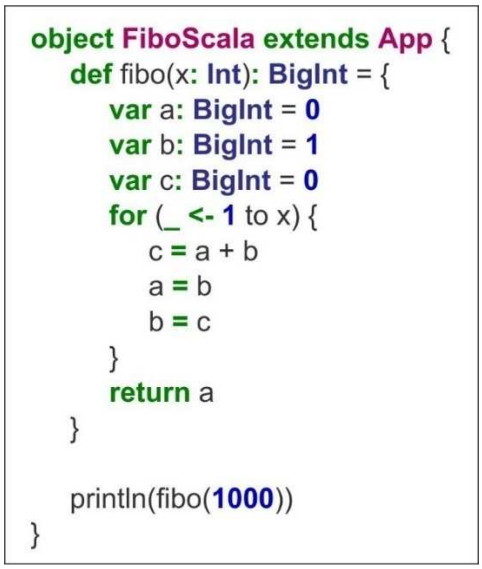
A programação funcional pode ajudar os desenvolvedores tornando-os mais produtivos por meio de conceitos como valores imutáveis, coleções, funções de alta ordem e casamento de padrões. No entanto, para aqueles que desconhecem o conceito de programação funcional, o entendimento do código gerado pode ser complicado. Como exemplo a seguir, dois programas para imprimir o milésimo elemento da sequência Fibonnaci, um escrito em Java e outro em Scala, porém, ambos escritos de forma iterativa.

```
import java.math.BigInteger;

public class FiboJava {
    private static BigInteger fibo(int x) {
        BigInteger a = BigInteger.ZERO;
        BigInteger b = BigInteger.ONE;
        BigInteger c = BigInteger.ZERO;
        for (int i = 0; i < x; i++) {
            c = a.add(b);
            a = b;
            b = c;
        }
        return a;
    }

    public static void main(String args[]) {
        System.out.println(fibo(1000));
    }
}
```

Figura 7 – Programa Java para impressão do milésimo elemento da sequência Fibonacci (Fonte: Artigo sobre Scala no site InfoQ[2])

[2]Disponível em: <http://www.infoq.com/br/articles/avaliando-scala> Acesso em fev. 2014.

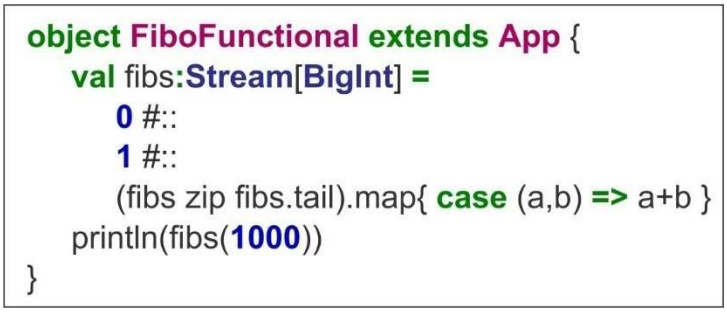
A imagem a seguir apresenta o programa escrito em Scala



```
object FiboScala extends App {
    def fibo(x: Int): BigInt = {
        var a: BigInt = 0
        var b: BigInt = 1
        var c: BigInt = 0
        for (_ <- 1 to x) {
            c = a + b
            a = b
            b = c
        }
        return a
    }

    println(fibo(1000))
}
```

Figura 8 – Programa Scala para impressão do milésimo elemento da sequência Fibonacci (Fonte: Artigo sobre Scala no site InfoQ[3])

Agora, vejamos o mesmo programa escrito de forma funcional, fazendo uso de sequências infinitas e tuplas.



```
object FiboFunctional extends App {
    val fibs:Stream[BigInt] =
        0 #::
        1 #::
        (fibs zip fibs.tail).map{ case (a,b) => a+b }
    println(fibs(1000))
}
```

Figura 9 – Programa Scala para impressão do milésimo elemento da sequência Fibonacci escrito de forma funcional (Fonte: Artigo sobre Scala no site InfoQ[4])

---

[3]Disponível em: <http://www.infoq.com/br/articles/avaliando-scala> Acesso em fev. 2014.

[4]Disponível em: <http://www.infoq.com/br/articles/avaliando-scala> Acesso em fev. 2014.

A programação funcional gira em torno de duas ideias principais.

A primeira é que funções são valores de primeira classe. Em uma linguagem funcional, o seu tipo é mesmo de uma *String* ou um *Integer*, por exemplo, sendo assim, é possível passar funções como argumento para outras funções, retornar os valores de função para outra função e ainda armazená-las em variáveis. A criação de funções de dentro de funções e criação de variáveis dentro de seu escopo também é possível, assim como a criação de uma função sem nome (ODERSKY; SPOON; VENNERS, 2008, p.46-48).

Funções com valores de primeira classe são muito boas para abstração e criação de estruturas de controle.

A segunda ideia é a que as operações de um programa devem mapear os valores de entrada para um valor de saída sem alterar os valores de entrada. Como exemplo, considere a implementação de *Strings* em Java e Ruby. Em Ruby, uma *String* é uma sequência de caracteres, que podem ser alterados dentro do mesmo objeto *String*. Em Java e Scala, uma *String* também é uma sequência de caracteres, a diferença entre as linguagens aparece quando se deseja alterar algum caractere dentro da sequência. Em Java, utilizando-se do método *replace* para fazer esta alteração, o resultando da operação será um objeto *String* diferente do objeto de entrada, por exemplo, uma *String* s, ao utilizarmos o método *s.replace('a', 'b'),* um novo objeto *String* é gerado, diferente de s. Sendo assim, as *Strings* em Java são imutáveis, enquanto em Ruby, são mutáveis. Métodos como o *replace* são denominados referencialmente transparentes, pois realiza determinada ação sem modificar o dado de entrada. Algumas linguagens funcionais encorajam o uso de métodos referencialmente transparentes e valores imutáveis, enquanto outras linguagens até mesmo exigem o uso desses conceitos. Em Scala, o usuário pode escrever seu código de forma imperativa quando se trabalha com valores mutáveis e funções que modificam dados de entrada. No entanto, a linguagem sempre oferece uma alternativa funcional melhor (ODERSKY; SPOON; VENNERS, 2008, p.46-48).

Ao final deste capitulo pode-se constatar que os dois paradigmas apresentados pela linguagem agregam diferentes qualidades a ela, qualidades essas que combinadas podem dar surgimento a uma nova forma de programação.

4. CARACTERÍSTICAS DA LINGUAGEM

Este capitulo vista apresentar o funcionamento de alguns pontos fundamentais para a linguagem Scala assim como para demais linguagens, como definições de classes, variáveis, expressões e funções.

4.1 VARIÁVEIS

Em Scala não há necessidade de se declarar o tipo de variável, isso torna o código muito mais limpo. Quanto a definição destas variáveis elas podem ser do tipo *val*ou *var.* Variáveis declaradas como *val* são constantes, ou seja, seus valores não poderão ser alterados, enquanto as variáveis do tipo *var* têm valores mutáveis. A linguagem aconselha o uso de variáveis do tipo *val* já que a maioria dos programas não precisam de variáveis mutáveis. As variáveis podem ser declaradas em uma única linha sem separação de ponto e vírgula quando possuírem o mesmo valor, por exemplo: (HORSTMANN, 2012, p.04-12).

```
val mensagem, comentario, descricao = null
var num, num1, num2, num3 = 0
```

Figura 10 – Declaração de Variáveis em Scala

Na imagem acima, são instanciadas três constantes que estão recebendo *null* e quatro variáveis que recebem o valor *0*.

4.1.1 LAZY VALUES

Variáveis declaradas como *lazy*(preguiçosas) só serão inicializadas quando acessadas pela primeira vez, por exemplo:

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

Figura 11 – Declaração de LazyValues em Scala

Essa chamada irá fazer a gravação de um arquivo de texto para uma String. No entanto, se o programa não chegar a acessar a variável words, o arquivo de texto nunca será aberto. *LazyValues* são úteis quando se deseja atrasar declarações que têm um grande custo de inicialização (HORSTMANN, 2012, p.04-12).

Variáveis declaradas como *lazy* podem ser vistas como um meio-termo entre uma *val* e uma *def.*

```
val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
//Arquivo é carregado assim que a variável é instanciada

lazy val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
//Arquivo é carregado assim que a variável é usada

def words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
//Arquivo é carregado toda vez que a variável é usada
```

Figura 12 – Diferença entre variáveis, funções e lazy values

4.2 SOBRECARGA DE OPERADORES ARITMÉTICOS

Operadores aritméticos trabalham em Scala da mesma forma que trabalham em outras linguagens, com a diferença de que em Scala, operadores aritméticos são métodos. Por exemplo, *a+b* é um atalho para *a.+(b)*. A linguagem não proíbe o uso de caracteres não alfanuméricos nos nomes de métodos, ela mesma faz uso de métodos com nomes deste tipo, por exemplo, a classe *BigInt* têm um método chamado */%*que retorna o quociente e o resto de uma divisão. Outra coisa interessante que torna o código muito mais limpo é o uso de atalhos para métodos,

por exemplo, ao invés de se escrever *1.to(10)* pode se usar o atalho *1 to 10* (HORSTMANN, 2012, p.04-12).

Os operadores aritméticos também podem ser usados com objetos da classe *BigInt* e *BigDecimal*. O que em Java seria *x.multiply(x). multiply(x)*, em Scala é simplesmente *x*x*x.*

Scala não faz uso dos operadores ++ e --, para fazer incremento e decremento de valores usa-se +=1 e -=1.

4.3 CHAMADA DE FUNÇÕES E MÉTODOS

Scala possui além de métodos, funções. Funções facilitam muito a vida do programador quando se deseja por exemplo trabalhar com funções matemáticas, como *min* ou *pow*. Diferentemente de métodos, para se trabalhar com funções não é necessário estar preso a um objeto. Scala não possui métodos estáticos, mas possui uma característica bem parecida, chamada *singleton objects.* Frequentemente, as classes Scala são acompanhadas por um *companion object*, cujos métodos se comportam de maneira parecida com os métodos estáticos. Por exemplo, o *companion object* da classe *BigInt* implementa um método chamado *probablePrime*, que gera um número primo randômico dado com um dado número de bits (HORSTMANN, 2012, p.04-12).

```
BigInt.probablePrime(100, scala.util.Random)
```

Figura 13 – Método ProbablePrime

Outro aspecto interessante da linguagem é o fato de que métodos que não possuem parâmetros e não modificam o objeto não precisam de parênteses. Um exemplo é o método da classe *StringOps* chamado *distinct*, que retorna os caracteres distintos de uma determinada String. Esse método não tem parâmetros e também não altera seu objeto, então sua chamada é feita sem o uso de parênteses:

```
"Não preciso de parenteses".distinct
```

Figura 14 – Método distinct

4.4 MÉTODO APPLY

Em Scala, qualquer objeto que tiver o método *apply* implementado pode ser chamado com o *.apply* omitido. Por exemplo, na classe *StringOps* está presente a seguinte definição:

```
def apply(n:Int): Char
```

Figura 15 – Definição do método apply

Essa função retorna o caractere de uma String presente em uma posição passada por parâmetro ao método. Ela pode ser chamada simplesmente assim:

```
"Scala"(4)
```

Figura 16 – Chamada método apply da classe String

Onde para a String*Scala* o quarto caractere e o valor retornado pelo método é a letra ‘l’. Outro exemplo pode ser visto no *companion object* da classe *BigInt*, onde o método *apply* é implementado para possibilitar a conversão de Strings ou números para *BigInt*, por exemplo:

```
BigInt("220393") //Atalho para o método BigInt.apply(“220393”).
```

Figura 17 – Chamada do método apply da classe BigInt

4.5 LOOPS

A linguagem também faz uso dos laços de repetições *while* e *do* da mesma forma que Java e C++. O loop *for* da linguagem Scala é um pouco diferente das outras linguagens, onde tem-se a inicialização, o teste e o incremento.

```
for(i = 0; i < 10; i++)
```

Figura 18 – Exemplo loop for em Java

Em Scala, o *for* é feito da seguinte maneira:

```
for(i<- 1 to 10)
```

Figura 19 – Exemplo loop for em Scala

Onde o método *to* faz parte da classe *RichInt*, método este que retorna a distância entre o número inicial, no caso 1 que é passado a variável i, e a faz assumir todos os números no intervalo até o número final que é 10.

Quando se trabalha com Strings ou Arrays, é interessante que a distância trabalhada pelo *for* seja de *0 até n-1.* Para estes casos, ao invés do método *to,* pode-se utilizar o método *until* (HORSTMANN, 2012, p.20-22).

```
val s = "Scala"
for(i<- 0 until s.length)
```

Figura 20 – Exemplo loop for com condição until

4.6 FUNÇÕES

Scala trabalha com métodos e funções. A diferença principal entre um método e uma função é que a primeira trabalha com um objeto, enquanto função não. A linguagem C++ também trabalha com funções, enquanto Java tenta imitá-las com métodos estáticos. Para a declaração de uma função são especificados seu nome, parâmetros e corpo, dessa maneira:

```
def testaPar(x: Double) = if (x % 2 == 0) "par" else "impar"
```

Figura 21 – Definição de funções em Scala

Quando se trabalha com funções, todos os parâmetros devem ter seus tipos especificados, e caso a função não seja recursiva, também não é necessário especificar o tipo de seu retorno, já que o compilador da linguagem determina como tipo do retorno o tipo da última expressão ao lado direito do símbolo “=”. Caso a função requeira mais de uma expressão, é recomendado o uso de blocos, assim, o retorno da função será o valor da última expressão do bloco (HORSTMANN, 2012, p.22).

```
def fac(n : Int) = {
   var r = 1
   for (i <- 1 to n) r = r * i
   r
}
```

Figura 22 – Exemplo do valor de função em Scala

No exemplo acima, o valor retornado pela função é *r* logo depois do loop *for*.

Quando se fala em funções recursivas, o tipo do retorno deve sim ser especificado para que o compilador saiba o tipo a retornar.

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

Figura 23 – Exemplo função recursiva em Scala

Diferente do que vemos em Java, em Scala é aconselhável não utilizar a palavra *return* em uma função nomeada, já que em funções anônimas, a palavra *return* funciona como um *break* para a função.

4.6.1 ARGUMENTOS

Em Scala, argumentos podem ser nomeados e até mesmo pode-se passar vários deles a uma função que não especifica a quantidade de valores que precisa receber.

4.6.1.1 Argumentos Padrão e Nomeados

A linguagem Scala permite a declaração de parâmetros com valores já pré-determinados, que serão usados quando valores para esses determinados parâmetros não forem passados, por exemplo:

```
def enfeitar(str: String, esquerda: String = "[", direita: String = "]") = {
    esquerda + str + direita
}
```

Figura 24 – Exemplo de parâmetros pré-determinados em Scala

A função tem três argumentos, nome, esquerda e direita, tendo dois deles valores padrões já determinados, esquerda = “[” e direita = ”] ”.

Se o método enfeitar(“Maria”) fosse chamado, o resultado seria [Maria], no entanto, os valores pré-determinados para os parâmetros podem ser substituídos, basta apenas passar valores a eles, chamando agora ao invés de *enfeitar(“Maria”)*, *enfeitar(“Maria”, “<<”, “>>”)* e o resultado será <<Maria>>. Os valores são atribuídos

aos parâmetros de acordo com a ordem de definição da função. Caso a função enfeitar fosse chamada da seguinte forma, *enfeitar(“Maria”, “<<”)*, com a passagem de apenas dois valores, o resultado retornado seria <<Maria], ou seja, seria atribuído o valor padrão para o último parâmetro, já que para ele não foram passados valores específicos.

Parâmetros em Scala também podem ser nomeados, evitando assim à necessidade de se saber a ordem em que estes foram definidos na função.

```
def enfeitar(esquerda:String = "<<<", str:String = "Hello", direita:String = ">>>") = {
  esquerda + str + direita
}

//Chamada do método
enfeitar(direita = ">>", esquerda = "<<", str = "Hello")
```

Figura 25 – Exemplo de parâmetros nomeados em Scala

A ordem segue conforme a declaração da função, ou seja, chamada essa função, o resultado seria *<<Hello>>.*

4.6.1.2 Argumentos Variáveis

A criação de funções que podem receber um número *n* de argumentos é uma das possibilidades que a linguagem Scala oferece, por exemplo:

```
    def ImprimirNomes(Nomes: String*){
      for(i<-0 until Nomes.length){
        println(i, Nomes(i))
      }
    }
}
```

Figura 26 – Exemplo de definição função que pode receber um número indeterminado de argumentos

No exemplo acima, o operador * depois da declaração do tipo do argumento indica que a função pode ser chamada com a passagem de vários argumentos. O que a função realmente recebe é um único parâmetro do tipo *Seq,* que pode ter suas posições visitadas através de um loop *for.*

```
ImprimirNomes("Daniel", "Pedro", "Juliana", "Larissa")
```

Figura 27 – Exemplo de chamada de função com um número indeterminado de argumentos

4.7 PROCEDURES

Scala tem uma notação especial para funções que não retornam valores. Essas funções são chamadas de *procedures*. Procedures tem como característica não apresentar o sinal “=” depois dos parênteses em que fica o corpo da função. Não é correto dizer quer procedures não retornam valores, as funções deste tipo têm como retorno um tipo chamado *Unit*.

Procedures podem ser declaradas de duas maneiras. Na primeira, declara-se a função como se fosse uma outra qualquer, mas omite-se o sinal “=”.

```
def ImprimirNome(nome: String){//Omite-se o sinal '='
    println("Nome: " + nome)
}
```

Figura 28 – Declaração de uma procedure com omissão de retorno em Scala

Na segunda maneira, declara-se a função especificando o tipo de retorno, e adicionando o sinal “=”

```
def ImprimirNome(nome: String):Unit = {//Especifica-se o tipo de retorno.
println("Nome: " + nome)
}
```

Figura 29 – Declaração de uma procedure sem omissão de retorno em Scala

4.8 EXCEÇÕES

As exceções da linguagem Scala funcionam da mesma maneira que em Java e C++. Quando uma exceção é lançada, a execução é abortada e o sistema em tempo de execução procura um *exception handler* que possa aceitar o tipo de exceção lançada. Se o *handler* for encontrado, o programa é resumido, do contrário ele termina sua execução. Assim como em Java, os objetos que lançam erro devem pertencer a subclasse *java.long.Throwable.* No entanto, em Scala, não é preciso indicar que uma função poderá lançar exceções.

Uma expressão do tipo *throw* tem um tipo especial *Nothing*, que pode ser muito usado em expressões de tipo *if/else.* Se um dos braços da função for do tipo *Nothing*, o tipo da expressão será o mesmo tipo do outro braço (HORSTMANN, 2012, p.26-29).

```
if (x >= 0) {
  sqrt(x)
} else {
  throw new IllegalArgumentException("x não deve ser negativo")
}
```

Figura 30 – Exemplo de expressão do tipo throw em Scala

Vale ressaltar que em Scala tudo tem valor, uma expressão condicional de tipo *if/else* também tem um valor, que depende de sua condição e de seus valores, por exemplo:

```
if(num % 2 == 0) "par" else "impar"
```

Figura 31 – Exemplo de valores de função do tipo if/else

Caso a condição seja aceita pela expressão i*f/else,* seu valor será “par”, do contrário, “impar”. Isso torna muito mais fácil trabalhar com expressões deste tipo, pois possibilita algo como:

```
val resposta = if(num % 2 == 0) "par" else "impar"
```

Figura 32 – Variável recebendo valor de função tipo if/else

A sintaxe das expressões *try/catch* é semelhante a Java, e assim as exceções mais específicas devem vir antes das mais gerais.

```
try {
    process(new URL("http://horstmann.com/fred-tiny.gif"))
} catch {
  case _: MalformedURLException => println("Bad URL: " + url)
  case ex: IOException => ex.printStackTrace()
}
```

Figura 33 – Definição blocos try/catch (In: Horstmann, 2012, p.27)

O bloco da expressão *finally* também é semelhante as outras linguagens.

```
var in = new URL("http://horstmann.com/fred.gif").openStream()
try {
    process(in)
} finally {
    in.close()
}
```

Figura 34 – Definição bloco finally (In: Horstmann, 2012, p.30)

4.9 CLASSES

Definições de classes em Scala são muito semelhantes à Java e C++. Alguns pontos que podem ser destacados em relação a criação de classes em Scala são:

• Campos criados em classes Scala já contém *getters* e *setters*.

• Toda classe tem um construtor primário entrelaçado com sua definição. Seus parâmetros são convertidos para os campos da classe e a execução de todas as suas declarações são realizadas no corpo da classe.

• Assim como em Java, Scala permite a criação de construtores customizados, que são chamados de *this.*

Em Scala, uma classe não é declarada como pública em sua criação. Um arquivo de código-fonte em Scala pode conter várias classes, e todas elas têm visibilidade pública (HORSTMANN, 2012, p.50-64).

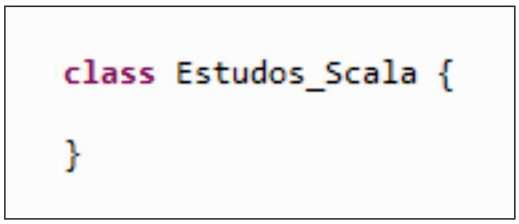


Figura 35 – Definição de classe em Scala

Diferentemente de Java, onde criamos classes que serão executadas e dentro destas instanciamos objetos, em Scala, o que é executado pela JVM (Java Virtual Machine) são os *Objects*(Objetos). Assim como indicamos em Java qual será a classe a ser executada através do método principal “main(String[] args)”, em Scala também indicamos qual Object a ser executado através do método “main(args: Array[String])”, ou simplesmente falamos que o Object estende a *traitApp* (HORSTMANN, 2012, p.50-64).

`public class HelloWorld {`
`  public static void main(String[] args)`
`  System.out.println("HelloWord");`
`  }`
`}`
	`object HelloWorld {`
	`  def main(args: Array[String]){`
	`  println("Hello World")`
	`  }`
	`}`
		`object HelloWorld extends App{`
		`  println("HelloWorld")`
		`}`

JAVA	SCALA	SCALA EXTENDENDO APP

Figura 36 – Diferenciação métodos main em Scala e Java

Os *getters* e *setters* trabalham da mesma maneira em Java e Scala. Como já dito acima, esses métodos são criados automaticamente em Scala quando criamos os atributos de uma classe. Lembrando que a todos os atributos criados em Scala devem ser atribuídos um valor inicial (HORSTMANN, 2012, p.50-64).

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

Figura 37 – Criação de atributos em Scala

Para o exemplo acima, os *getters* e *setters* do atributo nome são respectivamente chamados de nome e nome_=. Por exemplo:

```
val pessoa = new Pessoa
println(pessoa.idade) //É chamado o método pessoa.idade()
pessoa.idade = 21 //É chamado o método pessoa.idade_=(21)
```

Figura 38 – Métodos getters e setters em Scala

4.9.1 OBJETOS PRIVADOS

Em Scala, assim como outras linguagens orientadas a objetos, um método pode acessar todos os atributos privados dos objetos pertencentes a sua mesma classe.

```
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 39 – Definição variável privada em Scala (In: Horstmann, 2012, p.56)

Scala oferece um acesso ainda mais restrito a campos privados com o qualificativo *[this].*

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

Figura 40 – Definição variável privada com qualificativo [this]

(In: Horstmann, 2012, p.56)

Com o uso deste qualificativo, os métodos da classe *Counter* só poderão acessar o valor do campo *value* do objeto atual.

4.9.2 CONSTRUTORES

A linguagem possibilita a criação de classes sem um construtor.Quando isso acontece, um construtor primário sem parâmetros é atribuído a classe. O número de construtores que a linguagem possibilita é indefinido. Os construtores são divididos em dois tipos, primários e auxiliares (HORSTMANN, 2012, p.50-64).

4.9.2.1 Construtor Primário

Em Scala, toda classe tem um construtor primário que é entrelaçado com a definição da classe. Os parâmetros deste construtor devem ser colocados imediatamente após o nome da classe.

```
class Pessoa(val nome:String, val idade:Int) {}
```

Figura 41 – Construtor primário em classe Scala

Os parâmetros do construtor primário se tornam os atributos da classe.

Todas as declarações do construtor primário são executadas na definição da classe.

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

Figura 42 – Exemplo de métodos ligados ao construtor primário

A declaração *println* faz parte do construtor primário e é executado sempre que um novo objeto da classe é instanciado.

Isso tem muita utilidade quando se precisa configurar um campo no momento de sua criação.

```
class MeuPrograma {
  private val prop = new Properties
  prop.load(new FileReader("meuprograma.properties"))
  //Declaração acima faz parte da declaração do construtor
  ...
}
```

Figura 43 – Configuração de campos com construtor primário

Quando não são inseridos parâmetros após o nome da classe, então esta tem um construtor primário sem parâmetros.

4.9.2.2 Construtores Auxiliares

Assim como Java, Scala possibilita a criação de diversos construtores. No entanto, quando um construtor é mais importante que os demais, ele é chamado de construtor primário. Como já mencionado, é possível a criação de n números de construtores auxiliares, que são chamadas em Scala de *this.* Estes construtores auxiliares devem sempre fazer chamada a um construtor auxiliar previamente declarado ou ao construtor primário (HORSTMANN, 2012, p.50-64).

```
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 44 – Construtores auxiliares em Scala

5. IMPLEMENTAÇÃO

Neste capitulo, será mostrado como foi realizada a implementação de um pequeno cadastro utilizando Scala em conjunto com algumas ferramentas desenvolvidas para se trabalhar com Java.

O objetivo da implementação é realizar uma operação simples no banco de dados PostgreSQL utilizando o framework hibernate para fazer toda a parte de conversação entre o sistema feito em Scala e o banco de dados.

5.1 FERRAMENTAS UTILIZADAS

Para o desenvolvimento da implementação foi utilizado a IDE (*Integrated Development Environment*) Eclipse com o plug-in de desenvolvimento Scala instalado. Para demonstrar a fácil integração da linguagem com Java, a interface foi construída utilizando Swing através de um plug-in do eclipse chamado WindowBuilder.

5.1.1 Eclipse

O Eclipse é uma ferramenta IDE gratuita com foco para desenvolvimento Java, mas que também compreende várias outras linguagens como Android e Scala, além de oferece suporte para instalação de plug-ins que facilitam o desenvolvimento de aplicações que necessitem de diferentes ferramentas.

Figura 45 – Logotipo IDE Eclipse

5.1.2 PostgreSQL

O PostgreSQL é um sistema gerenciador de banco de dados objeto relacional open-source que teve início na década de 80 com o projeto ingres. Ele é uns dos SGBD(Sistema Gerenciador de Banco de dados) open-source mais avançado e conta com recursos como:

- Consultas complexas
- Integridade transacional
- Suporte ao modelo hibrido objeto-relacional
- Implementação de gatilhos e visões.

Figura 46 – Logotipo PostgreSQL

5.1.3 WindowBuilder

O WindowBuilder é um plug-in para Eclipse que possibilita a criação de interfaces Java em SWT ou Swing de forma simples, sem que o programador precise perder tempo escrevendo código. Com ele é possível a criação de JFrames, JDialogs e diversos outros tipos de componentes através de controles *drag-and-drop.*

5.1.4 Hibernate

Hibernate é um framework para mapeamento objeto-relacional escrito em Java. Ele facilita o mapeamento de atributos entre uma base tradicional de dados relacionais e o modelo onjeto de uma plicação mediante o uso de arquivos XML (*eXtensible Markup Language*) ou anotações Java.

Figura 47 – Logotipo Hibernate

5.2 MODEL OBJECT

O termo *model object* ou objeto modelo é um termo informal sem definição amplamente aceita, eles se referem a classes que encapsulam item próximos e relacionados. Para um entendimento mais fácil, pose-se dizer que objetos modelo são representações de objetos do mundo real transformados em uma entidade computacional.

A imagem a seguir mostra a classe modelo chamada CandidatoBEAN escrita em Scala que foi utilizada na implementação do trabalho, onde encontram-se a declaração dos atributos do candidato e a criação de um construtor auxiliar. *Veja capítulo 4.9.2.2.*

```
class CandidatoBEAN {
  @BeanProperty
  var id: Integer = 0
  @BeanProperty
  var nome: String = ""
  @BeanProperty
  var nascimento: String = ""
  @BeanProperty
  var rg: String = ""
  @BeanProperty
  var cpf: String = ""
  @BeanProperty
  var pretencao: Float = 0
  @BeanProperty
  var cidade: String = ""
  @BeanProperty
  var area: String = ""

  def this(nome: String, nascimento: String,
      rg:String, cpf: String, pretencao: Float,
      cidade: String, area: String) {
    this()
    this.nome = nome
    this.nascimento = nascimento
    this.rg = rg
    this.cpf = cpf
    this.pretencao = pretencao
    this.cidade = cidade
    this.area = area
  }
}
```

Figura 48 – Classe modelo CandidatoBEAN

5.3 DATA ACCESS OBJECT

Objeto de acesso a dados ou simplesmente DAO (*Data Access Object*) é um padrão de persistência de dados que permite separar as regras de negócio das regras de acesso ao banco de dados. Para uma arquitetura que utilize a arquitetura MVC

(*Model-View-Controller*) todas as ações que envolvem a interação entre a aplicação e o banco de dados devem ser feitas por classes DAO.

Para o projeto do cadastro em Scala, o hibernate foi responsável por realizar a comunicação entre a aplicação e o banco de dados PostgreSQL. A imagem a seguir mostra o código usado para configurar o hibernate dentro da aplicação a partir de uma classe Scala.

```
class CandidatoDAO {

  var cfg = new Configuration();
  cfg.setProperty("hibernate.connection.driver_class", "org.postgresql.Driver");
  cfg.setProperty("hibernate.connection.username", "postgres");
  cfg.setProperty("hibernate.connection.password", "postgres");
  cfg.setProperty("hibernate.connection.url", "jdbc:postgresql:bcc");
  cfg.setProperty("hibernate.dialect", "org.hibernate.dialect.PostgreSQLDialect");
  cfg.setProperty("hibernate.show_sql", "true");
  cfg.addResource("CandidatoBEAN.hbm.xml")
```

Figura 49 – Configuração Hibernate

Na última linha do código descrito acima, o método *addResource* faz referência ao arquivo XML de mapeamento da classe candidatoBEAN.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/
Hibernate Mapping DTD 3.0//EN" "http://www.hibernate.org
/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="estudos">
    <class name="bcc_bean.CandidatoBEAN" table="CANDIDATO">
        <id name="id" column="ID" type="integer">
            <generator class="assigned"/>
        </id>
        <property name="nome" column="NOME" type="string" />
        <property name="nascimento" column="NASCIMENTO" type="string" />
        <property name="rg" column="RG" type="string" />
        <property name="cpf" column="CPF" type="string" />
        <property name="pretencao" column="PRETENCAO" type="float" />
        <property name="cidade" column="CIDADE" type="string" />
        <property name="area" column="AREA" type="string" />
    </class>
</hibernate-mapping>
```

Figura 50 – Mapeamento Classe CandidatoBEAN

A seguir, a imagem apresenta o método responsável por gravar objetos do tipo candidato no banco de dados. Método este que também está presente na classe CandidatoDAO.

```
def GravarCandidato(candidato: CandidatoBEAN) {
  try {
    val sf = cfg.buildSessionFactory()
    val session = sf.openSession()
    val ts = session.beginTransaction();
    session.save(candidato)
    ts.commit()
  } catch {
    case e: Exception => e.printStackTrace()
  }
}
```

Figura 51 – Método GravarCandidato

Primeiramente, é criada uma *Session Factory* e a ela é atribuída o valor do objeto de configuração do hibernate criado mais acima. Depois disto, uma *Session* é criada seguidamente de uma *Transaction.* O método *save* é chamado e a ele é passado como parâmetro o objeto candidato que é recebido como argumento pelo método GravarCandidato.

```
def SelecionarCAndidatos(): java.util.List[CandidatoBEAN] = {
  val sf = cfg.buildSessionFactory()
  val session = sf.openSession()
  val query = session.createQuery("from CandidatoBEAN")
  val candidatos: java.util.List[CandidatoBEAN] =
      query.list.asInstanceOf[java.util.List[CandidatoBEAN]].toList
  return candidatos
}
```

Figura 52 – Método SelecionarCandidatos

A imagem acima mostra a implementação do método SelecionarCandidatos, que é responsável por fazer a consulta no banco de dados e retornar uma Lista de objetos do tipo candidato através de uma *Session.*

```
def Deletar(candidato: CandidatoBEAN) {
  val sf = cfg.buildSessionFactory()
  val session = sf.openSession()
  val ts = session.beginTransaction()
  session.delete(candidato)
  ts.commit()
}
```

Figura 53 – Método Deletar

A imagem acima mostra a implementação do método Deletar, que fica responsável por realizar um *Delete* no banco de dados tendo como referência para a ação o argumento recebido pelo método.

5.4 INTERAÇÃO ENTRE SCALA E JAVA

Até o presente momento da implementação, todos os códigos foram escritos em classes Scala. A interface da aplicação, como já mencionada, foi construída com Swing, portanto, está escrita em uma classe Java.

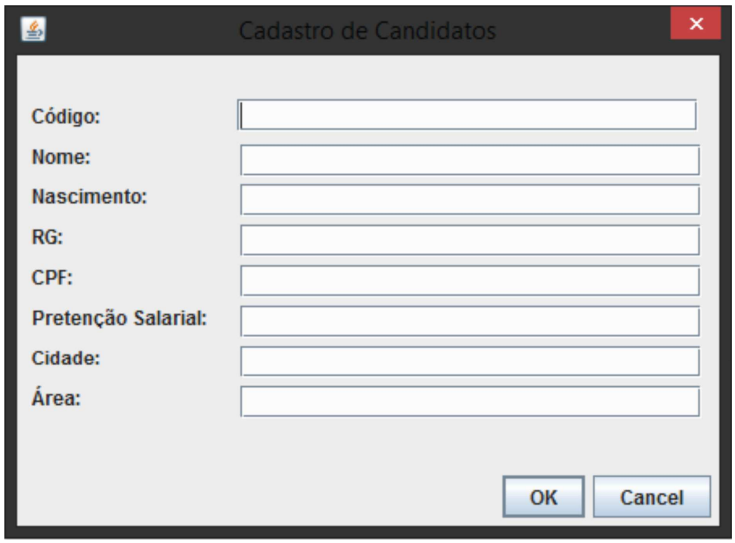


Figura 54 – Interface da Aplicação

A chamada aos métodos escritos em Scala dentro da classe escrita em Java é feita da mesma maneira como se faria trabalhando exclusivamente com Java.

```
public void CarregarCandidato() {

    // Criação do Objeto Scala dentro do código Java
    CandidatoBEAN bCandidato = new CandidatoBEAN();

    // Preenchendo objeto Scala com campos de texto
    bCandidato.setId(Integer.parseInt(txtId.getText()));
    bCandidato.setNome(txtNome.getText());
    bCandidato.setNascimento(txtNascimento.getText());
    bCandidato.setRg(txtRg.getText());
    bCandidato.setCpf(txtCpf.getText());
    bCandidato.setPretencao(Float.parseFloat(txtPretSal.getText()));
    bCandidato.setCidade(txtCidade.getText());
    bCandidato.setArea(txtArea.getText());

    // Instanciando objeto Scala e chamando o método de inserção
    CandidatoDAO dCandidato = new CandidatoDAO();

    try {
        dCandidato.GravarCandidato(bCandidato);
        JOptionPane.showMessageDialog(null,
                "Candidato cadastrado com sucesso");
    } catch (Exception e) {
        JOptionPane.showMessageDialog(null, "Erro ao cadastrar Candidato");
        e.printStackTrace();
    }
}
```

Figura 55 – Criação dos Objetos Scala

Primeiramente, um objeto da classe CandidatoBEAN que foi escrito em Scala é instanciando e a ele são passados os valores captados pelos campos de texto presentes na interface da aplicação. Depois do preenchimento do objeto bCandidato, um objeto da classe CandidatoDAO também é instanciado e o método GravarCandidato é invocado recebendo como argumento o objeto bCandidato previamente criado e carregado.

A integração entre as linguagens Scala e Java é feita de forma harmoniosa, não é preciso um intermediador entre as duas, que por terem seu código transformado em bytecodes possuem uma arquitetura muito semelhante quando interpretada pela JVM.

6. CONCLUSÃO

A linguagem Scala pode trabalhar satisfatoriamente com as linguagens Java e C#. Sua interoperabilidade é um grande fator de sucesso para o crescimento da linguagem.

A liberdade que a linguagem oferece ao programador de poder trabalhar de forma imperativa ou funcional não o limita a um único paradigma, a fusão dos dois conceitos de programação visando obter o melhor que cada paradigma pode oferecer fortalece amplamente o código gerado no final.

A arquitetura da linguagem é muito semelhante a Java e C#. Sua tipagem estática garante ganho de tempo por parte dos programadores, porém, torna o código um pouco mais complicado de ser entendido por pessoas que desconheçam as características da linguagem.

A implementação descrita no trabalho mostrou a fácil integração entre a linguagem Java e Scala. A configuração do framework hibernate não apresentou problemas e foi feita da mesma forma que seria em uma aplicação Java.

A linguagem vem ganhando espaço no mercado, programadores que já possuem uma boa base em linguagens como Java e C# tendem a ter mais facilidade em dominar a linguagem, ressaltando que a arquitetura entre essas linguagens é muito semelhante.

REFERENCIAS BIBLIOGRAFICAS

ECKEL, Bruce. **Scala: The Static Language that Feels Dynamic**. Disponível em <http://www.artima.com/weblogs/viewpost.jsp?thread=328540>. Acesso em: 04 outubro 2012.

HORSTMANN, Cay S. **Scala for the Impatient**. 2012. Addison-Wesley Professional; 1 edition – 384 p.

ODERSKY, Martin. **Scala by Example**.École Polytechinique Fédérale de Lausanne – EPFL – Suíça, 2013 – 137 p.

ODERSKY, Martin. **The Scala Language Especification**. École Polytechinique Fédérale de Lausanne – EPFL – Suíça, 2014 – 183 p.

ODERSKY, Martin; SPOON, Lex; VENNERS, Bill. **Programming in Scala.**2008.Artima Press: Califórnia, 2008 – 754 p.

SALGADO, Filipe Ferraz. **Uma Implementação do AIRS em Scala,** Universidade de São Paulo – USP – São Paulo – Brasil – Dissertação apresentada ao Instituto de matemática e Estatística da Universidade de São Paulo para obtenção do título de Mestre em Ciências– Março de 2012.

SATO, Larisa Matsumoto. **Um Sistema de Programação e Processamento para Sistemas Multi Processadores,** Escola Politécnica da universidade de São Paulo – São Paulo – Brasil, 1992.

SCHEPKE, Claudio. **Ambientes de Programação Paralela**, Universidade Federal do Rio Grande do Sul – UFRGS – Porto Alegre, RS – Brasil – Programa de Pós Graduação em Computação, 2009.