RELATÓRIO LINGUAGENS DE PROGRAMAÇÃO

LINGUAGEM GROOVY

Professor:

Andrei Rimsa

Alunos:

Arthur Severo De Souza

Victor Le Roy Matos

1. Introdução

A Groovy é uma linguagem de programação (LP) criada por James Strachan que teve sua primeira versão completa lançada para o público em 2007, é orientada a objetos e tem como característica sua extrema compatibilidade com Java. Hoje a LP se encontra na fase alfa do desenvolvimento de sua versão 4.0 e tem suporte contínuo para o modelo 3.0 anterior, que é adotado por nomes como o LinkedIn, BestBuy, Netflix, Google e Gradle. Alem disso, a Groovy pode ser utilizada em desenvolvimento web, utilizando de, por exemplo, Grails, complemento em linguagens embarcadas por meio da construção de plataformas estendidas em tempo de execução pelos usuários com a inclusão, a edição e a remoção de scripts, e criação de ferramentas de automatização.

2. Motivação

Desde o início de seu desenvolvimento, em 2003, a linguagem foi idealizada como uma alternativa a seu parente mais popular (Java) e neste mesmo ano ganhou o Java Community Process (JCP), que é um processo que permite ao ganhador participação no desenvolvimento de versões futuras do Java. Em 2007 ganhou o prêmio de inovação JAX e em 2008 um de seus frameworks web, Grails, ficou em segundo na mesma premiação. Foi comprada pela SpringSource, que por sua vez foi comprada pela VMWare em 2009, responsável pelo lançamento da versão 2.0. Em 2015 teve seu patrocínio dissolvido pela empresa e passou a integrar o projeto de incubação da Apache Software Foundation até ser promovido à um projeto top-level no final do mesmo ano.

É uma linguagem baseada em Java e possui características em comum com as linguagens Python, Ruby e SmallTalk. Após ser compilada, gera bytecode Java próprio para ser interpretado em uma JVM (Java Virtual Machine) e tem sintaxe facilmente integrável em projetos desenvolvidos na linguagem. Além disso, a Groovy é totalmente integrada ao Java no mais baixo nível. Por exemplo: se você instanciar um objeto do tipo Date, esse nada mais é do que uma instância de java.util.Date. E tudo funciona de maneira transparente visto que, por debaixo dos panos, tudo é bytecode Java.

2.1. O que é Groovy, afinal?

" Apache Groovy is a powerful, optionally typed and dynamic language, with static-typing and static compilation capabilities, for the Java platform aimed at improving developer productivity thanks to a concise, familiar and easy to learn syntax. It integrates smoothly with any Java program, and immediately delivers to your application powerful features, including scripting capabilities, Domain-Specific Language authoring, runtime and compile-time meta-programming and functional programming." **[referência 1]**

Portanto, Groovy possui uma sintaxe mais simples e enxuta do que o Java e possui recursos poderosos que não são encontrados na linguagem Java, como closures. Além disso, possui o Grails, um framework de produtividade baseado em Spring e Hibernate que permite o desenvolvimento de aplicações Java EE com a mesma agilidade que Ruby on Rails;

```
class Main {
    static void main (String[] args) {
        println ('Hello World');
    }
}
```

Figura 1 - Hello World

3. Características da linguagem

- Orientada a objetos;
- Mutualmente estática e dinâmica, além de fortemente tipada;
- Closures e Traits;
- Sintaxe nativa para listas, arrays associativos, vetores, e expressões regulares;
- Escopo dinâmico;
- Sobrecarga de operadores;
- Possui características em comum com Java, já que é baseada nela;
- É compacta:
    - O aprendizado é simplificado devido a menor quantidade de dependências que, inevitavelmente, surgem durante o desenvolvimento Java. Portanto, alguém que queira aprender a programação em Groovy pode começar com o básico do Java e seguir para níveis mais avançados com menos dificuldade.
- GDK:
    - Assim como o Java possui a JDK, o Groovy possui a GDK. É fácil confundir-se quando pensamos nisso pois todo objeto Java pode ser usado dentro do Groovy, mas o contrário não é verdade. A GDK estende a JDK adicionando métodos que não existem originalmente nos objetos Java.

3.1. Particularidades

- Utilizamos o comando “groovy [source file]” para executar o programa.
- O comando “groovyc” serve para compilar o programa e criar os arquivos compilados no formato .class.

- Temos um console de script, basta utilizar o comando “groovysh”.
- Como temos um console de script, consequentemente conseguimos criar scripts com a LP.
  - O símbolo **“;”** é opcional, a não ser que coloque mais de um comando na mesma linha.
  - Para definir uma variável, utilizamos a palavra reservada ***def***, que possibilita a alteração do tipo da variável em tempo de execução.
  - Podemos usar DataTypes que proíbe a alteração do tipo da variável durante a execução, isto é, declarar variáveis com tipos definidos, como ***int*** ou ***double***.
  - Possui precedência de operadores.

```
groovy:000> 2 + 2 * 5 / 2
===> 7
```

Figura 2 - Sobrecarga/Precedência de operadores

3.2. Tipagem estática e dinâmica

Um ponto curioso da Groovy é a possibilidade de desenvolver código dinamicamente enquanto se baseia na escrita estática, técnica que não é empregada em Java. A dinamicidade da linguagem possibilita que dados usualmente tratados de forma estática (em tempo de compilação) sejam resolvidos durante a execução do programa permitindo, por exemplo, o tratamento de invocações de métodos inexistentes.

```groovy
class Main {
    static void main (String[] args) {
        def a = 2;
        a = 'Oi';

        int b = 3;

        String c = "Ola";

        def x = 0..5;
        println('x = ' + x);
        println('x[2] = ' + x.get(2));

        def heterogeneous = [1, "a", true];
    }
}
```

Figura 3 - Demonstração tipagem estática e dinâmica

3.3. Classes

São similares a classes Java e são compatíveis com Java no nível da JVM. Existem alguns tipos de classe, sendo eles: Normal, Aninhada, Abstrata, Interface e Traits. Por padrão, todas as classes ou métodos são públicas, caso não tenha a anotação de visibilidade. Além disso, métodos ***Getters and Setters*** são criados automaticamente e dinamicamente.

3.3.1. Classes Normais

São classes definidas como top-level e são concretas. Isso significa que podem ser instanciadas por outras classes ou scripts. Dessa forma, elas só podem ser classes públicas (que são definidas por padrão) e podem ser instanciadas ao serem chamadas a partir de seu construtor.

3.3.2. Classes Aninhadas

São classes definidas dentro de outras classes, normalmente, a classe mais externa acessa a classe mais interna, porem as classes internas podem acessar os dados das classes externas, mesmo se forem privados e outras classes com exceção a externa, não podem acessar a interna.

Alguns motivos para se usar as classes aninhadas são: aumenta o encapsulamento, já que a classe interna está escondida das outras que não precisam utiliza-la, melhoram a organizam do código, pois está agrupado dentro da única classe que utiliza a classe interna e promovem códigos mais fáceis de realizar a manutenção.

```
class Outer {
      private String privateStr

      def callInnerMethod() { new Inner().methodA() }

      class Inner {
            def methodA() { println "${privateStr}." }
      }
}
```

Figura 4 - Exemplo de classe aninhada

3.3.3. Classes Abstratas

Representam conceitos genéricos de classe, não podem ser instanciadas, e, por isso, são criadas com intuito de serem implementadas pelas subclasses. Incluem campos, propriedades e métodos que podem ser abstratos ou concretos.

```
abstract class Abstract {
      String name

      abstract def abstractMethod()

      def concreteMethod() {
            println ('concrete')
      }
}
```

Figura 5 - Exemplo de classe abstrata

3.3.4. Interfaces

Uma interface define um contrato que as classes que a implementam precisam seguir. A interface define apenas uma lista de métodos (sua assinatura) que as classes devem implementar, sendo que estes métodos devem ser sempre públicos. As classes que forem implementar a interface, deve ter explicito esta implementação e para isso podemos usar o operador de coerção ***as***.

```
interface Greeter { void greet(String name) }
```

```
class SystemGreeter implements Greeter {
     void greet(String name) { println "Hello $name" }
}
def greeter = new SystemGreeter()
assert greeter instanceof Greeter
```

Figura 6 - Exemplo de construção e declaração de interface

1. **SystemGreeter** declara **Greeter** interface e implementa o método **greet**
2. Qualquer instancia de **SystemGreeter** é também instancia da interface

```
greeter = new DefaultGreeter()
coerced = greeter as Greeter
assert coerced instanceof Greeter
```

*Figura 7 - Declaração de interface utilizando **as***

1. Cria uma instancia de **DefaultGreeter** que não implementa a interface
2. Coloca a instância dentro da interface **Greeter** durante a execução
3. A instancia se torna uma implementação da interface **Greeter**

3.3.5. Traits

Traits são estruturas que permitem criar uma composição de comportamentos, implementação de interfaces em tempo de execução, override de comportamentos e compatibilidade com tipos ***static*** durante a compilação. De forma geral, podem ser considerados interfaces que carregam implementações default e estado.

```
trait FlyingAbility {
      String fly() { "I'm flying!" }
}

class Bird implements FlyingAbility {}
def b = new Bird()
assert b.fly() == "I'm flying!"
```

Figura 8 - Exemplo básico de criação de trait

1. Declaração de um **Trait**
2. Declaração de um método dentro de um **Trait**
3. Adiciona o trait **FlyingAbility** dentro da classe **Bird**
4. Instancia um novo **Bird**
5. A classe **Bird** automaticamente recebe o comportamento de **FlyingAbility**

Traits podem implementar interfaces, propriedades, campos e métodos. Além disso, podemos fazer composição dos comportamentos fazendo uma classe implementar mais de um Trait.

```
trait FlyingAbility {
      String fly() { "I'm flying!" }
}
trait SpeakingAbility {
      String speak() { "I'm speaking!" }
}

class Duck implements FlyingAbility, SpeakingAbility {
      String quack() { "Quack!" }
      String speak() { quack() }
}

def d = new Duck()
assert d.fly() == "I'm flying!"
assert d.quack() == "Quack!"
assert d.speak() == "Quack!"
```

Figura 9 - Override de comportamento de um trait

1. A classe **Duck** implementa, ambos, FlyingAbility e SpeakingAbility.
2. Nova instância de Duck.
3. Podemos, então, chamar o método FlyingAbility, assim como, o método speak de SpeakingAbility.
4. Define um método específico em Duck, chamado quack.
5. Override a implementação padrão de speak para que possamos usar quack.
6. Quack vem da classe Duck.
7. Speak não usa mais a implementação do trait.

3.3.6. Closures

Closures nada mais são do que pedaços de código tratados como objetos, e como tal podem receber parâmetros e retornar valores. E como a JVM não sabe se o código que está rodando é Groovy ou Java, é perfeitamente possível dizer que um closure é apenas mais objeto para a JVM, tal qual uma String ou um Integer. Sua sintaxe é **{[closureParameters ->] statements}**.

```
{ item++ }
{ -> item++ }
{ println it }
{ it -> println it }
{ name -> println name }

{ String x, int y ->
      println "hey ${x} the value is ${y}"
}

{ reader ->
      def line = reader.readLine()
      line.trim()
}
```

Figura 10 - Exemplos basicos de construcao de um closure

```
class Aula {
  ArrayList alunos = new ArrayList()

  def adicionaAluno(Aluno nomeAluno) {
    alunos.add(nomeAluno)
  }

  def removeAluno(Aluno nomeAluno) {
    alunos.remove(nomeAluno)
  }

  def imprimeListaAlunos() {
    alunos.each() { alunoAtual -> println alunoAtual.toString() }
  }
}
```

```
public class Main {

  public static void main(String[] args) {
    Aluno aluno01 = new Aluno("Marcelo");
    Aluno aluno02 = new Aluno("Giuliana");
    Aluno aluno03 = new Aluno("Ana Beatriz");

    Aula historia = new Aula();
    historia.adicionaAluno(aluno01);
    historia.adicionaAluno(aluno02);
    historia.adicionaAluno(aluno03);

    historia.imprimeListaAlunos();
  }
}
```

Figura 11 - Exemplo de aplicacao de closure

Ao chamarmos o método imprimeListaAlunos() faremos uso de um closure para percorrer todos os membros do ArrayList da classe Aula (através do each()). Agora faça um teste e escreva as classes Aluno e Aula em Java e veja quantas linha a mais de código seriam necessárias para o mesmo resultado.

3.4. Métodos

Possuem implementação similar a Java, seus parâmetros podem ter ou não tipos definidos e podemos colocar valores padrões a eles. Além disso, o retorno dos métodos também pode ser arbitrário, assim como os parâmetros.

```
class Main {

    static def olaMetodos () {
        println ('Ola Metodos!\n')
    }

    static def int somaNumeros (int a, int b) {
        return a + b;
    }

    static def String concatenaString (String a, String b) {
        return a + b;
    }

    static def teste (a, b) {
        return a + b;
    }
}
```

Figura 12 - Exemplos de construção de métodos

Os parâmetros dos métodos podem ser nomeados, mas para isso devem receber os parâmetros como um Map, como mostrado a seguir:

```
def foo(Map args) { "${args.name}: ${args.age}" }
foo(name: 'Marie', age: 1)
```

Figura 13 - Parâmetros nomeados de um método

Além disso, há suporte com parâmetros com um número variável de parâmetros, o que possibilita também o override dos métodos. Caso o parâmetro passado seja um array, o argumento se transforma neste array passado.

```
def foo(Object[] args) { args.length }
assert foo() == 0
assert foo(1) == 1
assert foo(1, 2) == 2
```

Figura 14 - Exemplo de parâmetros infinitos

```
def foo(Object... args) { 1 }
def foo(Object x) { 2 }
assert foo() == 1
assert foo(1) == 2
assert foo(1, 2) == 1
```

Figura 15 - Override de métodos a partir de seus parâmetros

```
def foo(Object... args) { args }
Integer[] ints = [1, 2]
assert foo(ints) == [1, 2]
```

Figura 16 - Recebimento de um array em um método com parâmetros variáveis

3.5. Campos e propriedades

Assim como Java, podemos ter campos declarados nas classes que podem ser declarados como públicos e privados. Estes recebem valores e necessitam de métodos get e set para serem alterados. Por outro lado, temos as propriedades, estas servem para expor os campos, isto é, podemos utilizar seu valor e alterar o mesmo sem a necessidade de se criar métodos get e set. Ambos campos e propriedades podem ser declarados e utilizados em métodos e em traits.

```
trait Named {
    String name
}
class Person implements Named {}
def p = new Person(name: 'Bob')
assert p.name == 'Bob'
assert p.getName() == 'Bob'
```

Figura 17- Exemplo de properties

```
trait Counter {
    private int count = 0
    int count() { count += 1; count }
}
class Foo implements Counter {}
def f = new Foo()
assert f.count() == 1
assert f.count() == 2
```

Figura 18 - Exemplo de campos privados

```
trait Named {
    public String name
}
class Person implements Named {}
def p = new Person()
p.Named__name = 'Bob'
```

Figura 19 - Exemplo de campos publicos

4. Conclusão

A partir do demonstrado no relatório, conclui-se que a LP Groovy é uma alternativa extremamente viável para pessoas que desejam utilizar uma linguagem próxima a Java. Isto acontece, pois, a LP, além de ter uma sintaxe similar e possuir integração com Java, Groovy é mais otimizado, possui sintaxe mais simples, possui mais vantagens, como os traits, os tipos dinâmicos e todas características de Java, além de possuir integração maior com novos frameworks, como seu irmão Grails. Podemos dizer também que possui grande potencial para ultrapassar o uso de várias linguagens devido sua versatilidade e ferramentas disponíveis.

5. Referencias

[1] Groovy-Lang

groovy-lang.org/

[2] Groovy-Lang Documentation

groovy-lang.org/documentation.html

[3] Wikipedia: Groovy

pt.wikipedia.org/wiki/Groovy

[4] DevMedia: Artigo sobre Groovy

devmedia.com.br/artigo-java-magazine-69-um-pouco-de-groovy/12874

[5] Linguagens de Programação Groovy

inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-20182-seminario-groovy.pdf

[6] Groovy Tutorial For Beginners

youtube.com/watch?v=vDtENU-3Lwo

[7] HostGator: O que e groovy

hostgator.com.br/blog/o-que-e-groovy/

6. Apresentação