CAPÍTULO 13

APÊNDICE A - GUIA DE GROOVY PARA DESENVOLVEDORES JAVA

Groovy é a linguagem oficial do Spock framework, por isso é fundamental ter uma proficiência mínima com ela. Este capítulo é para o leitor que possui pouca ou nenhuma experiência coma essa linguagem. É um guia focado em desenvolvedores Java, para que façam o *"de x para"* (em Java é daquele jeito e em Groovy é desse) e tenham maior facilidade na escrita dos testes com Spock.

13.1 COMO FUNCIONA O GROOVY

A linguagem Groovy foi criada em **2003** por **James Strachan** e hoje possui seu código aberto sendo mantida pela **Apache Software Foundation** desde **2015**. É uma das linguagens cujo compilador gera arquivos executáveis pela **Máquina Virtual Java**, a **JVM**. Ou seja, aquivos escritos nessa linguagem, quando compilados, geram arquivos ***.class*** executáveis por qualquer **JVM** de versão de Java compatível. Na prática, isso significa que:

- Se uma IDE possui suporte a Java e Groovy, as classes escritas nessas duas linguagens conseguem trabalhar entre

si de forma transparente;

- Bibliotecas escritas em Groovy podem ser usadas por projetos Java e vice-versa.

Esse funcionamento é representado na figura a seguir.

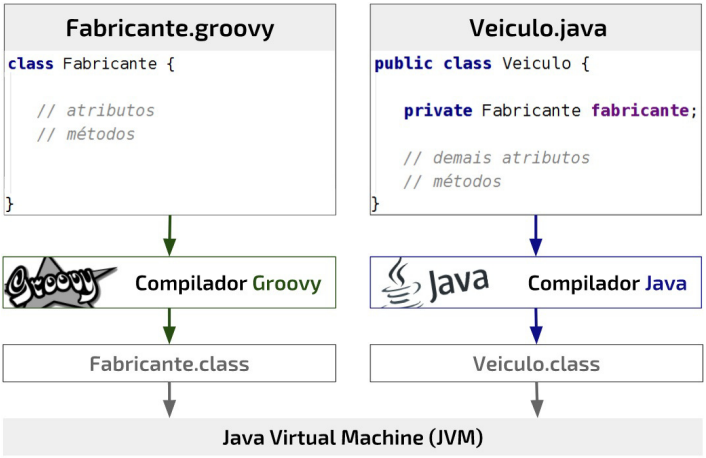

Figura 13.1: Funcionamento do Groovy na JVM.

É por isso que o Spock, que possui como linguagem de programação o Groovy, pode ser usado para criar testes para projetos em Java, Scala, Kotlin, Jython, JRuby ou qualquer outra linguagem que compile para a **JVM**. Consequentemente, também é possível criar testes para projetos que usam qualquer framework Java como **Spring MVC/Boot**, **Play**, **Struts**, **VRaptor** etc. e até mesmo para projetos **Android**.

Uma característica que reduz a "curva de aprendizado" dessa linguagem para desenvolvedores Java é a proximidade entre as sintaxes e a possibilidade de escrever um código Groovy exatamente como escreveria em Java, que vai compilar e funcionar

perfeitamente na imensa maioria dos casos. Ou seja, na dúvida em determinado trecho de código Groovy, faça exatamente como faria em Java.

13.2 CARACTERÍSTICAS E RECURSOS DO GROOVY

A seguir, alguns dos recursos e características do Groovy que podem ajudar muito na criação de testes com Spock.

Diferenças na sintaxe

A sintaxe do Groovy é muito parecida com do Java. Porém, algumas diferenças na sintaxe do Groovy podem melhorar a produtividade no desenvolvimento. A seguir, várias dessas diferenças serão apresentadas.

Ponto e vírgula são opcionais no final da linha de código

Caso o desenvolvedor queira abrir mão de usar ponto e vírgula (**;**) ao final de cada linha, o compilador Groovy vai considerar a quebra de linha como o fim da linha de código. Porém, se ela for utilizada, será o final explicito da linha de código, como é em Java. É possível usar linhas com e sem ponto e vírgula no final num mesmo arquivo Groovy sem problema algum.

`System.out` opcional

Caso seja necessário imprimir algo na saída padrão, é possível usar os `System.out.print()` e `System.out.println()` como em Java. Porém, como o `System.out` é opcional em Groovy, podem ser substituídos apenas por `print()` e `println()`,

respectivamente.

Parênteses são opcionais na invocação de métodos

Em Groovy, não é obrigatório o uso de parênteses na invocação de métodos, sejam eles com ou sem parâmetros. Como exemplo, temos o código a seguir, que imprime textos na saída padrão usando o `println` com e sem parênteses.

```
println("Não sou dono do mundo")
println "Mas sou filho do dono"
```

Outro exemplo: O código a seguir exemplifica a invocação de métodos da classe `java.lang.Math` sem parênteses, primeiro com um depois com dois parâmetros.

```
double raiz144 = Math.sqrt 144 // retornará a raiz de 144: 12
double doisAoCubo = Math.pow 2,3 // retornará 2 ao cubo: 8
```

Ratificando: O não uso de parênteses é **opcional**. Caso o leitor ache esse tipo de sintaxe confusa, pode usar os parênteses tal como usa na invocação de métodos em Java.

Por padrão, as classes são públicas

Quantas vezes você criou uma classe que não deveria ser `public`? Em Groovy, as **classes são públicas por padrão**. Ou seja, a classe `Futebolista` a seguir será compilada como pública pela JVM:

```
// Futebolista.groovy
class Futebolista {  // em tempo de execução: "public class Futeb
olista"
  // atributos, métodos, etc.
}
```

Por padrão, os atributos são privados

Quantas vezes você criou um atributo de instância que não deveria ser `private`? Em Groovy, os **atributos são privados por padrão**. Ou seja, na classe Groovy `Futebolista` a seguir, os atributos `nome` e `valorPasse` serão considerados privados pela JVM:

```
// Futebolista.groovy
class Futebolista {
  String nome // em tempo de execução: "private String nome"
  double valorPasse  // em tempo de execução: "private double val
orPasse"
}
```

Por padrão, os métodos são públicos

Você criou muito mais métodos `public`, `private` ou `default` em suas classes? Em Groovy, os **métodos são públicos por padrão**. Ou seja, a classe Groovy `Futebolista` a seguir teria um método considerado público (`treinar()`) e outro privado (`fugirDaConcentracao()`) pela JVM:

```
// Futebolista.groovy
class Futebolista {
  String nome
  double valorPasse

  void treinar() {  // em tempo de execução: "public void treinar
()"
    // treinando...
  }

  private void fugirDaConcentracao() {
    // fazendo a festa fora da concentração...
  }
}
```

É possível atribuir valores de atributos na criação de objetos

Em Java, quando queremos criar uma instância que já tenha

determinados valores em atributos, é necessário criar construtores para isso. Em Groovy, toda classe possui um construtor que aceita um `Map` como argumento (porém, com colchetes opicionais). Assim, é possível indicar os argumentos que quiser e seus valores. Por exemplo, seria possível instanciar um objeto do tipo `Futebolista` como nos exemplos do código a seguir:

```
 def jogadorA = new Futebolista(nome: 'Ze Boleiro', valorPasse: 1
00)
 def jogadorB = new Futebolista(valorPasse: 100, nome: 'Ze Boleir
o')
 def jogadorC = new Futebolista(nome: 'Ze Boleiro')
 def jogadorD = new Futebolista(valorPasse: 100)

 def jogadorE = new Futebolista([nome: 'Ze Boleiro', valorPasse:
100])

 Map mapaLoko = // Map criado em algum lugar...
 def jogadorF = new Futebolista(mapaLoko)
```

Métodos podem ter uma frase entre aspas como nome

Em classes de testes automatizados, normalmente um método testa um cenário. Imagine um cenario chamado "o valor do passe do jogador não pode ser menor ou igual a 0(zero)". Se fossemos nomear um método da maneira convencional, ou seja, uma palavra só em ***camelCase***, seu nome seria algo como...

```
def valorPasseJogaodorNaoAceitaZeroOuMenos() { ... }
```

Não acha que é um nome longo e difícil de ler? Uma alternativa seria reduzir o nome e por a descrição do cenário no `JavaDoc` do método. Porém, caso o teste falhe, é seu nome que é impresso no log da execução dos testes.

Usando Groovy, o nome do método poderia ser...

```
def 'valor do passe do jogador não pode ser menor ou igual a 0'()
```

`{ ... }`

Usando esse recurso, o cenário do teste fica literalmente descrito no nome do método. Outra vantagem é que se o teste falhar, o log de execução conterá os cenários e não nomes de métodos longos e/ou difíceis de ler.

Podem ser usadas aspas simples ou duplas e até mesmo caracteres especiais (letras acentuadas, cê-cedilha, etc.) podem ser usados.

A tipagem pode ser estática ou dinâmica

Para reduzir o tempo de programação, em Groovy é possível criar objetos e métodos com o operador `def`, usando o recurso da **tipagem dinâmica**. Em caso de métodos, é possível omitir o tipo dos parâmetros. Vejamos exemplos a seguir.

No exemplo a seguir, foi usada a tipagem dinâmica para ambos os métodos. Em tempo de execução, a JVM vai considerar que eles retornam *Object*.

```
  def metodoX(p1, p2) { // em tempo de execução: "public Object m
etodoX(Object p1, Object p2)"
    // return ...;
  }

  def metodoY(def n1, def n2) { // em tempo de execução: "public
Object metodoY(Object n1, Object n2)"
    // return ...;
  }
```

Se o método for `static`, até mesmo o `def` é opcional, como nos exemplos a seguir, cujos métodos serão considerados *públicos*, *estáticos* e com retorno *Object* em tempo de execução:

```
  static metodoX(p1, p2) { // em tempo de execução: "public stati
```

```
c Object metodoX(Object p1, Object p2)"
    // return ...;
  }

  static def metodoY(def n1, def n2) { // em tempo de execução: "
public static Object metodoY(Object n1, Object n2)"
    // return ...;
  }
```

Se o desenvolvedor preferir, pode deixar a declaração do método ou dos argumentos estática, como nos exemplos a seguir:

```
  Double metodoX(n1, n2) { // em tempo de execução: "public Doubl
e metodoY(Object n1, Object n2)"
    // return ...;
  }

  Double metodoY(Double n1, n2) { // em tempo de execução: "publi
c Double metodoY(Double n1, Object n2)"
    // return ...;
  }

  void metodoZ(Double n1, Double n2) { // em tempo de execução: "
public void metodoZ(Double n1, Double n2)"
    // return ...;
  }
```

Objetos também podem ser definidos com esse recurso, como no exemplo a seguir onde primeiro é criado um BigDecimal declarado com tipagem dinâmica e, depois um outro com tipagem estática.

```
  def numero0 = new BigDecimal(0) // em tempo de execução: BigDec
imal numero0 = ...
  BigDecimal numero1 = new BigDecimal(1)
```

Uma boa notícia para o desenvolvedor é que as principais IDEs Java do mercado - Eclipse, IntelliJ e NetBeans - sugerem os métodos e atributos do tipo usado na instanciação. Ou seja, no último código, após `numero0.` essas IDEs iriam sugerir métodos e

atributos da classe `BigDecimal`.

Operador `==` faz o papel do `.equals()`

Em Java, o operador `==` verifica se os membros comparados são, literalmente, o mesmo objeto, a mesma instância. Para saber se possuem *conteúdo* igual, deve-se usar o método `.equals()`. Inclusive é comum que iniciantes em Java tentem comparar duas Strings de mesmo conteúdo com `==` e acabam recebendo um `false` como resultado.

Acreditando que a maioria dos desenvolvedores faz muito mais comparação de conteúdo do que de referência de instância em seus projetos, os arquitetos da linguagem Groovy definiram que, nela, o `==` correspondesse ao `.equals()` do Java. Vejamos o exemplo a seguir, onde `texto1` e `texto2` têm seus conteúdos comparados:

```
   String texto1 = // valor recebido em tempo de execução
   String texto2 = // valor recebido em tempo de execução
   def teste = texto1 == texto2
   // se o conteúdo de "texto1" e "texto2" forem iguais, "teste"
será "true"
```

Caso exista a necessidade de saber se dois objetos são a mesma instância, deve-se usar o método `.is()`, conforme o exemplo a seguir:

```
   String texto1 = // valor recebido em tempo de execução
   String texto2 = // valor recebido em tempo de execução
   String texto3 = texto1
   def teste1e2 = texto1.is(texto2) // "false"
   def teste1e3 = texto1.is(texto3) // "true"
```

Concatenação de valores String mais dinâmica e simples

Em Groovy, um objeto `String` pode ser instanciado com aspas simples, como no exemplo a seguir:

```
String poema = 'Nas curvas do teu corpo capotei meu coração'
```

Porém, ao se fazer isso, perde-se a possibilidade de usar a capacidade de **concatenações dinâmicas do Groovy**, que só é possível quando se usa **aspas duplas**. Basta usar o operador `$` e o nome do objeto a ser concatenado no texto. Vejamos o exemplo a seguir, onde `time1`, `time2` e `dias` serão concatenados no conteúdo de `anuncio`:

```
def time1 = 'Tabajara'
def time2 = 'Sayajins'
def dias = 7
def anuncio = "O jogo será entre $time1 e $time2. Faltam $dias di
as, não percam!"
// "O jogo será entre Tabajara e Sayajins. Faltam 7 dias, não per
cam!"
```

A montagem de um texto pode incluir resultados da invocação de métodos e/ou de operações matemáticas. Assim como o recurso anterior, é possível **executar métodos e operações matemáticas e concatenar seus resultados** em String Groovy. Nesse caso, a expressão deve estar dentro de `${}`. No exemplo a seguir, o resultado de `toUpperCase()` em `time1` e de `substring(4)` sobre `time2`, bem como o resultado de `valor` dividido por 2 serão concatenados no conteúdo de `anuncio`.

```
def time1 = 'Tabajara'
def time2 = 'Sayajins'
def valor = 20
def anuncio = "O jogo será entre ${time1.toUpperCase()} e ${time2
.substring(4)}. Mulheres pagam ${valor/2}, não percam!"
// "O jogo será entre TABAJARA e jins. Mulheres pagam 10, não per
cam!"
```

Caso exista a necessidade de montar um texto que deva estar tão associado a um objeto ao ponto que seu conteúdo acompanhe alterações feitas nesse objeto, pode-se lançar mão de outro tipo de concatenação mais dinâmica ainda, **verificando o valor atual de um objeto sempre que a String for invocada**. Para isso, o objeto deve estar dentro de `${->}` .

No código do próximo exemplo, associamos `time1` e `time2` à `anuncio` de tal forma que, sempre que `anuncio` for usada, seu conteúdo pode variar, de acordo com o conteúdo de `time1` e `time2` .

```
def time1 = 'Tabajara'
def time2 = 'Sayajins'
def frase = "O jogo será entre ${-> time1} e ${-> time2}"
// nesse momento, "anuncio" é
// "O jogo será entre Tabajara e Sayajins"

time1 = 'Remo'
time2 = 'Paysandu'
// a partir desse momento, "anuncio" passa a ser
// "O jogo será entre Remo e Paysandu"
```

String podem ter múltiplas linhas sem concatenações

Quem já precisou escrever uma instrução SQL ou criar um *mock* de um JSON em Java já deve ter feito todo tipo de malabarismo para que uma String de várias linhas não fique tão ilegível no código. Em Groovy existe o recurso de múltiplas linhas. Basta iniciar o usar **três aspas simples ou duplas** para instanciar um String.

No exemplo de String de múltiplas linhas a seguir, é criada uma instrução SQL com 3 linhas.

```
def consultaTimesSemTitulo = '''
```

```
select * from tb_time
where num_titulos = 0
order by dt_cricao desc
'''
```

Caso você precise criar uma String de múltiplas linhas com concatenação de métodos e/ou operações matemáticas, basta usar **três aspas duplas** nos limites da String. O exemplo a seguir monta uma instrução SQL de múltiplas linhas concatenando seu conteúdo com `uf`.

```
def uf = 'AM'
def consultaTimesAmazonas = """
select * from tb_time
where estado = ${uf}
"""
```

Interoperabilidade com classes Java

A interoperabilidade de Groovy com classes Java é total, contanto que as versões de Java usadas na compilação de ambos sejam compatíveis. Trata-se da mesma restrição entre classes de arquivos escritos em Java.

Para exemplificar essa interoperabilidade, consideremos a classe Java `apploko.pjava.Futebolista`, que contém três atributos simples, um construtor que altera todos eles e seus *getters* e *setters*:

```
package apploko.pjava

// imports

public class Futebolista {
  private String nome;
  private double habilidade;
  private double velocidade;
```

```
  public Futebolista(String nome, double habilidade, double veloc
idade) {
    // atribuições simples
  }

  // getters e setters públicos de todos os atributos
}
```

A classe Groovy `FutebolistasLokos` a seguir usa a classe Java `apploko.pjava.Futebolista` de forma transparente: uma instância é criada usando seu construtor e um de seus métodos, o `setNome()`, é invocado.

```
import apploko.pjava.Futebolista
// demais imports

class FutebolistasLokos {
  Futebolista createHabilidoso() {
    def habilidoso = new Futebolista("Romário", 95, 90)
    habilidoso.setNome("Romário 11")
    return habilidoso
  }
}
```

Se o projeto utilizar **Maven** ou **Gradle**, qualquer classe das dependências também será acessível de forma transparente pelas classes Groovy.

NullPointerException é muito fácil de ser evitado

Se o leitor já trabalha com Java há algum tempo já deve ter visto o `java.lang.NullPointerException` (vulgo **NPE**) várias vezes quando executava seu projeto. Isso ocorre quando tentamos invocar um método de um objeto nulo (`null`), o que acontece com uma certa frequência porque é muito trabalhoso (e fácil de esquecer) verificar a cada *getXXX()* se no temos um `null`. Em Groovy esse problema é simples de resolver, pois existe o chamado

operador de navegação segura (**Safe navigation operator**), que previne o NPE apenas com o uso de `?` antes do `.`. Vide o próximo código.

```
  def jogador = // instanciado em tempo de execução a partir do b
anco de dados, por exemplo
  def pais = jogador?.getTime()?.getPais()?.getNome();
```

No último código, caso `jogador` ou `getTime()` ou `getPais()` seja `null`, o objeto `pais` simplesmente receberá `null`, **sem NPE**. Caso o operador `?` não seja usado, o risco de **NPE** é o mesmo de uma classe Java.

Manipulação de coleções é muito simples

A manipulação de coleções em Java é considerada *verbosa* demais por desenvolvedores acostumados com linguagens como **PHP**, **Python**, **Ruby** e **JavaScript**. Em Groovy, a manipulação de coleções é bem simples, se comparada com Java, conforme apresentado a seguir.

Criando uma coleção com valores Em Java, para criar uma coleção com valores deve-se recorrer a classes utilitárias, sejam elas do Java ou de bibliotecas de terceiros. Em Groovy, esse tipo de operação é muito simples, como nos exemplos a seguir. No próximo código, há uma exemplo da criação de uma `List` em Groovy. Ele cria uma instância de `ArrayList` com 4 (quatro) elementos.

```
def posicoes = ['goleiro', 'zagueiro', 'meia', 'atacante']
```

O interessante no código anterior é que as IDEs Eclipse e IntelliJ **inferem** o tipo de objeto da coleção quando se cria uma usando todos os elementos do mesmo tipo, Assim, quando tentar a

recuperação de elementos dessa coleção, a IDE exibirá que o retorno seria do tipo `String`. Caso não consigam fazer a inferência, a coleção será tratada como de `Object`.

No próximo código, há um exemplo da criação de um `Map` em Groovy. Ele cria uma instância de `HashMap` com 3 (três) elementos, com as chaves `derrotas`, `empates` e `vitorias` cujos valores são `0`, `1` e `3`, respectivamente.

```
def campanha = ['derrotas': 0, 'empates': 1, 'vitorias': 3]
```

Um detalhe bem útil em Groovy é a possibilidade de ser opcional o uso de aspas quando as chaves do mapa são `String`. Portanto, no código todas as três chaves poderiam estar sem aspas, como no código a seguir.

```
def campanha = [derrotas: 0, empates: 1, vitorias: 3]
```

Criando coleções vazias Em Java, para criar uma coleção vazia, devemos usar construtores como faríamos para qual outro tipo de classe. Em Groovy, é possível usar os mesmos construtores, porém as principais coleções podem ser criadas com o uso de operadores, como nos exemplos a seguir. No código Groovy a seguir há um exemplo de criação de `List` **vazia**, implementada como `ÀrrayList`.

```
def posicoes = []
```

No código a seguir há um exemplo de criação de `Map` **vazia** implementada como `HashMap` em Groovy.

```
def pontuacoes = [:]
```

Adicionando um elemento a uma coleção Em Java, para adicionar elementos a uma coleção devemos recorrer a métodos,

como `add()` ou `put()`. Em Groovy é possível usar exatamente os mesmos métodos, porém também é possível usar **operadores**, como nos exemplos a seguir. No próximo código, há uma exemplo da criação de uma `List` vazia e posterior inclusão de elementos nela.

```
def posicoes = []
posicoes += 'goleiro'
posicoes += 'zagueiro'
```

Há três maneiras de incluir um elemento num `Map` em Groovy, todas exemplificados no código a seguir.

```
def campanha = [:]
campanha.derrotas = 0     // chave: 'derrota' / valor: 0
campanha['empates'] = 1   // chave: 'empate' / valor: 1
campanha += [vitorias: 3] // chave: 'virotia' / valor: 3
```

Recuperação de itens de uma coleção Em Java, para recuperar um item de uma coleção devemos recorrer a métodos, como o `get()`. Em Groovy é possível usar exatamente os mesmos métodos, porém também é possível usar **operadores**, como nos exemplos a seguir. No próximo código, há exemplos da recuperação de um elemento de uma `List`.

```
def posicoes = ['goleiro', 'zagueiro', 'meia', 'atacante']
println(posicoes[0]) // imprimiria 'goleiro'
println(posicoes[2]) // imprimiria 'meia'
```

Há duas maneiras de recuperar um elemento de um `Map` em Groovy, todas exemplificados no código a seguir.

```
def campanha = [derrota: 0, empate: 1, vitoria: 3]
campanha.derrota    // 0
campanha['vitoria'] // 3
```

Recuperando o último elemento de uma lista ou vetor Imagige que você precise do último elemento em uma lista mas

não sabe quantos elemento ela conterá no momento em que precisar dessa informação. Em Java é preciso verificar o tamanho da lista e usar seu valor menos 1 para recuperar o último elemento de uma lista. Em Groovy, há um método chamado `last()` que abstrai isso e recupera o último elemento de uma lista, como no exemplo a seguir.

```
def listaDinamica = // recuperada de forma dinâmica (ex: de um ba
nco de dados)
def ultimo = listaDinamica.last()
```

O método `last()` **não existe em Mapas**, estando disponível apenas para listas e vetores.

Técnicas de iteração simples e embarcadas para números, coleções e Strings

É uma tarefa muito comum realizar iterações (repetições) para resolver problemas computacionais. Essas iterações são comumente baseadas em valores númericos, nos elementos de uma coleção ou nas linhas de um texto. A seguir, veremos como essas iterações mais comuns podem ser feitas em Groovy.

Iteração a partir de números Na necessidade da repetição de um trecho de código determinada por um número **N** em Java, é necessário criar estruturas de repetição como `for` , `do-while` ou `while` . Em Groovy, os números inteiros possuem um recurso embarcado que atende facilmente essa necessidade. É a closure `times{}` . No exemplo a seguir, está programada uma repetição de um trecho de código por 11 (onze) vezes.

```
11.times{
    // o código aqui repetirá 11 vezes
}
```

Em vez do número 11, poderíamos ter aplicado o mesmo recurso sobre uma variável do tipo `Integer`. Caso seja necessário saber em que passo da iteração estamos, basta usar a variável `it`, que é **injetada automaticamente** na Closure `times{}`. Ela vai **de 0 a N-1**, onde **N** é o número de iterações. O código do exemplo a seguir demonstra como poderíamos lançar mão desse recurso.

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.size().times{
    println(jogadores[it].getNome())
}
```

Por questões de legibilidade de código ou para o caso de iterações aninhadas, é possível definir um nome da variável do passo da iteração. No exemplo a seguir, usamos o nome `j`.

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.size().times{ j ->
    println(jogadores[j].getNome())
}
```

Iteração a partir de `List` e `Set`

Quando há repetição de um trecho de código para cada elemento de uma coleção em Java, é necessário criar estruturas de repetição como `for`, `do-while`, `while` ou usar os `streams` ou método `forEach()` do **Java 8**. Em Groovy, as coleções possuem um recurso embarcado que atende facilmente essa necessidade. São as closures `each{}` e `eachWithIndex{}`, apresentadas e exemplificadas a seguir. No exemplo a seguir, está programada uma repetição para os itens de uma `List`, mas que funcionaria da mesma forma para `Set`. O `it` dentro da closure é o nome que cada elemento da lista recebe na iteração.

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.each{
```

```
    println("Nome do Jogador: ${it.getNome()}")
}
```

Por questões de legibilidade de código ou para o caso de iterações aninhadas, é possível definir um nome do elemento na iteração. No exemplo a seguir, usamos o nome `jogador`.

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.each{ jogador ->
    println("Nome do Jogador: ${jogador.getNome()}")
}
```

Caso seja necessário saber em que passo da iteração estamos, podemos usar a closure `eachWithIndex`. Nesse caso, **é obrigatório** indicar os nomes do elemento e da variável que conterá o índice (passo) da iteração (que começa em 0), como demonstrado no código do exemplo a seguir, onde chamamos o elemento de `jogador` e o índice de `i`.

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.eachWithIndex{ jogador, i ->
    println("Nome do ${i+1} Jogador: ${jogador.getNome()}")
}
```

Iteração a partir de `Map` Quando há repetição de um trecho de código para cada elemento de um mapa (`Map`) em Java, é necessário criar estruturas de repetição como `for`, `do-while`, `while` ou usar os `streams` ou método `forEach()` do **Java 8**. Em Groovy, as coleções possuem um recurso embarcado que atende facilmente essa necessidade. É a closures `each{}`, apresentada e exemplificada a seguir.

No exemplo a seguir, está programada uma repetição para os itens de um `Map`. O `it` dentro da closure é o nome que cada elemento do map recebe na iteração. O atributo `key`, é o **chave** e o `value`, o **valor** do elemento.

```
def campanha = [derrotas: 0, empates: 2, vitorias: 5]
println("Campanha:")
campanha.each{
    println("${it.key}: ${it.value}") // ex de saída: "vitorias:
5"
}
```

Por questões de legibilidade de código ou para o caso de iterações aninhadas, é possível definir um nome do elemento na iteração. No exemplo a seguir, usamos o nome `campanha`.

```
def campanha = [derrotas: 0, empates: 2, vitorias: 5]
println("Campanha:")
campanha.each{ campanha ->
    println("${campanha.key}: ${campanha.value}") // ex de saída:
 "vitorias: 5"
}
```

É possível ainda definir diretamente um nome para a chave e outro para o valor, como no próximo código, onde os chamamos de `resultado` e `quantidade`, respectivamente.

```
def campanha = [derrotas: 0, empates: 2, vitorias: 5]
println("Campanha:")
campanha.each{ resultado, quantidade ->
    println("${resultado}: ${quantidade}") // ex de saída: "vitor
ias: 5"
}
```

Verificação de "verdadeiro/falso" expandida, porém facilitada

A verificação de **verdadeiro/falso** (**true/false**), provavelmente, a mais comum em testes automatizados. É possível fazer esse tipo de verificação de forma simplificada em Groovy conforme as situações a seguir.

`null` é `false` e não nulo é `true` Qualquer objeto que seja

`null`, quando testado em um `assert` ou `if` ou **operador ternário** ou atributido a uma variável `boolean`, será interpretado como `false`. Vide o código Groovy de exemplo a seguir, cujo `assert` **falharia**.

```
def jogador = null
assert jogador
```

Nas mesmas situações recém-citadas, qualquer objeto **não nulo**, **a princípio** é `true`. No código do exemplo a seguir, o `assert` **passaria**.

```
def jogador = new Futebolista()
assert jogador
```

Observe: foi dito que **a princípio**, o resultado é `true`. Existem casos onde o objeto não é nulo, mas sua verificação booleana pode ser `false`. Essas exceções serão apresentadas a seguir.

`String` vazia é `false` e não vazia é `true`

Qualquer `String` que seja **vazia**(`""` ou `''`), quando testado em um `assert` ou `if` ou **operador ternário** ou atributido a uma variável `boolean`, será interpretado como `false`. Vide o código Groovy de exemplo a seguir, cujo `assert` **falharia**.

```
def nomeNogador = ""
assert nomeJogador
```

Nas mesmas situações recém-citadas, qualquer `String` **não vazia**, será considerado `true`. No código do exemplo a seguir, o `assert` **passaria**.

```
def nomeNogador = "Zé Loko"
assert nomeJogador
```

Número 0(zero) é `false`. Qualquer outro é `true`

Qualquer número, qualquer tipo numérico (inclusive `BigDecimal`) que seja **0**(zero), quando testado em um `assert` ou `if` ou **operador ternário** ou atributido a uma variável `boolean`, será interpretado como `false`. Vide o código Groovy de exemplo a seguir, cujos `asserts` **falhariam**.

```
def numeroLoko1 = 0
def numeroLoko2 = 0.0
def numeroLoko3 = new BigDecimal(0)
assert numeroLoko1
assert numeroLoko2
assert numeroLoko3
```

Nas mesmas situações recém-citadas, qualquer número **diferente de zero, mesmo negativo**, será considerado `true`.

Coleções vazias são `false` e não vazias são `true` Qualquer coleção (`List`, `Set` ou `Map`) que seja **vazia**(sem elementos), quando testado em um `assert` ou `if` ou **operador ternário** ou atributido a uma variável `boolean`, será interpretado como `false`. Vide o código Groovy de exemplo a seguir, cujo `assert` **falharia**.

```
def jogadores = []
assert jogadores
```

Nas mesmas situações recém citadas, qualquer coleção **não vazia**, será considerado `true`. No código do exemplo a seguir, o `assert` **passaria**.

```
def campanha = [derrotas:0, vitorias:7]
assert campanha
```

Métodos embarcados para as conversões mais comuns

A conversão de tipos é uma operação muito comum nos problemas computacionais atuais, principalmente devido à grande quantidade de integração de sistemas que ocorre hoje. Em Java, as conversões são um tanto verbosas. Em Groovy há métodos de conversão embarcados para as conversões mais comuns do cotidiano, como exemplificado a seguir.

Conversão de `String` para qualquer tipo numérico

O tipo `String` em Groovy já possui métodos embarcados para a conversão para todos os tipos numéricos da plataforma Java. No código de exemplo a seguir, variáveis `String` são convertidas com sucesso para `Integer`, `Long`, `BigInteger`, `Float`, `Double` e `BigDecimal`.

```
def txtNumericoInteiro = '8'
Integer inteiro = txtNumericoInteiro.toInteger()
Long longo = txtNumericoInteiro.toLong()
BigInteger bigInteger = txtNumericoInteiro.toBigInteger()

def txtNumericoReal = '8.5'
Float flutuante = txtNumericoReal.toFloat()
Double duplo = txtNumericoReal.toDouble()
BigDecimal bigDecimal = txtNumericoReal.toBigDecimal()
```

Vale destacar que caso o valor da `String` não conter um número válido para a conversão solicitada, uma exceção será lançada.

Conversão de `String` para `Boolean`

O tipo `String` em Groovy já possui métodos embarcados para a conversão para `Boolean`. Os valores `"true"`, `"y"` e `"1"` são convertidos para `true`. Qualquer outro valor é convertido para `false`. No código do exemplo de conversão a seguir, todas as conversões resultariam em `true`, portanto, todos

os `assert` **passariam**.

```
def textoBoleano = 'true'
assert textoBoleano.toBoolean()
textoBoleano = 'y'
assert textoBoleano.toBoolean()
textoBoleano = '1'
assert textoBoleano.toBoolean()
```

No código do exemplo de conversão a seguir, todas as conversões resultariam em `false`, portanto, todos os `assert` **falhariam**.

```
def textoBoleano = 'false'
assert textoBoleano.toBoolean()
textoBoleano = 'oi'
assert textoBoleano.toBoolean()
textoBoleano = ''
assert textoBoleano.toBoolean()
```

Manipulação de datas é muito simples

A manipulação de datas é uma operação muito comum nos problemas computacionais atuais. Em Java, as conversões `Date` para `String` e vice versa, bem como adicionar ou subtrair dias a datas são um tanto verbosas, mesmo na versão 8. Em Groovy há métodos de conversão embarcados que facilitam as operações mais comuns com datas, como exemplificado a seguir.

Conversão de `String` para `Date` Para converter uma `String` para `Date`, basta usar o método embarcado `parse()`, informando o formato (*pattern*) e o texto a ser convertido. O código a seguir demonstra como converter um texto em data no formato `"dd/MM/yyyy"`.

```
def txtDataLoka = '01/01/1980'
def dataLoka = Date.parse('dd/MM/yyyy', txtDataLoka)
```

Conversão de `Date` para `String` Para converter uma `Date` para `String`, basta usar o método embarcado `format()`, informando o formato (*pattern*). O código a seguir demonstra como converter uma data em texto no formato `"dd/MM/yyyy"`.

```
def agora = new Date()
def txtAgora = agora.format('dd/MM/yyyy')
```

Adicionando ou subtraindo dias de `Date` Em Groovy, para adicionar ou subtrair dias de uma data pode-se simplesmente usar os operadores `+` e `-`, literalmente, como é exemplificado no código a seguir, que **subtrai 1 dia** a uma data e depois **adiciona 2 dias** a outra.

```
def hoje = new Date() // por exemplo: 02/01/2000
def ontem = hoje-1    // 01/01/2000
def amanha = ontem+2  // 03/01/2000
```

No código anterior, o objeto `hoje` **não foi alterado**. Para que a data seja realmente alterada, é preciso, literalmente, substituir seu valor, como no exemplo a seguir.

```
def dataLoka = new Date() // por exemplo: 02/01/2000
dataLoka = dataLoka-1     // 01/01/2000
dataLoka = dataLoka+2     // 03/01/2000
dataLoka++                // 04/01/2000
```

Caso o operador `-` seja aplicado entre datas, o resultado será **a diferença de dias entre a primeira e a segunda**. O código a seguir demonstra isso ao comparar a data de hoje com a de amanhã.

```
def hoje = new Date() // por exemplo: 01/01/2000
def amanha = hoje+1   // 02/01/2000
println(amanha-hoje)  // resultado: 1
println(hoje-amanha)  // resultado: -1
```

Indo para a zero hora de uma `Date` Em Groovy, é possível ir para a *zero hora* de uma data usando o método embarcado

`clearTime()`. **Importante:** este método **altera** a data no qual é invocado, não retorna uma versão ajustada para a *zero hora*. O exemplo a seguir ajustaria a data na qual foi invocado para a zero hora de seu dia.

```
def hoje = new Date() // por exemplo: 01/01/2000 15:30:00
hoje.clearTime()      // 01/01/2000 00:00:00
```

Acesso direto a métodos privados

Em Java, um método privado (`private`) não pode ser invocado, a não ser com uso da `Reflection API`, o que não é tarefa simples. Em Groovy, o acesso a métodos privados é transparente, sendo possível invocá-los como se fossem públicos. O código do exemplo a seguir é de uma classe Java que possui o método privado `atualizarClassificacao()` que é invocado por seus outros dois métodos que são públicos, o `gol()` e o `finalizar()`.

```
public class Partida {
    public void gol(Jogador autor) {
       // ações a cada gol...
       this.atualizarClassificacao();
    }

    public void finalizar() {
       // ações ao finalizar partida
       this.atualizarClassificacao();
    }

    private void atualizarClassificacao() {
       // ações da atualização da classificação
    }
}
```

Para invocar o método `atualizarClassificacao()` de uma instância de `Partida` a partir de uma classe Groovy, basta invocá-

lo diretamente, como se fosse público. O trecho de código Groovy a seguir instancia uma `Partida` e invoca seu método privado. Inclusive as IDEs Eclipse, Intellij e NetBeans sugerem e aceitam a invocação de métodos privados durante a edição de código Groovy.

```
def partidaLoka = new Partida();
partidaLoka.atualizarClassificacao()
```

Esse recurso é muito útil em casos como o do último exemplo, onde métodos privados possuem muita relevância, pois permite a criação de vários cenários de teste diretamente para eles.

Alteração do comportamento de métodos em classes e objetos em tempo de execução

Em algumas situações, métodos podem fazer ações muito complexas e/ou que dependem de elementos externos. Como exemplos temos: acesso a bancos de dados, consumo de APIs e uso de bibliotecas de terceiros. Para casos como esse, pode-se **sobrescrever** o comportamento do método em tempo de execução em Groovy, sem a necessidade de uma subclasse anônima nem de uma instância *mock*. Tomemos como base a classe a seguir como exemplo, a `ClienteRestTimesFutebol` e seu método `getToken()`. Esse método solicitaria um *token* de autenticação junto a uma API a partir de um *usuario* e uma *senha*.

```
public class ClienteRestTimesFutebol {
    public String getToken(String usuario, String senha) {
        // chamada à API para obter 'token'
    }
}
```

No código a seguir, definimos que **uma determinada instância**

de `ClienteRestTimesFutebol`, a que chamamos de `clienteLoko` passa a ter um comportamento **diferente** para o método `getToken()`, enquanto que a instância `clienteNormal` continua com seu método original:

```
def clienteLoko = new ClienteRestTimesFutebol()
cliente1.metaClass.getToken = { String u, String s ->
    return "fake-token"
}

def clienteNormal = new ClienteRestTimesFutebol()

def token1 = clienteLoko.getToken(null, null) // apenas retorna "
fake-token"
def token2 = clienteNormal.getToken(null, null) // invocaria a ve
rsão original do "getToken()"
```

Caso seja necessário que todos os objetos de uma classe tenham o comportamento de um método alterado, é possível fazer esse ajuste **na classe**, consequentemente, **em todas as suas instâncias**. Como no exemplo a seguir, que mudaria o `getToken()` para qualquer instância de `ClienteRestTimesFutebol`:

```
ClienteRestTimesFutebol.metaClass.getToken = { String u, String s
 ->
    return "fake-token"
}
def cliente1 = new ClienteRestTimesFutebol()
def cliente2 = new ClienteRestTimesFutebol()

def token1 = cliente1.getToken(null, null) // apenas retorna "fak
e-token"
def token2 = cliente2.getToken(null, null) // apenas retorna "fak
e-token"
```

getters e *setters* podem invocados apenas com o nome do atributo

Os métodos `getters` e `setters` fazem parte do dia a dia dos desenvolvedores da plataforma Java. Fazem parte do padrão **JavaBeans** (https://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html) e são usados pelos principais *frameworks* Java. Seu uso é tão comum que as principais IDEs Java possuem assistentes para a rápida criação desses métodos. Em Groovy, existe a possibilidade invocar esses métodos apenas usando o nome do atributo. `valorPasse` privado, acessível com um `get` e um `set` escritos de maneira correta:

```
public class Futebolista {
  private double valorPasse;
  public double getValorPasse() {
      return this.valorPasse;
  }
  public void setValorPasse(double valorPasse) {
      this.valorPasse = valorPasse;
  }
}
```

Para acessar os `setValorPasse()` e `getValorPasse()` em uma classe Groovy, poderíamos escrever um código como o do exemplo a seguir:

```
Futebolista jogadorLoko = // obtendo o Futebolista
jogadorLoko.valorPasse = 50000000
def passeDoLoko = jogadorLoko.valorPasse
```

Apesar de parecer que houve um acesso direto ao atributo, **não houve**. O compilador Groovy verifica se existe um `getValorPasse()` na segunda linha e se existe um `setValorPasse(double parametro0)` na terceira. Assim, a versão compilada do último código, para a JVM, seria como o código a seguir:

```
Futebolista jogadorLoko = // obtendo o Futebolista
jogadorLoko.setValorPasse(50000000)
double passeDoLoko = jogadorLoko.getValorPasse()
```

Caso o desenvolvedor ache melhor usar os `getters` e `setters` de forma explícita em suas classes Groovy, pode fazê-lo sem problema algum.

Conclusão

Os recursos apresentados neste capítulo são o suficientes para o leitor iniciante em Groovy, porém existem ainda muito mais recursos interessantes nessa linguagem. Se o leitor desejar um guia completo, sua documentação oficial (http://groovy-lang.org/documentation.html) é muito bem escrita e rica em exemplos.