

# Linguagem de Programação **Groovy**



**Alunos:**

. Arthur Severo - 20183021106

. Victor Le Roy – 20183021222

# Introdução

- LP criada por James Strachan que teve a ideia de desenvolvimento em 2003.
- Teve sua primeira versão disponível ao público em 2007 e hoje se encontra na fase alfa do desenvolvimento em sua versão 4.0, com suporte contínuo a 3.0.
- Atualmente usada por grandes empresas, como LinkedIn, BestBuy, Netflix e Google.

# Introdução: Utilizacao

- **Desenvolvimento web**
  - Por ser uma linguagem ágil e dinâmica, o Groovy pode ser usado em outros frameworks além do Grails, como Micronaut, Spring Boot, Ratpack e outros.
- **Linguagem embarcada**
  - O Groovy pode ser utilizado de forma complementar ao Java por meio da construção de plataformas estendidas em tempo de execução pelos usuários com a inclusão, a edição e a remoção de scripts.
- **Ferramenta de automatização**
  - O código Groovy pode ser executado na forma de scripts – e isso abre caminho para a automatização, como agendamento de tarefas de manutenção, raspagem de dados, implementação de crawlers e muito mais.

# Motivação

- A linguagem foi idealizada como uma alternativa à seu parente mais popular (Java).
- Foi enviado ao Java Community Process (JCP), que é um processo que permite ao ganhador participar nas versões futuras do Java, e foi aceita .
- Em 2007 ganhou em primeiro lugar o prêmio de inovação JAX e em 2008 um de seus frameworks web, *Grails*, ficou em segundo lugar na mesma premiação.
- Foi comprada pela SpringSource e por sua vez, foi comprada pela VMWare em 2009. Em 2015 teve seu patrocínio dissolvido pela empresa anterior e foi integrada à Apache Software Foundation até ser promovido à um projeto top-level no final do mesmo ano.

# Motivação

- É baseada em Java e possui características de linguagens como Python, Ruby e Smalltalk.
- É compilada em bytecode Java próprio para ser interpretado em uma JVM (Java Virtual Machine) e tem sintaxe facilmente integrável em projetos desenvolvidos na linguagem.
- Groovy é totalmente integrado 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 por que, por debaixo dos panos, tudo é bytecode Java.

# O que é o Groovy afinal?

- De acordo com o site da linguagem:

*" **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."*

- 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');  
    }  
}
```

# 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.

# Características da linguagem

- Escopo dinâmico.
- Sobrecarga de operadores.
- Possui características em comum com Java, já que é inspirada nela.
- Compatibilidade:
  - 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 contrario não é verdade. A GDK estende a JDK adicionando métodos que não existem originalmente nos objetos Java.



# 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.

# Particularidades

- O símbolo ";" eh 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 eh, declarar variáveis com tipos definidos, como **int** ou **double**.
- Possui precedência de operadores (sobrecarga de operadores).

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

# Tipagem estática e dinâmica

- Um ponto curioso da Groovy é a possibilidade de desenvolver código dinamicamente enquanto se baseia na escrita estática de código, 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.

```
class Main {  
  
    static void main (String[] args) {  
  
        def a = 2;  
        println('a = ' + a)  
        a = 'Oi';  
        println('a = ' + a);  
  
        int b = 3;  
        println('b = ' + b);  
  
        String c = "Ola";  
        println('c = ' + c);  
  
        def x = 0..5;  
        println('x = ' + x);  
        println('x[2] = ' + x.get(2));  
  
        def heterogeneous = [1, "a", true];  
        println('heterogeneous = ' + heterogeneous);  
  
    }  
  
}
```

# Classes

- Existem alguns tipos de classe, sendo eles: Normal, Aninhada, Abstrata, Interface e Traits.
- Métodos ***Getters and Setters*** são criados automaticamente e dinamicamente, porém podemos evitar a criação do setter e, para isso, devemos declarar um campo como ***final***.

# Classe Normal

- São definidas como top level e são concretas.
- Podem ser instanciadas sem restrições por classes e scripts.
- São publicas por padrão.
- São instancias ao serem chamadas pelo seu construtor.

```
class Person {  
    String name  
    Integer age  
  
    def increaseAge(Integer years) {  
        this.age += years  
    }  
}  
  
def p = new Person()
```

# Classe Aninhada

- Definidas dentro de outra classe.
- Classe mais externa acessa classe interna normalmente.
- Classe interna acessa dados da classe externa mesmo que sejam privados.
- Outras classes, com exceção da classe externa, não têm acesso à classe interna.

```
class Outer {  
    private String privateStr  
  
    def callInnerMethod() {  
        new Inner().methodA()  
    }  
  
    class Inner {  
        def methodA() {  
            println "${privateStr}."  
        }  
    }  
}
```

# Classe Abstrata

- Representam conceitos genéricos de classes.
- Não pode ser instanciada.
- Incluem campos/propriedades e métodos abstratos ou concretos.
- Métodos abstratos devem ser implementados pelas subclasses.

```
abstract class Abstract {  
    String name  
  
    abstract def abstractMethod()  
  
    def concreteMethod() {  
        println 'concrete'  
    }  
}
```

# Interface

- Interface define um “contrato” em que a classe precisa seguir.
- Define apenas uma lista de métodos (assinatura) que precisam ser implementados.
- Métodos precisam ser sempre públicos.
- Diferença para classes abstratas:
  - Classes abstratas podem conter propriedades e/ou métodos concretos, enquanto interfaces contém apenas as assinaturas dos métodos.

```
interface Greeter {  
    void greet(String name)  
}  
  
class SystemGreeter implements Greeter {  
    void greet(String name) {  
        println "Hello $name"  
    }  
}
```



# Traits

- **Traits** são estruturas da linguagem que possibilitam composição de comportamentos, implementação de interfaces em tempo de execução, *override* de comportamentos, compatibilidade com tipos **static** na compilação. Em resumo, são interfaces carregando, ambos, implementações default e estados.
- Podem ser usados como uma interface normal usando a keyword ***implements***.
- Podem ter métodos declarados, porem só suportam métodos que sejam ***public*** ou ***private***.

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

# Traits

- Podem ter campos declarados, podendo ser estes *public* ou *private*, que geralmente são feitos para salvar estados dos objetos.
- Podem ter propriedades.
- Podem implementar interfaces.

```
trait Greetable {  
  abstract String name()  
  String greeting () { "Hello, ${name()}!" }  
}  
  
class Person implements Greetable {  
  String name () { 'Bob' }  
}  
  
def p = new Person()  
println(p.greeting())
```

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

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

# Métodos

- Implementação similar a Java;
- Os parâmetros de métodos podem ter ou não tipos definidos e podemos colocar valores padrões a eles;
- O retorno dos métodos também podem ser arbitrários, assim como os parâmetros;

# Métodos

```
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;  
    }  
  
    static void main (String[] args) {  
        olaMetodos();  
  
        println('somaNumeros = ' + somaNumeros(1,2));  
        println('concatenaString = ' + concatenaString('Ola', ' Metodos'));  
  
        println('\n');  
  
        println('teste = ' + teste(1,2));  
        println('teste = ' + teste('Ola', ' Metodos'));  
    }  
}
```

# Métodos

- Podem ter parâmetros nomeados, e precisam receber os parâmetros como um Map.

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

- Suporte a métodos com um numero variável de parâmetros.
  - Caso o parâmetro passado seja um array, o args se transforma neste array passado.

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

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

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

- Podemos fazer override com o nome dos métodos, a partir da quantidade de argumentos definidos.

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

# Campos e Propriedades

- 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.
- As propriedades 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'
```

```
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
```

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

# Conclusão

- Groovy é uma alternativa extremamente viável para pessoas que desejam utilizar uma linguagem próxima a Java.
- Possui grande potencial para ultrapassar o uso de várias linguagens devido sua versatilidade e ferramentas disponíveis.
- Groovy é mais otimizado, possui sintaxe mais simples e possui integração maior com novos frameworks, como seu irmão Grails.

# Referências

- [1] Groovy-Lang
  - [groovy-lang.org](http://groovy-lang.org)
- [2] Groovy-Lang Documentation
  - [docs.groovy-lang.org/docs/groovy-2.5.3/html/documentation/](http://docs.groovy-lang.org/docs/groovy-2.5.3/html/documentation/)
- [3] Wikipedia: Groovy
  - [pt.wikipedia.org/wiki/Groovy](http://pt.wikipedia.org/wiki/Groovy)
- [4] DevMedia: Artigo sobre Groovy
  - [devmedia.com.br/artigo-java-magazine-69-um-pouco-de-groovy/12874](http://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](http://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](https://youtube.com/watch?v=vDtENU-3Lwo)
- [7] HostGator: O que e groovy
  - [hostgator.com.br/blog/o-que-e-groovy/](http://hostgator.com.br/blog/o-que-e-groovy/)