





Tema:

Diferenças do Groovy com outras linguagens

Grupo

Marco Olimpio - TCE/RN

Rebecca Betwel - TCE/RN



Introdução

Groovy é baseado em Java, para desenvolvedores Java a curva de aprendizagem é bem curta, precisando apenas entender suas diferenças.



Linguagem de 2003 e a última versão estável é de 14 dias atrás na versão 2.5.2

Multiparadigma: Orientada a objetos, Imperativa e scripting

Typing discipline: Dynamic, static, strong, duck

Busca ser mais conciso do que Java



1. Default imports

All these packages and classes are imported by default, i.e. you do not have to use an explicit `import` statement to use them:

- `java.io.*`
- `java.lang.*`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.net.*`
- `java.util.*`
- `groovy.lang.*`
- `groovy.util.*`

2. Multi-methods

In Groovy, the methods which will be invoked are chosen at runtime. This is called runtime dispatch or multi-methods. It means that the method will be chosen based on the types of the arguments at runtime. In Java, this is the opposite: methods are chosen at compile time, based on the declared types.

The following code, written as Java code, can be compiled in both Java and Groovy, but it will behave differently:

```
int method(String arg) {  
    return 1;  
}  
int method(Object arg) {  
    return 2;  
}  
Object o = "Object";  
int result = method(o);
```



In Java, you would have:

Static Information
`assertEquals(2, result);`

Whereas in Groovy:

RunTime Information
`assertEquals(1, result);`

Também possuí outra forma de declarar variáveis...

```
void myMethod() {  
    def arrayList = new ArrayList<String>()  
    def intValue = 5  
    def sValue = '123'  
    def otherString = "123"  
    def bashLike = "$sValue = $otherString" //"123 = 123"  
}
```

Sem necessidade de declaração do tipo da variável

Strings podem utilizar ' ou "

Strings podem usar bash-style

```
void myMethod() {  
    List<String> arrayList = new ArrayList<>();  
    int intValue = 5;  
    String sValue = "123";  
    String otherString = "123";  
}
```



3. Array initializers

In Groovy, the `{ ... }` block is reserved for closures. That means that you cannot create array literals with this syntax:

```
int[] array = { 1, 2, 3 }
```

You actually have to use:

```
int[] array = [1,2,3]
```


4. Package scope visibility

In Groovy, omitting a modifier on a field doesn't result in a package-private field like in Java:

```
class Person {  
    String name  
}
```

Instead, it is used to create a *property*, that is to say a *private field*, an associated *getter* and an associated *setter*.

It is possible to create a package-private field by annotating it with `@PackageScope`:

```
class Person {  
    @PackageScope String name  
}
```

```
package examples  
  
class MyClass {  
}
```

```
package examples;  
  
public class MyJavaClass {  
}
```

```
public class MyJavaClass {  
    public void methodOne() {  
        //....  
    }  
  
    public String methodTwo() {  
        return "abc";  
    }  
  
    public Object methodThree() {  
        return "xyz";  
    }  
}
```

‘Public’ por padrão

Retorno é o último statement executado


Retorno do tipo ‘def’

```
class MyClass {  
    void methodOne() {  
        //....  
    }  
  
    String methodTwo() {  
        "abc" //return value  
    }  
  
    def methodThree() {  
        "xyz" //returns string  
    }  
}
```

```
class MyClass {  
    int intProperty  
     def anotherIntProperty = 5  
    def stringProperty = "some string value"
```

Modificador de acesso 'public' por padrão

Variável 'def'

```
public class MyJavaClass {  
    private int intProperty;  
  
    public int getIntProperty() {  
        return intProperty;  
    }  
  
     public void setIntProperty(int intProperty) {  
        this.intProperty = intProperty;  
    }
```



```
@TupleConstructor  
@EqualsAndHashCode  
@ToString
```

```
class MyClass {  
    String one  
    def two = 2
```

- Get e Set auto gerados
- Para conseguir isso em java existe o Project Lombok

```
public class MyJavaClass {  
    private String one;  
    private int two = 2;  
  
    public MyJavaClass(String one, int two) {  
        this.one = one;  
        this.two = two;  
    }  
  
    public String getOne() {  
        return one;  
    }  
  
    public void setOne(String one) {  
        this.one = one;  
    }  
  
    public int getTwo() {  
        return two;  
    }  
  
    public void setTwo(int two) {  
        this.two = two;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof MyJavaClass)) return false;  
        MyJavaClass that = (MyJavaClass) o;  
        return Objects.equals(two, that.two) &&  
            Objects.equals(one, that.one);  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hashCode(one, two);  
    }  
  
    @Override  
    public String toString() {  
        return Objects.toStringHelper(this)  
            .add("one", one)  
            .add("two", two)  
            .toString();  
    }  
}
```

```
public class SimpleJavaBean {  
    private int a;  
    private String b;  
    private boolean c;  
  
    public SimpleJavaBean() {  
    }  
  
    public void setA(int a) {  
        this.a = a;  
    }  
  
    public void setB(String b) {  
        this.b = b;  
    }  
  
    public void setC(boolean c) {  
        this.c = c;  
    }  
}
```

```
SimpleJavaBean bean = new SimpleJavaBean();  
bean.setA(123);  
bean.setB("abc");  
bean.setC(true);
```



```
def bean = new SimpleJavaBean(a: 123, b: "abc", c: true)
```

- Acesso a valores em mapas e propriedades são similares
- Utilização pela propriedade ou index
- ada (Python)

```
def x = new MyJavaClass()  
def strValue = x.one  
def strValue2 = x["one"]  
assert strValue.equals(strValue2)
```

```
def map = [one: "value one", two: 2]  
strValue = map.one  
strValue2 = map["one"]  
assert strValue.equals(strValue2)
```



5. ARM blocks

ARM (Automatic Resource Management) block from Java 7 are not supported in Groovy. Instead, Groovy provides various methods relying on closures, which have the same effect while being more idiomatic. For example:

```
Path file = Paths.get("/path/to/file");
Charset charset = Charset.forName("UTF-8");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```




5. ARM blocks

ARM (Automatic Resource Management) block from Java 7 are not supported in Groovy. Instead, Groovy provides various methods relying on closures, which have the same effect while being more idiomatic. For example:

can be written like this:

```
new File('/path/to/file').eachLine('UTF-8') {  
    println it  
}
```

or, if you want a version closer to Java:

```
new File('/path/to/file').withReader('UTF-8') { reader ->  
    reader.eachLine {  
        println it  
    }  
}
```



6 - Inner Class

The implementation of anonymous inner classes and nested classes follows the Java lead, but you should not take out the Java Language Spec and keep shaking the head about things that are different. The implementation done looks much like what we do for `groovy.lang.Closure`, with some benefits and some differences. Accessing private fields and methods for example can become a problem, but on the other hand local variables don't have to be final.



6.1. Static inner classes

Here's an example of static inner class:

```
class A {  
    static class B {}  
}
```

```
new A.B()
```

The usage of static inner classes is the best supported one. If you absolutely need an inner class, you should make it a static one



7. Lambdas

Java 8 supports lambdas and method references:

```
Runnable run = () -> System.out.println("Run");  
list.forEach(System.out::println);
```

Java 8 lambdas can be more or less considered as anonymous inner classes. Groovy doesn't closures instead:

```
Runnable run = { println 'run' }  
list.each { println it } // or list.each(this.&println)
```



8 GStrings

Represents a String which contains embedded values such as "hello there \${user} how are you?" which can be evaluated lazily. Advanced users can iterate over the text and values to perform special processing, such as for performing SQL operations, the values can be substituted for ? and the actual value objects can be bound to a JDBC statement.

The problems is when you have a '\$' in your text like "Eu tenho R\$ 10,00"

As double-quoted string literals are interpreted as **GString** values, Groovy may fail with compile error or produce subtly different code if a class with **String** literal containing a dollar character is compiled with Groovy and Java compiler.



```
String normalString = "hello world"
def defString = "same as above"
def singleQuote = 'you can use "single quotes" like in JS'
def bashString = "Message of the day is ${normalString} or $defString"
def multiLine = """
You can use triple-double-quotes for multiline
bash style strings
$singleQuote
include closures also ${ -> System.currentTimeMillis()}
"""
```

Suporte a ' , " e """

9. String and Character literals



Singly-quoted literals in Groovy are used for `String`, and double-quoted result in `String` or `GString`, depending whether there is interpolation in the literal.

```
assert 'c'.getClass()==String
assert "c".getClass()==String
assert "c${1}".getClass() in GString
```

Groovy will automatically cast a single-character `String` to `char` only when assigning to a variable of type `char`. When calling methods with arguments of type `char` we need to either cast explicitly or make sure the value has been cast in advance.

```
char a='a'
assert Character.digit(a, 16)==10 : 'But Groovy does boxing'
assert Character.digit((char) 'a', 16)==10

try {
    assert Character.digit('a', 16)==10
    assert false: 'Need explicit cast'
} catch(MissingMethodException e) {
}
```



Groovy supports two styles of casting and in the case of casting to `char` there are subtle differences when casting a multi-char strings. The Groovy style cast is more lenient and will take the first character, while the C-style cast will fail with exception.

```
// for single char strings, both are the same
assert ((char) "c").class==Character
assert ("c" as char).class==Character

// for multi char strings they are not
try {
    ((char) 'cx') == 'c'
    assert false: 'will fail - not castable'
} catch(GroovyCastException e) {
}

assert ('cx' as char) == 'c'
assert 'cx'.asType(char) == 'c'
```


10. Primitives and wrappers

Because Groovy uses Objects for everything, it autowraps references to primitives. Because of this, it does not follow Java's behavior of widening taking priority over boxing. Here's an example using `int`

```
int i  
m(i)
```

```
void m(long l) {  
    println "in m(long)"  
}
```

1

```
void m(Integer i) {  
    println "in m(Integer)"  
}
```

2

- 1 This is the method that Java would call, since widening has precedence over unboxing.
- 2 This is the method Groovy actually calls, since all primitive references use their wrapper class.



11. Behaviour of `==`

In Java `==` means equality of primitive types or identity for objects. In Groovy `==` translates to `a.compareTo(b)==0`, if they are `Comparable`, and `a.equals(b)` otherwise. To check for identity, there is `is`. E.g. `a.is(b)`.



13. Extra keywords

There are a few more keywords in Groovy than in Java. Don't use them for variable names etc.

- `as`
- `def`
- `in`
- `trait`



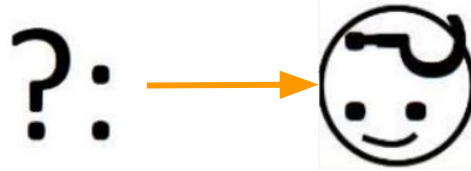
Maneiras de atribuição

```
int a = 1;  
String b = "abc";  
boolean c = true;
```

OU

```
def (a, b, c) = [1, "abc", true]
```

Operador Elvis



```
public String elvis(String param) {  
    return (param != null && param.length() > 0) ? param : "default value";  
}
```

```
def elvis(String param) {  
    param ?: "default value"  
}
```

Deferência segura

```
def upperCaseParent(File f) {  
    f?.parentFile?.path?.toUpperCase()  
}
```

É nulo? Se sim return null

Retorna NULL se algum dos objetos na cadeia for NULL sem disparar `NullPointerException`



Duck Typing

```
class Duck {  
    quack() { println "I am a Duck" }  
}  
  
class Frog {  
    quack() { println "I am a Frog" }  
}  
  
quackers = [ new Duck(), new Frog() ]  
for (q in quackers) {  
    q.quack()  
}
```

Output:

```
I am a Duck  
I am a Frog
```

Observe que realizar a criação de uma lista não é necessário que Duck e Frog herdem de uma classe comum, como é feito em C++ e em Java.



GROOVY-BASED

Apache Groovy is a language for the Java platform designed to enhance developers' productivity. It is an optionally-typed and dynamic language but with static-typing and static compilation capabilities.

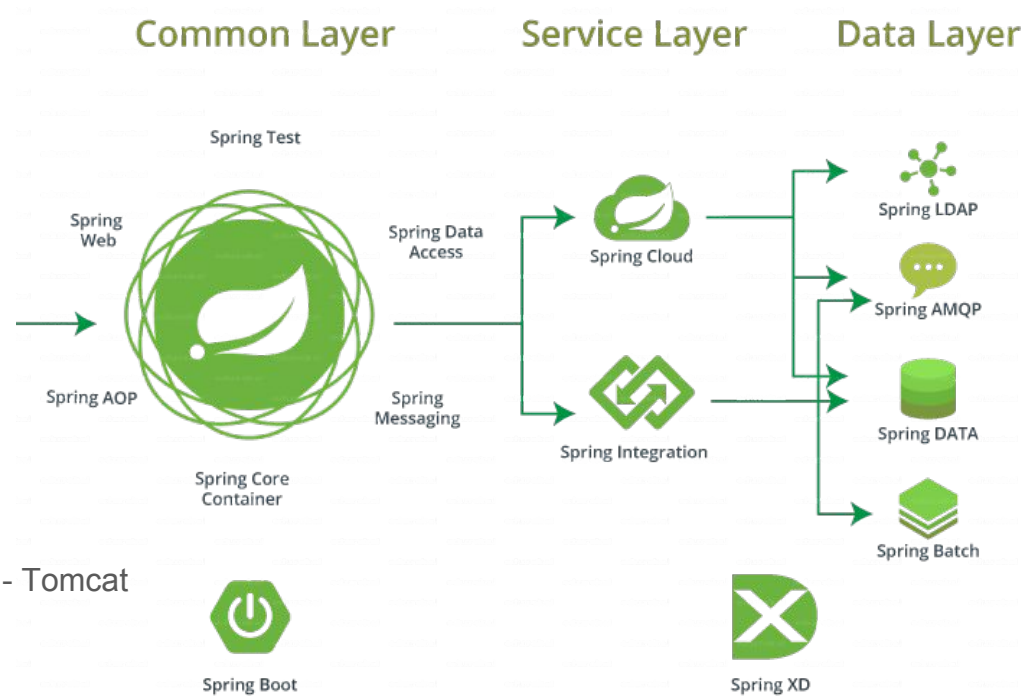


SMOOTH JAVA INTEGRATION

Grails seamlessly and transparently integrates and interoperates with Java, the JVM, and existing Java EE containers.



- Embedded Servlet Container - Tomcat
- Database Manager - H2
- Scaffolding
- Gerência de Pacotes - Gant
- Testes com JUnit e Spock
- Utilização do Groovy como linguagem de programação
- Entre várias outras facilidade





grooscript

library to convert groovy code to javascript

Why?

Groovy and java tools to create js apps

Gradle, Grails, Spock, Codenarc, ...

@Ast's, Dsl's, scripts, traits, beans, ...

Enjoy js libs from a new point of view

Type checking if you need it

Amazing interoperability with js code

Just Groovy, use js when you want to

One environment, don't repeat yourself

Create Require.js modules from groovy code

Very fast javascript code generated



Referências

<https://allegro.tech/2015/05/Reasons-to-learn-groovy.html> -BOM

<http://groovy-lang.org/differences.html> - The Apache Groovy programming Language