Groovy Cheatsheet

Sylvain Leroy

September 12, 2022

Contents

1 Démarrer avec Groovy 1

2 Les particularités du language 2

3 La syntaxe Groovy 4
- 3.1 Les commentaires . . . 4
- 3.2 Shebang et script . . . 4
- 3.3 Les identifiants . . . 4
- 3.4 Les identifiants complexes . . . 4
- 3.5 Les chaînes de caractères . . . 4
- 3.6 Les nombres . . . 4
- 3.7 Les opérateurs . . . 5
- 3.8 Les expressions régulières . . . 5

4 Les structures de données 6

5 Les collections et les maps 7
- 5.1 Les closures . . . 8

6 Le Groovy Development Kit (GDK 9
- 6.1 Manipulation de fichiers . . . 9
- 6.2 Manipulation des Threads . . . 10
- 6.3 XML . . . 10
- 6.4 JSON . . . 10
- 6.5 SQL . . . 11

7 MetaProgramming à l'exécution 12

1 Démarrer avec Groovy

A propos de Groovy

Groovy est un langage dynamique pour la Machine Virtuelle Java (JVM). Le langage offre des fonctionnalités inédites en termes de programmation objet, fonctionnelle, de scriptabilité, de typage optionnel, de customisation d'opérateurs et de concepts avancés comme les closures et la meta-programmation

Utiliser Groovy

Installer Groovy à partir de `https://groovy.apache.org/download.html`

Commande	Usage
groovy	Exécute du code groovy
groovyc	Compile du code groovy
groovysh	Ouvre le shell Groovy
groovyConsole	Ouvre la console graphique
java2groovy	outil de conversion

La commande `groovy` vient avec l'option `-h` pour obtenir la liste des options et des commandes.

Executer un fichier Script.groovy

1. `groovy Script.groovy`

Evaluer depuis la ligne de commande

1. `groovy -e "print 12.5*Math.PI"`

Compiler du code Groovy

1. `groovyc MyClasse.groovy`
2. `java -cp .  MyClasse`

Développer avec Groovy

Il y a au moins trois IDE qui supportent correctement Groovy :

1. vscode avec l'extension Groovy `https://code.visualstudio.com/`
2. IntelliJ (community or ultimate)
3. Eclipse avec le plugin Groovy (attention toutefois aux versions supportées par le plugin)

Compatiblité Java / Groovy

Depuis Groovy, il est possible d'appeler n'importe quel code Java comme vous le feriez normalement. Il est important que la version de Groovy soit supérieure ou égale à 3.0 pour les fonctionnalités de Java 1.8+
Depuis Java, il est possible d'appeler Groovy de plusieurs façons. Il vous sera nécessaire d'avoir les dépendances Groovy dans votre classpath.

Cross-compilation

1. En utilisant `groovyc`, Maven ou la tâche ANT, vous pouvez compiler votre code Groovy et l'intégrer dans votre application Java. C'est pratique si vous souhaitez simplifier l'écriture de votre application grâce à la syntaxe Groovy (scripts, tests)

Eval

1. Utiliser la classe `groovy.util.Eval` pour évaluer un code simple qui est capturé dans une simple chaîne de caractères:
2. `Eval.xyz(1,2,3,"x+y+z");`
3. `Eval.me("1+2+3");`

GroovyShell

1. Utiliser `groovy.util.GroovyShell` pour plus de flexibilité via l'objet **Binding**.

Utilisation de GroovyShell

```
1 GroovyShell shell= new GroovyShell();
2 Script scpt = shell.parse("y = x*x");
3 Binding binding = new Binding();
4 scpt.setBinding(binding);
5 binding.setVariable("x", 2);
6 scpt.run();
7 (int) binding.getVariable("y");
```

Résumé de l'intégration Groovy/Java

Option d'intégration	Caractéristiques
Eval/GroovyShell	pour de petites expressions
GroovyScriptEngine	pour l'utilisation de scripts dépendant de l'environment Java
GroovyClassLoader	la solution pour toutes les situations, bas niveau.
Spring Beans	Intégration avec Spring
JSR-223	API neutre fourni depuis Java 6. Ne gère pas le rechargement de scripts

2 Les particularités du language

La notion de Script en Groovy

Si un fichier Groovy ne déclare pas de classe publique, alors ce fichier est considéré comme un `script`.
Ainsi tout code en dehors d'une déclaration d'une classe sera considéré comme un script :

Utilisation de GroovyShell

```
println "Exemple de script"
```

Les **scripts** diffèrent des classes car ils sont instancés avec un objet Binding qui joue le rôle de containers de valeurs non déclarées et que ces fichiers sont directement exécutables avec la commande `groovy`.
Les arguemnts passés à la commande groovy sont stockées dans la variable **args** de Binding.

Exemple de script et utilisation du Binding

```
println text // la valeur est attendue dans
  Binding
result = 1 // la valeur est stockée dans Binding
```

Transcompilation d'un script

```
import org.codehaus.groovy.runtime.InvokerHelper
class Main extends Script {
    def run() {
        println 'Groovy world!'
    }
    static void main(String[] args) {
        InvokerHelper.runScript(Main, args)
    }
}
```

Les caractéristiques de Groovy

Typage optionnel

Comem d'autres langages comme Javascript, il est possible de ne pas spécifier les types des variables et des objets manipulés. Le typage statique comme Java est posssible d'être utilisé mais est optionnel.
Groovy a la compilation détermine et évalue si possible le type réel des variables (@CompileStatic)
Le typage dynamique doit être utilisé avec le mot clé `def`.
Les paramètres des méthodes et les déclarations de closure peuvent omettre le mot clé `def`.

Les propriétés

Les propriétés sont déclarées comme des champs dont la visiblité est privée. Le compilateur Groovy génère et utilise un getter et un setter pour chacun de ces champs.

Déclaration de propriétés

```
class MaClass {
String stringProp
def dynamicProp
}
```

Pour utiliser une propriété, vous pouvez la référencer comme un champ, Groovy s'occupe d'appeler le getter ou le setter respectivement.

Utilisation de propriété

```
println obj.stringProp // getter
obj.dynamicProp = 1 // setter
```

Par conséquent, lors de l'utilisation d'un champ en Groovy, l'appel au getter ou au setter est effectué (même sur les classes Java).

Multi méthodes

L'appel de méthode ne fonctionne pas pareil en **Groovy** qu'en Java. Pour résoudre l'appel d'une méthode, le type de la valeur **à l'exécution** est résolu alors qu'en Java, le type déclaré dans le code source est utilisé :

Résolution des appels

```
class Personne {
String nom
boolean equals(Personne autre) {
nom == autre.nom
}
}
assert new Personne(nom:'x') == new
  Personne(nom:'x')
assert new Personne(nom:'x') != 1
```

Le style dans Groovy 1/4

Typage optionnel

Les types peuvent être omis dans vos déclarations en utilisant soit `def` ou `var` (JDK11+).

Point-virgule optionnel

En Java, les point-virgules sont obligatoires, en Groovy ils sont optionnels, vous pouvez les enlever.

Mot-clé return est optionnel

En Groovy, la dernière expression évaluée dans le corps d'une méthode peut être retournée sans l'utilisation du mot-clé return. C'est en particulier pratique pour les méthodes courtes et les closures.

Omission de return dans les méthodes

```
String toString() { return "a server" }
String toString() { "a server" }
```

Mais il n'est pas recommandé d'omettre systématiquement les **return** sous peine d'avoir des codes un peu étrange comme :

Omission de return dans les closures

```
def props() {
    def m1 = [a: 1, b: 2]
    m2 = m1.findAll { k, v -> v % 2 == 0 }
    m2.c = 3
    m2
}
```

Méthodes définies avec def et valeur de retour

Attention une méthode définie par def accepte implicitement une valeur de retour. Aussi, si vous souhaitez protéger votre code, déclarez systématiquement les méthodes dont le type de retour est void. Sinon des valeurs peuvent être retournées accidentellement.

Omettre return dans les structures de contrôle

Les instructions comme `if/else`, `try/catch` peuvent aussi retourne une valeur, aussi la dernière expression évaluée sera retournée :

Valeur de retour

```
def foo(n) {
    if(n == 1) {
        "Roshan"
    } else {
        "Dawrani"
    }
}

assert foo(1) == "Roshan"
assert foo(2) == "Dawrani"
```

Le style dans Groovy 2/4

Public par defaut

Par défault, Groovy considère les classes et méthodes comme publiques. Par conséquent, vous n'avez pas besoin d'utiliser le modifier public. Les modificateurs de visiblités doivent être utilisés pour restreindre l'accès aux champs ou aux méthodes.
A la place de :

Avec utilisation de public

```
1 public class Server {
2     public String toString() { return "un
      serveur" }
3 }
```

Préférez la forme plus concise :

Sans utilisation de public

```
1 class Server {
2     String toString() { "un serveur" }
3 }
```

Utilisation de la visibilité "package-scope"

L'absence de modificateurs en Groovy ne signifie pas comme en Java que le champ est visible dans le package. Actuellement, il existe une annotation spéciale Groovy pour utiliser une telle visibilité :

Déclaration de champs package-scope

```
1 class Server {
2     @PackageScope Cluster cluster
3 }
```

Le style dans Groovy 3/4

Omission des parenthèses

Groovy permet d'omettre les parenthèses sur les expressions de premier niveau comme pour la commande **println**:

Déclaration de champs package-scope

```
1 println "Hello"
2 method a, b
```

avant :

Avec parenthèses

```
1 println("Hello")
2 method(a, b)
```

Quand une **closure** est le dernier paramètre d'un appel de méthodes, comme l'utilisation de la méthode **each**, alors plutôt que de construire la closure au sein des paramètres de l'appel, il est possible d'ignorer les paramètres et d'écrire la closure en dehors de l'appel :

Appel avec closure

```
1 list.each( { println it } )
2 list.each(){ println it }
3 list.each  { println it }
```

Il est recommandé d'utiliser la troisième forme qui est plus naturelle qu'une paire de parenthèses vides.

Quand les parenthèses sont obligatoires ?

Dans certains cas, comme les appels de méthodes chaînés ou les expressions trop complexes entre les parenthèses ou les appels de méthodes sans paramètres.

Appel avec closure

```
1 def foo(n) { n }
2 def bar() { 1 }
3 println foo 1 // ne fonctionne pas
4 def m = bar   // ne fonctionne pas
```

Le style dans Groovy 4/4

Classes comme citoyens de première classe

Le suffix **.class** n 'est pas nécessaire en Groovy.
Par exemple:

Code java classique

```
1 connection.doPost(BASE_URI + "/modify.hqu",
  params, ResourcesResponse.class)
```

En utilisant les GStrings :

Référence directe des classes

```
1 connection.doPost("^\${BASE_URI}/modify.hqu",
  params, ResourcesResponse)
```

3 La syntaxe Groovy

3.1 Les commentaires

Les différents commentaires

```groovy
// a standalone single line comment

/* a standalone multiline comment
   spanning two lines */

/**
 * Creates a greeting method for a certain person.
 *
 * @param otherPerson the person to greet
 * @return a greeting message
 */

 /**@
 * Some class groovydoc for Foo
 */

 assert Foo.class.groovydoc.content.contains('Some class
 groovydoc for Foo')
```

3.2 Shebang et script

```groovy
#!/usr/bin/env groovy
println "Hello from the shebang line"
```

3.3 Les identifiants

```groovy
def name
def item3
def with_underscore
def $dollarStart
```

3.4 Les identifiants complexes

```groovy
def map = [:]
map.'single quote'
map."double quote"
map.'''triple single quote'''
map."""triple double quote"""
map./slashy string/
map.$/dollar slashy string/$
```

3.5 Les chaînes de caractères

Types de chaînes

```groovy
'chaine entre apostrophes'

// Concaténation
assert 'ab' == 'a' + 'b'

'''Cette
chaîne est
convertie en String
non pas en GString'''

"GString entre apostrophes"
```

Interpolation

```groovy
//Interpolation
def name = 'Guillaume' // a plain string
def greeting = "Bonjour ${name}"
assert greeting.toString() == 'Bonjour Guillaume'


def person = [name: 'Guillaume', age: 36]
assert "$person.name has $person.age ans" ==
  'Guillaume a 36 ans'
```

Attention

Si la propriété référencée dans la chaîne de caractères n'existe pas, une Exception `groovy.lang.MissingPropertyException` est lancée.

Chaînes de caractères passives

```groovy
def number = 1
def statiqueGString = "value == ${number}"
def passiveGString = "value == ${ -> number }"

assert statiqueGString == "value == 1"
assert passiveGString ==  "value == 1"

number = 2
assert statiqueGString == "value == 1"
assert passiveGString ==  "value == 2"
```

GString multi-lignes

Utilisation comme template

```groovy
def name = 'Groovy'
def template = """
    Cher Mr ${name},

    Vous êtes le gagnant de la lotterie.

    Cordialement,

    Dave
"""

assert template.toString().contains('Groovy')
```

Résumé

Syntaxe	Interpolation	Multiline	Echappement
'...'			\
'''...'''		√	\
"..."	√		\
""".."""	√	√	\
/.../	√	√	\
/.../	√	√	$

3.6 Les nombres

Les nombres

Tous les nombres Groovy sont des objets, et non pas des types primitifs. Toutes les déclarations de litéraux sont:

Type	Exemples
java.lang.Integer	15, 0x1234ffff
java.lang.Long	100L, 100l
java.lang.Float	1.23f, 4.56F
java.lang.Double	1.23d, 4.56D
java.math.BigInteger	123g, 456G
java.math.BigDecimal	1.23, 4.56, 1.4E4, 2.8e4, 1.23g, 1.23G

Les types de données simples

Voici quelques exemples mathématiques et les types produits.

Exemple	Type produit
`1f * 2f`	Double
`1f / 2f`	Double
`(Byte)1 + (Byte)2`	Integer
`1 * 2L`	Long
`1 / 2`	BigDecimal (0.5)
`(int)(1/2)`	Integer (0)
`1.intdiv(2)`	Integer (0)
`Integer.MAX_VALUE+1`	Integer
`2**31`	Integer
`2**33`	Long
`2**3.5`	Double
`2G + 1G`	BigInteger
`2.5G + 1G`	BigDecimal
`1.5G == 1.5F`	Boolean (true)
`1.1G == 1.1F`	Boolean (false)

3.7 Les opérateurs

Les opérateurs

Surcharge d'opérateurs

Tous les opérateurs Groovy peuvent être surchargés en implémentant une méthode spécifique dans votre classe.

Opérateur	Méthode
`a + b`	a.plus(b)
`a - b`	a.minus(b)
`a * b`	a.multiply(b)
`a / b`	a.div(b)
`a % b`	a.mod(b)
`a++, ++a`	a.next()

Les opérateurs spéciaux

Opérateur	Usage	Nom (anglais)
a ? b : c	if (a) b else c	ternary if
a ?: b	a ? a : b	Elvis
a?.b	a==null ? a : a.b	null safe
a*.list)	a(list[0], list[1], ...)	spread
list*.a()	[list[0].a(), list[1].a() ...]	spread-dot
a.&b	référence à la méthode b dans l'objet a comme une closure	method closure
a.@field	accès direct à un champ	dot-at

Les autres opérateurs

Opérateur	Méthode
a-,-a	a.previous()
a**b	a.power(b)
a | b	a.or(b)
a&b	a.and(b)
a^b	a.xor(b)
~ a	~ a a.bitwiseNegate() // sometimes referred to as negate | +a a.positive() | -a a.negative()
a[b]	a.getAt(b)
a[b] = c	a.putAt(b, c)
a << b	a.leftShift(b)
a >> b	a.rightShift(b)
a >>> b	a.rightShiftUnsigned(b)
switch(a) case b:	b.isCase(a) // b is a classifier
[a].grep(b) if(a in b)	
a == b	a.equals(b)
a != b !	a.equals(b)
a <=> b	a.compareTo(b)
a > b	a.compareTo(b) > 0
a >= b	a.compareTo(b) >= 0
a < b	a.compareTo(b) < 0
a <= b	a.compareTo(b) <= 0
a as B	a.asType(B)

Exemples

```groovy
def twister = 'she sells sea shells'
// contient le mot 'she'
assert twister =~ 'she'
// débute avec le mot 'she' et termine avec le mot 'shells'
assert twister ==~ /she.*shells/
// même chose pré-compilé
def pattern = ~/she.*shells/
assert pattern.matcher(twister).matches()
// Les motifs sont itérables
// Les mots qui commencent par 'sh'
def shwords = (twister =~ /\bsh\w*/).collect{it}.join(' ')
assert shwords == 'she shells'
// Remplacement par logique
assert twister.replaceAll(/\w+/){
it.size()
} == '3 5 3 6'
// Les groupes de regexp sont manipulables par une closure.
// Trouver les mots avec le même début et fin
def matcher = (twister =~ /(\w)(\w+)\1/)
matcher.each { full, first, rest ->
assert full in ['sells','shells']
assert first == 's'
}
```

3.8 Les expressions régulières

Les expressions régulières

Les expressions régulières utilisent l'API Java mais son utilisation est simplifiée.

Les opérateurs

1. L'opérateur de recherche =~ .
2. L'opérateur de correspondance ==~ .
3. L'opérateur pour construire un Pattern ~.

Comment écrire une expression régulière

Symbole	Signification
.	n'importe quel caractère
^	début de ligne ou document
$	fin de ligne ou document
\ d	un chiffre
\ D	n'importe quel caractère sauf un chiffre
\ s	espace
\ S	n'importe quel caractère sauf les espaces
\ w	un mot
\ W	tout caractère sauf les mots
\ b	séparation de mots
()	création de groupe
|	(x|y) x ou comme en (Groovy|Java|Ruby)
x*	zero ou plusieurs occurrences de x.
x+	une ou plusieurs occurrences de x.
x?	zero ou une occurrence de x.
xm,n	au moins "m" et au plus "n" occurrences de x.
xm	exactement "m" occurrences de x. a-f>a-f
[^a]	classe de caractères contenant 'a', 'b', 'c', 'd', 'e', 'f'
(?is:x)	classe de caractères contenant tous les caractères sauf 'a' active le mode pendant l'évaluation de x; le mode i active ignoreCase, s le mode ligne unique
(?=regex)	recherche en avant
(?<=text)	recherche en arrière

4 Les structures de données

Structure d'un programme

Les structures de données disponibles

1. La déclaration de package
2. Les imports
3. Les méthodes
4. Les classes
5. Les enums
6. Les constructeurs
7. Les interfaces
8. Les définitions d'annotations
9. Les **traits**

Exemple

```groovy
package com.byoskill.example
import java.util.Properties

enum Privacy { PRIVATE, PUBLIC }
interface Greeter {
    void greet(String name)
}
trait FlyingAbility {
        String fly() { "I'm flying!" }
}
class MailConfig {
    String from
    String to
    String body
    Privacy privacy

    MailConfig(from, to, body, privacy) {
        this.from = from
        this.to = to
        this.body = body
        this.privacy = privacy
    }

    def write() {
        """
        From: ${from}
        To: ${to}
        Body:
        ${body}
        """
    }
}
```

Les constructeurs

Groovy supporte plusieurs méthodes d'invocation des constructeurs :

- les paramètres positionnels
- les paramètres nommés

paramètres positionnels

```groovy
class PersonConstructor {
    String name
    Integer age

    PersonConstructor(name, age) {
        this.name = name
        this.age = age
    }
}
// Possibilités d'appels
def person1 = new PersonConstructor('Marie', 1)
def person2 = ['Marie', 2] as PersonConstructor
PersonConstructor person3 = ['Marie', 3]
```

paramètres nommés

```groovy
class PersonWOConstructor {
    String name
    Integer age
}

def person4 = new PersonWOConstructor()
def person5 = new PersonWOConstructor(name: 'Marie')
def person6 = new PersonWOConstructor(age: 1)
def person7 = new PersonWOConstructor(name: 'Marie',
age: 2)
```

Les méthodes

Les méthodes supportent également les deux types d'invocation. Le type de retour et des paramètres est optionnel avec le keyword `def`.

- les paramètres positionnels
- les paramètres nommés les arguments par défaut

paramètres nommés

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

paramètres mélangés nommés et positionnels

```groovy
def foo(Map args, Integer number) {
"${args.name}: ${args.age}, and the number is
${number}" }
foo(name: 'Marie', age: 1, 23)
foo(23, name: 'Marie', age: 1)
```

les arguments par défaut

```groovy
def foo(String par1, Integer par2 = 1) { [name:
par1, age: par2] }
assert foo('Marie').age == 1
```

les paramètres variables

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

5 Les collections et les maps

Les types de containers

Groovy offre de nombreux types pour contenir vos données.

- Les intervalles (Ranges)
- Les listes
- Les maps

Les intervalles

Les intervalles(Ranges) sont des ensembles de valeurs continues inclusifs comme $0..10$ ou semi-exclusifs comme $0.. < 10$. Elles sont souvent encapsulées entre parenthèses puisque l'opérateur à une priorité faible.

```
assert (0..10).contains(5)
assert (0.0..10.0).containsWithinBounds(3.5)
for (item in 0..10) { println item }
for (item in 10..0) { println item }
(0..<10).each { println it }
```

Les intervalles d'entier sont souvent utilisés pour produire des sous-listes. Les intervalles peuvent être $previous()$, $next()$ et $implementsComparable$. Notable examples are String and Date.

Les maps

Maps sont comme les listes qui possèdent un type arbitraire pour les clés à la place d'un entier. La syntaxe est très proche.

```
def map = [a:0, b:1]
```

Les Maps peuvent être accédées avec des crochets ou bien en utilisant la clé comme une propriété de la map.

```
assert map['a'] == 0
assert map.b == 1
map['a'] = 'x'
map.b = 'y'
assert map == [a:'x', b:'y']
```

Il existe aussi une méthode explicite qui prend optionnellement une valeur par défaut

```
assert map.c == null
assert map.get('c',2) == 2
assert map.c == 2
```

Les méthodes d'itération d'une Map prennent en compte la nature des objets Map.Entry.

```
map.each { entry ->
println entry.key
println entry.value
}
map.each { key, value ->
println "$key $value"
}
for (entry in map) {
println "$entry.key $entry.value"
}
```

Les listes

Les listes sont comme des tableaux mais ils ont le type $java.util.List$ plus de nouvelles méthodes.

```
[1,2,3,4] == (1..4)
[1,2,3] + [1] == [1,2,3,1]
[1,2,3] << 1 == [1,2,3,1]
[1,2,3,1] - [1] == [2,3]
[1,2,3] * 2 == [1,2,3,1,2,3]
[1,[2,3]].flatten() == [1,2,3]
[1,2,3].reverse() == [3,2,1]
[1,2,3].disjoint([4,5,6]) == true
[1,2,3].intersect([4,3,1]) == [3,1]
[1,2,3].collect{ it+3 } == [4,5,6]
[1,2,3,1].unique().size() == 3
[1,2,3,1].count(1) == 2
[1,2,3,4].min() == 1
[1,2,3,4].max() == 4
[1,2,3,4].sum() == 10
[4,2,1,3].sort() == [1,2,3,4]
[4,2,1,3].findAll{it%2 == 0} == [4,2]
def anims=['cat','kangaroo','koala']
anims[2] == 'koala'
def kanims = anims[1..2]
anims.findAll{it =~ /k.*/} ==kanims
anims.find{ it =~ /k.*/} ==kanims[0]
anims.grep(~/k.*/) ==kanims
```

Trier les listes

La méthode $sort()$ est souvent utilisée et vient en trois "saveurs":

Sort call	**Usage**
col.sort()	tri naturel pour les objets comparables
col.sort {	
it.propname	applique une closure sur chaque élément avant de comparer les résultats.
}	
col.sort { a,b ->	
a <=>b	closure defines a comparator for each comparison
}	

Les listes

Index négatifs

Lists can also be indexed with negative indexes and reversed ranges.

```
def list = [0,1,2]
assert list[-1] == 2
assert list[-1..0] == list.reverse()
assert list == [list.head()] + list.tail()
Les affectations de sous-liste peuvent
faire grossir ou diminuer la liste et les
listes peuvent contenir différents types
de données.
list[1..2] = ['x','y','z']
assert list == [0,'x','y','z'
```

Les instructions GPath

Appeler une propriété sur une liste retourne la liste des propriétés pour chaque éléments de la liste.

- employes.adresse.ville
- => Retourne une liste de ville.

Il est possible de faire la même chose via des appels de méthode en utilisant l'opérateur **spread-dot**.

```
employees*.bonus(2008)
```

Appelle la méthode $bonus$ sur chaque employé et retourne une liste en résultat.

5.1 Les closures

Comment utiliser les closures

Les Closures (ou Lambdas en Java) capture un morceau de logique et la portée l'englobant. Ce sont des objets de premier niveau qui peuvent recevoir des messages et être retournées par des appels de méthodes, stockés dans des champs ou utilisés comme arguments d'appels de méthode.

Utilisation dans un paramètre de méthode.

```
def forEach(int i, Closure yield){
```

```
for (x in 1..i) yield(x)
```

Utilisation comme dernier argument d'une méthode.

```
forEach(3) { num -> println num }
```

Construction et affectation à une variable locale.

```
def squareIt = { println it * it}
forEach(3, squareIt)
```

Associer la closure la plus à gauche à un argument figé.

```
def multIt = {x, y -> println x * y}
forEach 3, multIt.curry(2)
forEach 3, multIt.curry('-')
```

Utilisation avancée de closures :

```
switch ('xy'){
case {it.startsWith('x')} :...
}
[0,1,2].grep { it%2 == 0 }
```

Closure	Paramètres
{ ... }	aucune ou un ('it' implicite)
{->... }	zero
{x ->... }	un
{x=1 ->... }	un ou zero avec valeur défaut
{x,y ->... }	deux
{ String x ->... }	un avec type statique

6 Le Groovy Development Kit (GDK

Amélioration de la classe java.lang.Object

Groovy ajoute de nombreuses méthodes à la classe `java.lang.Object`.
Get object info

```
println obj.dump()
```

or in a GUI

```
import groovy.inspect.swingui.*
ObjectBrowser.inspect(obj)
```

Afficher les propriétés, méthodes et champs de l'objet.

```
println obj.properties
println obj.class.methods.name
println obj.class.fields.name
```

Deux méthodes pour appeler une méthode dynamiquement

```
obj.invokeMethod(name, paramsAry)
obj."$name"(params)
```

Autres méthodes fournies

```
is(other) // vérification de l'identité
isCase(candidate) //default:equality
obj.identity {...};
obj.with {...}
print(); print(value),
println(); println(value)
printf(formatStr, value)
printf(formatStr, value[])
sleep(millis)
sleep(millis) { onInterrupt }
use(categoryClass) { ... }
use(categoryClassList) { ... }
```

Méthodes d'itération

Groovy marque tous les objects comme *Iterable*.
Il est donc possible d'utiliser n'importe quel objet dans une boucle comme
for (element in obj) { ... }

Important

Implémentez la méthode iterator() afin de retourne un objet Iterator et ainsi donner une vraie utilité à itérer sur les objets de votre classe.
Il est possible d'utiliser les méthodes suivantes .

Valeur de retour	Usage
Boolean	any {...}
List	collect {...}
Collection	collect(Collection collection) {...}
(void)	each {...}
(void)	eachWithIndex {item, index->...}
Boolean	every {...}
Object	find {...}
List	findAll {...}
Integer	findIndexOf {...}
Integer	findIndexOf(startIndex) {...}
Integer	findLastIndexOf {...}
Integer	findLastIndexOf(startIndex) {...}
List	findIndexValues {...}
List	findIndexValues(startIndex) {...}
Object	inject(startValue) {temp, item ->...}
List	grep(Object classifier)

6.1 Manipulation de fichiers

Les fichiers et les I/O

Méthodes fréquemment utilisées :

```
def dir = new File('somedir')
def cl = {File f -> println f}
dir.eachDir cl
dir.eachFile cl
dir.eachDirRecurse cl
dir.eachFileRecurse cl
dir.eachDirMatch(~/.*/, cl)
dir.eachFileMatch(~/.*/, cl)
```

Méthodes utiles pour la lecture

```
def file = new File('/data.txt')
println file.text
(also for Reader, URL, InputStream,Process)
def listOfLines = file.readLines()
file.eachLine { line -> ... }
file.splitEachLine(/\s/) { list -> }
file.withReader { reader -> ... }
(also for Reader, URL, InputStream)
file.withInputStream { is -> ...}
(also for URL
```

Méthodes utiles pour l'écriture

```
out << 'content'
pour out de type File, Writer, OutputStream, Socket, and Process
file.withWriter('ASCII') {writer -> }
file.withWriterAppend('ISO8859-1'){
    writer -> ...
}

// Si l'objet supporte la méthode writeTo()

writer << o
```

Méthodes utiles pour lire / écrire des String

```
def out = new StringWriter()
out << 'something'
def str = out.toString()
def rdr = new StringReader(str)
println rdr.readLines()
```

Connecter Reader et Writers

```
writer << reader
```

Transformer un flux et filter son contenu

```
reader.transformChar(writer){c -> }
reader.transformLine(writer){line-> }
// Pour src dans File, Reader, InputStream
src.filterLine(writer){line-> }
writer << src.filterLine {line ->
```

6.2 Manipulation des Threads

Manipulation des threads et des processus

Deux façons de démarrer un thread

```
def thread = Thread.start { ... }
def t = Thread.startDaemon { ... }
```

Deux façons de parler à un processus externe

```
today = 'cmd /c date
/t'.execute().text.split(/\D/)
proc = ['cmd','/c','date'].execute()
Thread.start {System.out << proc.in}
Thread.start {System.err << proc.err}

proc << 'no-such-date' + "\n"
proc << today.join('-') + "\n"
proc.out.close()
proc.waitForOrKill(0)
```

6.3 XML

Lecture XML

Il existe deux parseurs en fonction que vous souhaitez obtenir une structure de données ou parcourir un flux

```
def parser = new XmlParser()
def slurper = new XmlSlurper()
```

Méthodes de parsing

Les méthodes **parse()** retournent des objets différents (Node vs. GPathResult) mais il est possible d'utiliser les méthodes suivantes dessus :

```
result.name()
result.text()
result.toString()
result.parent()
result.children()
result.attributes()
result.depthFirst()
result.iterator() // see GDK hot tip
```

Exemple lire les 10 premiers titres d'un blog

```
def url= 'http://www.groovyblogs.org/feed/rss'
def rss = new XmlParser().parse(url)
rss.channel.item.title[0..9]*.text()
```

Accès aux élements du XML

['elementName'] .elementName	All child elements of that name
[index]	Child element by index
['@attributeName'] .'@attributeName' .@attributeName	The attribute value stored under that name

Ecrire du XML

Groovy (Streaming-) MarkupBuilder vous permet de produire votre propre XML tout en conservant un style déclaratif

MarkupBuilder

```
def b=new groovy.xml.MarkupBuilder()
b.outermost {
    simple()
    'with-attr' a:1, b:'x', 'content'
    10.times { count ->
        nesting { nested count }
    }
}
```

6.4 JSON

Lecture JSON

Groovy fournit une API très pratique pour lire des payloads JSON
La classe $JsonSlurper$ est une classe qui parse du texte JSON ou lit son contenu et le convertit en structures de données Groovy comme des maps, listes et types primitifs.

Lire du JSON avec Groovy

```
def jsonSlurper = new JsonSlurper()
def object = jsonSlurper.parseText('{ "name":
"John Doe" } /* some comment */')

assert object instanceof Map
assert object.name == 'John Doe'
```

Les expressions $GPath$ peuvent être utilisées sur les objets produits par $JSONSlurper$.

Alternatives

Classe	Particularité
JSonSlurper, JsonParserCharArray	Implémentation par défaut
JsonFastParser	Implémentation la plus rapide
JsonParserLax	JSON avec commentaires autorisés
JsonParserUsing CharacterSource	Implémentation pour fichiers larges

Ecrire du JSON

JsonOutput est responsable de la sérialisation d'objets Groovy en String JSON.
JsonOutput vient avec une collection de méthodes statiques **toJson**.
Le résultat de l'appel de **toJson** est une chaine de caractères contenant le code JSON.

```
def json = JsonOutput.toJson([name: 'John Doe', age:
42])

assert json == '{"name":"John Doe","age":42}'
```

Conversion d'objets

```
class Person { String name }

def json = JsonOutput.toJson([ new Person(name:
'John'), new Person(name: 'Max') ])

assert json == '[{"name":"John"},{"name":"Max"}]'
```

Conversion d'objets

Autres possibilités de génération

```groovy
def generator = new JsonGenerator.Options()
    .excludeNulls()
    .dateFormat('yyyy@MM')
    .excludeFieldsByName('age', 'password')
    .excludeFieldsByType(URL)
    .build()

assert generator.toJson(person) ==
'{"name":"John","dob":"1984@12"}'
```

Construire un JSON programmatiquement

JsonBuilder permet de créer facilement pour créer un JSON.

```groovy
JsonBuilder builder = new JsonBuilder()
builder.records {
  car {
        name 'HSV Maloo'
        make 'Holden'
        year 2006
        country 'Australia'
        record {
            type 'speed'
            description 'production pickup truck
            with speed of 271kph'
        }
  }
}
String json =
JsonOutput.prettyPrint(builder.toString())
```

6.5 SQL

Connexion à la base de données

Obtenir une instance de l'objet Sql directement. Par exemple, une base de données HSQLDB.

```groovy
import groovy.sql.Sql

def db = Sql.newInstance(
 'jdbc:hsqldb:mem:GInA',
 'user-name',
 'password',
 'org.hsqldb.jdbcDriver')

// Alternative en utilisant une source de
données

import org.hsqldb.jdbc.*

def source = new jdbcDataSource()
source.database = 'jdbc:hsqldb:mem:GInA'
source.user = 'user-name'
source.password = 'password'
def db = new groovy.sql.Sql(source)
```

Soumettre une requête

Quand une requête contient des wildcards, il est recommandé d'utiliser un **PreparedStatement**.
Groovy SQL le réalise automatiquement si vous passez la liste de valeurs dans une liste supplémentaires ou si vous utilisez une GString :

```groovy
method('SELECT ... ')
method('SELECT ...?,?', [x,y])
method("SELECT ... $x,$y")
```

Autres méthodes

Retourne	Méthode	Paramètres
boolean	execute	prepStmt
Integer	executeUpdate	prepStmt
void	eachRow	prepStmt { row ->}
void	query	prepStmt { resultSet ->... }
List	rows	prepStmt
Object	firstRow	prepStmt

Au dessus, les attributs peuevent être obtenus par index ou par nom.

```groovy
db.eachRow('SELECT a,b ...'){ row ->
println row[0] + ' ' + row.b
```

Utilisation avec GPath

```groovy
List hits = db.rows('SELECT ...')
hits.grep{it.a > 0}
```

DataSet

Pour des opérations de base de données simplifiées, sans SQL :

```groovy
def dataSet = db.dataSet(tablename)
dataSet.add (
a: 1,
b: 'something'
)
dataSet.each { println it.a }
dataSet.findAll { it.a < 2 }
```

Dans la dernière instruction, l'expression de la closure du findAll convertit directement l'expression en clause WHERE.

7 MetaProgramming à l'exécution

Amélioration de la classe java.lang.Object

Catégories

Un groupe de méthodes affectées à l'exécution à des classes arbitraitres peut satisfaire un objectif commun. Ces méthodes s'appliquent à un thread unique. La portée est limitée à une closure.

Utilisation de méthodes personnalisées

```
1 class IntCodec {
2  static String encode(Integer self){self.toString()}
3  static Integer decode(String
   self){self.toInteger()}
4 }
5 use(IntCodec) {42.encode().decode()}
```

ExpandoMetaClass

Le même exemple mais appliqué à tous les threads et dans une scope globale.

Autres méthodes fournies

```
1 Integer.metaClass.encode << {delegate.toString()}
2 String.metaClass.decode << {delegate.toInteger()}
3 42.encode().decode()
```

Hooks pour l'invocation de méthodes

Dans votre classe Groovy, implémentez la méthode pour intercepter toutes les méthodes non définies.

Metaprogramming

```
1 Object invokeMethod(String name, Object args)
```

Il est possible d'intercepter les appels à des méthodes existantes en utilisant l'interface **GroovyInterceptable**.

Intercepter les getters et setters

Surcharger les getters / setters

```
1 Object getProperty(String name)
2 void setProperty(
3 String name, Object value)
```

Une autre alternative plus simple :

Autres méthodes fournies

```
1 Object methodMissing(String name, Object args)
2 Object propertyMissing(
3 String name, Object args)
```

Enfin,

Autres méthodes fournies

```
1 Integer.metaClass.methodMissing << {
2 String name, Object args ->
3 Math."$name"(delegate)
4 }
5 println 3.sin()
6 println 3.cos()
```