





Groovy

David THIBAU – 2020

david.thibau@gmail.com



Agenda

- Présentation

- Groovy et Java
- Un bref tour de Groovy
- Installation, Exécution, Compilation IDE

- Le GDK

- Libraries cœur
- Groovy et SQL
- Json et Groovlets

- Les bases de Groovy

- Types de données
- Collections
- Orientation Objet
- Les closures

- Cas d'usage

- Les Tests
- Intégrer Groovy
- Les DSLs

- Méta-programming

- Programmation dynamique
- Transformations AST, annotations



Groovy et Java

Un bref tour de Groovy

Installation



Introduction

Groovy ressemble à Java

Objectif principal : simplifier certains éléments de syntaxe requis par Java

=> Le code devient juste plus compacte et plus facile à lire

Exemple en Java

```
java.net.URLEncoder.encode("a b", "UTF-8");
```

En Groovy:

```
URLEncoder.encode 'a b', 'UTF-8'
```



Beaucoup de ressemblances

- Les commentaires
- Les packages
- Les instructions, les structures de contrôle, les opérateurs, les expressions et affectations
- Les classes, les interfaces, les *enum*, les attributs et méthodes, les classes imbriquées.
- Le traitement des exceptions
- La déclaration des littéraux (à l'exception de l'initialisation de tableaux à cause de la signification de {} dans Groovy)
- La déclaration et l'utilisation de génériques et des annotations.
- L'instanciation d'objets, les références, l'appel de méthodes.



Mais aussi des ajouts

- L'accès simplifié aux objets à travers de nouvelles expressions et opérateurs.
- De nouvelles façons de créer des objets en utilisant des littéraux.
- De nouveaux contrôles de structure
- L'utilisation d'annotations pour générer du code
- De nouveaux types avec de nouveaux opérateurs.
- Le backslash pour continuer une instruction sur une nouvelle ligne
- Import automatique de *groovy.lang.** , *groovy.util.** , *java.lang.** , *java.util.** , *java.net.** , and *java.io.** et *java.math.BigInteger* et *BigDecimal*
- ...



Groovy est Java

Le code Groovy s'exécute dans une JVM et suit le modèle Objet de Java

Les classes groovy peuvent s'exécuter dans la JVM de 2 façons :

- On peut **pré-compiler** les fichiers Groovy en des classes Java et les positionner dans le classpath.
Les objets sont alors chargés par le classloader classique de Java
- On peut **directement** travailler avec les fichiers **.groovy* et récupérer les objets via les classloader de Groovy.
=> Dans ce cas, aucun fichier **.class* n'est généré mais les classes sont générées dynamiquement et présentes exclusivement en mémoire



GDK

La librairie **GDK** ramenée par *Groovy* est une extension du JDK :

- Elle fournit de nouvelles classes (par exemple pour faciliter l'accès BD ou le traitement de XML)
- Elle ajoute des fonctionnalités aux classes existantes du JDK.

Le *GDK* est organisé en modules



Exemple d'extension

Type	Determine the size in JDK via ...	Groovy
Array	length field	size() method
Array	java.lang.reflect.Array.getLength(array)	size() method
String	length() method	size() method
StringBuffer	length() method	size() method
Collection	size() method	size() method
Map	size() method	size() method
File	length() method	size() method
Matcher	groupCount() method	size() method



Groovy est dynamique

Avec Groovy, les méthodes appelées sont choisies lors de l'exécution.

Même si les scripts et classes Groovy sont transformés en classes Java, Groovy a un typage dynamique

- Principalement grâce au pattern **method dispatch** et la notion de **MetaClass**
- Par exemple, l'appel *foo()* à l'intérieur d'une classe Groovy est compilée en :

```
getMetaClass().invokeMethod(this, "foo", EMPTY_PARAMS_ARRAY)
```

La **méta-classe** peut alors apporter tout l'aspect dynamique de Groovy en interceptant, ajoutant ou modifiant des méthodes lors de l'exécution.



Groovy peut être statique

On peut forcer certaines parties du code à être statique en utilisant l'annotation ***@TypeChecked***.

```
class Universe {  
    @groovy.transform.TypeChecked  
    int answer() { "forty two" }  
}
```

=> Erreur de compilation



Groovy et Java

Un bref tour de Groovy

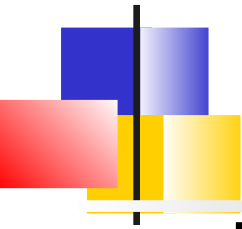
Installation



Sources

Les fichiers sources de *Groovy* sont soit :

- Des définitions de **classe(s)**
- Des **scripts** qui pourront alors être démarrés par une commande en ligne.



Déclaration de classes

La déclaration de classe est similaire à Java. Elles ont des **champs** et des **propriétés** (JavaBean) sur lesquels on peut appliquer des *modifiers* de visibilité.

Les principales différences avec Java sont :

- les classes et les méthodes sans *modifier* sont *public* par défaut
- Les champs sans *modifiers* deviennent des propriétés (accessible via getter/setter)
- Le fichier source peut ne pas avoir le même nom que la classe déclarée et peut déclarer plusieurs classes



Exemple

```
class Book {  
    // propriétés  
    String title  
    // Champ privé  
    private Integer currentPage ;  
  
    Book (String theTitle) {  
        title = theTitle  
    }  
  
    // Méthode publique  
    def tournerPage(){  
        currentPage++ ;  
    }  
}
```




Scripts

Les **scripts** sont des fichiers texte (ayant par convention l'extension **.groovy*)

Il peuvent être exécutés par :

> `groovy myfile.groovy`

- Les scripts contiennent des instructions qui ne sont pas encapsulées par une déclaration de classe.
- Ils peuvent contenir des définitions de méthodes à l'extérieur d'une définition de classe
- Un script est parsé intégralement avant son exécution



Exemple de script

Book.groovy est dans le classpath

```
Book gina = new Book('Groovy in Action')  
assert gina.getTitle() == 'Groovy in Action'
```

Appel de méthode avant sa déclaration

```
assert getTitleBackwards(gina) == 'noitcA ni yvoorG'  
String getTitleBackwards(book) {  
    String title = book.getTitle()  
    return title.reverse()  
}
```



Annotations

Groovy permet d'utiliser et de définir des **annotations** comme Java

- Exemple : *@Immutable* qui est sensiblement équivalent à *final*

Les annotations *Groovy* instruisent le compilateur afin que le compilateur effectue une transformation AST (*abstract syntax tree*), i.e. ajout/suppression de méthodes, modification de la structure de code.

Il est possible de définir ses propres transformations (*meta-programming* à la compilation)



@Grab

L'annotation **@Grab** est utilisée pour définir explicitement les dépendances vers des librairies externes.

A la compilation et l'exécution, la librairie est téléchargée si besoin (*.groovy/grape*) et ajoutée au classpath.

```
@Grab('commons-lang:commons-lang:2.4')
import org.apache.commons.lang.ClassUtils
class Outer {
    class Inner {}
}
assert !ClassUtils.isInnerClass(Outer)
assert ClassUtils.isInnerClass(Outer.Inner)
```



Gstring et regexp

En *Groovy*, les littéraux String peuvent utiliser les simples, double ou triple-quotes.

La version double-quotes permet l'utilisation d'expressions qui sont résolues à l'exécution : les *GStrings*

```
def nick = 'ReGina'  
def book = 'Groovy in Action, 2nd ed.'  
assert "$nick is $book" == 'ReGina is Groovy in Action, 2nd ed.'
```

Groovy facilite également l'utilisation des expressions régulières



Pas de types primitifs

En *Groovy*, tout est objet.

- Tout nombre, booléen est converti en une objet Java.
- Les notations des types primitifs sont cependant toujours supportées :

```
def x = 1
int y = 2
assert x + y == 3
assert x.plus(y) == 3
assert x instanceof Integer
```



Collections

Groovy facilite la manipulation des collections en ajoutant

- des opérateurs,
- des instanciations via des littéraux
- et de nouvelles méthodes

Il introduit également un nouveau type :
Range



Exemples

```
// roman est une List
def roman = ['', 'I', 'II', 'III', 'IV', 'V', 'VI', 'VII']
// On accède à un élément comme un tableau Java
assert roman[4] == 'IV'
// Il n'y a pas d'ArrayIndexOutOfBoundsException
roman[8] = 'VIII'
assert roman.size() == 9
// Les maps peuvent être facilementinstanciées
def http = [
  100 : 'CONTINUE',
  200 : 'OK',
  400 : 'BAD REQUEST'
]
// Accès aux éléments avec la notation Array Java
assert http[200] == 'OK'
// Méthode put simplifiée
http[500] = 'INTERNAL SERVER ERROR'
assert http.size() == 4
```




Closures

Les ***closures*** Groovy permettent la programmation fonctionnelle.

Un bloc d'instructions peut être passé en paramètre à une méthode.

- Cependant, différent des lambda de Java8
- Un objet de type *Closure* travaille en sous-main

```
[1, 2, 3].each { entry -> println entry }
```



Structures de contrôle

```
if (false) assert false // if sur une ligne
```

```
if (null) { // null est false
```

```
    assert false
```

```
}
```

```
// Boucle while classique
```

```
def i = 0
```

```
while (i < 10) {
```

```
    i++
```

```
}
```

```
assert i == 10
```

```
// for in sur un intervalle
```

```
def clicks = 0
```

```
for (remainingGuests in 0..9) {
```

```
    clicks += remainingGuests
```

```
}
```

```
assert clicks == (10*9)/2
```



Structures de contrôle (2)

// for in sur une liste

```
def list = [0, 1, 2, 3]
for (j in list) {
    assert j == list[j]
}
```

// each avec une closure

```
list.each() { item ->
    assert item == list[item]
}
```

// Switch

```
switch(3) {
    case 1 : assert false; break
    case 3 : assert true; break
    default: assert false
}
```



Groovy et Java

Un bref tour de Groovy

Installation



Environnements Mac OsX, Linux Cygwin

Dans ces environnements, on peut utiliser SDKMAN qui permet de gérer les différentes versions installées sur son poste

#Installation SDK

```
curl -s get.sdkman.io | bash
```

#Initialisation env SDK

```
source "$HOME/.sdkman/bin/sdkman-init.sh"
```

Installation Groovy

```
sdk install groovy
```

Vérification version

```
groovy -version
```



Autre façons d'obtenir Groovy

Mac OS X

- Avec MacPorts
`sudo port install groovy`
- Avec Homebrew
`brew install groovy`

Windows

- Installeur fourni par la communauté
<https://bintray.com/groovy/Distributions/Windows-Installer/groovy-3.0.3-installer#files>



Téléchargement

<http://groovy-lang.org/install.html>

Plusieurs distributions sont proposées :

- Binaires
- Documentation
- Binaires / Sources et Documentation



Mise en place manuelle du binaire

Positionner la variable d'environnement ***GROOVY_HOME*** vers le répertoire de décompression de la distribution.

Ajouter *GROOVY_HOME/bin* à la variable d'environnement ***PATH***

Positionner ***JAVA_HOME*** vers un JDK.



Intégration Groovy

Lors de l'intégration de *Groovy* dans une application, on peut utiliser Maven ou Gradle pour indiquer les dépendances.

Plusieurs options sont possibles :

- Dépendances sur le cœur de Groovy
- Dépendance sur un module de Groovy
- Dépendance sur tous les modules,
(*groovy-all*)



Exemple Maven

```
<!-- Groovy Core -->
```

```
<groupId>org.codehaus.groovy</groupId>
```

```
<artifactId>groovy</artifactId>
```

```
<version>2.4.12</version>
```

```
<!-- Module json -->
```

```
<groupId>org.codehaus.groovy</groupId>
```

```
  <artifactId>groovy-json</artifactId>
```

```
  <version>2.4.12</version>
```

```
<!-- Tous les modules -->
```

```
<groupId>org.codehaus.groovy</groupId>
```

```
  <artifactId>groovy-all</artifactId>
```

```
  <version>2.4.12</version>
```



Commandes d'exécution

Il y a 3 commandes permettant d'exécuter du code *Groovy* :

- **groovy** : Exécution d'un script *Groovy*.
- **groovysh** : Shell groovy permettant l'exécution interactive.
- **groovyConsole** : Interface graphique pour exécution interactive et chargement de script



Compilation statique

La compilation de script groovy s'effectue via le compilateur **groovyc** .

Le compilateur génère au moins un fichier *.class* pour chaque fichier Groovy source

Génération dans le répertoire classes
`groovyc -d classes Gold.groovy`



Exécution des classes

Les classes compilées sont ensuite exécutées avec la JRE en positionnant le classpath vers les librairies de Groovy.

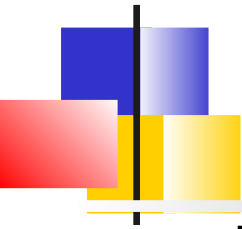
```
java -cp %GROOVY_HOME%/embeddable/groovy-all-  
2.4.0.jar;classes Gold
```



IDEs

Pour une utilisation réduite, un simple éditeur suffit mais si l'on désire réellement développer avec Groovy, il faut un IDE complet (Refactoring, debugging , profiling, test)

- A cause du caractère dynamique de Groovy, le compilateur ne peut pas détecter les erreurs d'appels à des méthodes
- Mais un IDE peut prévenir d'une erreur de typo en mettant en surbrillance les méthodes inconnues et éventuellement effectuer de l'inférence de type afin de proposer de la complétion de code.



Plugins

Des plugins pour Groovy sont disponibles pour la plupart des IDEs :

- IntelliJ IDEA plug-in JetBrains,
- Groovy Eclipse plugin
- Netbeans
- Emacs
- Extension VisualStudioCode
- ...



Les bases de Groovy

Types de données

Collections

Orientation objet

Closures



Tout Objet

Dans *Groovy*, tout est objet. Il n'y a pas de type primitif comme en Java

Groovy inter-opère avec Java via du *boxing* et du *unboxing* automatique

Primitive type	Wrapper type	Description
byte	<code>java.lang.Byte</code>	8-bit signed integer
short	<code>java.lang.Short</code>	16-bit signed integer
int	<code>java.lang.Integer</code>	32-bit signed integer
long	<code>java.lang.Long</code>	64-bit signed integer
float	<code>java.lang.Float</code>	Single-precision (32-bit) floating-point value
double	<code>java.lang.Double</code>	Double-precision (64-bit) floating-point value
char	<code>java.lang.Character</code>	16-bit Unicode character
boolean	<code>java.lang.Boolean</code>	Boolean value (true or false)



Type optionnel

Groovy permet de spécifier explicitement le type d'une variable **ou** de l'omettre

Le mot-clé ***def*** est utilisé pour indiquer qu'aucun type n'est spécifié.

Statement	Type of value	Comment
<code>def a = 1</code>	<code>java.lang.Integer</code>	Implicit typing
<code>def b = 1.0f</code>	<code>java.lang.Float</code>	
<code>int c = 1</code>	<code>java.lang.Integer</code>	Explicit typing using the Java primitive type names
<code>float d = 1</code>	<code>java.lang.Float</code>	
<code>Integer e = 1</code>	<code>java.lang.Integer</code>	Explicit typing using reference type names
<code>String f = '1'</code>	<code>java.lang.String</code>	



Type safe

Même si le type de variable peut être implicite, Groovy est ***type safe*** :
un objet d'un type particulier ne peut pas être assigné à un autre type si il n'y a pas de conversion définie.

=> On peut donc omettre les marqueurs de type mais *Groovy* effectuera les vérifications de type à l'exécution.

`org.codehaus.groovy.runtime.typehandling.GroovyCastException`



Intérêt du typage explicite

On peut explicitement spécifier des types :

- pour forcer le type d'une méthode
- Ou pour bénéficier des vérifications du compilateur



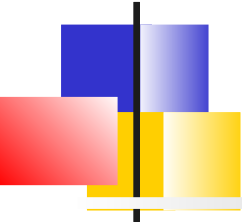
Type optionnel

Les types optionnels apportent des avantages :

- Par exemple, lorsque l'on relaie un objet à d'autre méthode sans le manipuler :

```
def node = document.findMyNode()  
log.info node  
db.store node
```

- Ou lorsque l'on appelle des méthodes sur des objets qui n'ont pas de type garanti.
=> Il suffit que les objets implémentent les mêmes méthodes ...
mais sans spécialement avoir une relation d'héritage, ni même partager une interface (*Duck typing*)



Surcharge d'opérateurs

Groovy base ses opérateurs sur des appels de méthodes.

- Ainsi, les opérateurs peuvent être surchargés et s'appliquer à différents types
- Les opérateurs peuvent alors être utilisés sur nos propres classes du moment qu'elles fournissent la méthode associée (Encore une fois, pas besoin d'implémenter une interface comme en Java !).



Examples

Operator	Name	Method	Works with
<code>a + b</code>	Plus	<code>a.plus(b)</code>	Number, String, StringBuffer, Collection, Map, Date, Duration
<code>a - b</code>	Minus	<code>a.minus(b)</code>	Number, String, List, Set, Date, Duration
<code>a * b</code>	Star	<code>a.multiply(b)</code>	Number, String, Collection
<code>a / b</code>	Divide	<code>a.div(b)</code>	Number
<code>a % b</code>	Modulo	<code>a.mod(b)</code>	Integral number
<code>a++</code> <code>++a</code>	Postincrement Preincrement	<code>def v = a; a = a.next(); v</code> <code>a = a.next(); a</code>	Iterator, Number, String, Date, Range
<code>a--</code> <code>--a</code>	Postdecrement Predecrement	<code>def v = a; a = a.previous(); v</code> <code>a = a.previous(); a</code>	Iterator, Number, String, Date, Range
<code>-a</code>	Unary minus	<code>a.unaryMinus()</code>	Number, ArrayList



Exemple de surcharge

```
import groovy.transform.Immutable
@Immutable // Surcharge implicite de l'opérateur == via equals()
class Money {
    int amount
    String currency
    // Implémentation de + via plus()
    Money plus (Money other) {
        if (null == other) return this
        if (other.currency != currency) {
            throw new IllegalArgumentException("cannot add $other.currency to $currency")
        }
        return new Money(amount + other.amount, currency)
    }
    // Une autre implémentation de +
    Money plus (Integer more) {
        return new Money(amount + more, currency)
    }
}

Money buck = new Money(1, 'USD')
assert buck + buck == new Money(2, 'USD')
assert buck + 1 == new Money(2, 'USD')
```




String

Les chaînes de caractères *Groovy* peuvent être de type :

- ***java.lang.String***
- ***groovy.lang.Gstring***

Les *GStrings* permettent de spécifier des emplacements (placeholder) pouvant être évalués à l'exécution (i.e. interpolation de *String*)



Littéraux

Groovy permet différentes options pour les littéraux :

- Les **simple quotes** pour les *String* Java
- Les **double-quotes** sont équivalentes au simple quote sauf si le texte contient un **\$** (placeholder). Dans ce cas, il s'agit d'une *Gstring*.
- Les **triples-quotes** (*simple* ou *double*) permettent du texte sur plusieurs lignes . Elles peuvent être de simples *String* ou des *GString*
- Si le texte démarre par un **/** ou un **\$/** (légères différences), Il n'est pas nécessaire d'échapper les caractères particulier (ex. backslash)
=> Cela facilite l'écriture des expressions régulières



Examples

Start/end characters	Example	Placeholder resolved?	Backslash escapes?
Single quote	'hello Dierk '	No	Yes
Double quote	"hello \$name "	Yes	Yes
Triple single quote (' ' ')	'''===== Total: \$0.02 ====='''	No	Yes
Triple double quote (" " ")	"""first \$line second \$line third \$line"""	Yes	Yes
Forward slash	/x(\d*)y/	Yes	Occasionally
Dollar slash	\$/x(\d*)y/\$	Yes	Occasionally



Caractères spéciaux

Escaped special character	Meaning
<code>\b</code>	Backspace
<code>\t</code>	Tab
<code>\r</code>	Carriage return
<code>\n</code>	Linefeed
<code>\f</code>	Form feed
<code>\\</code>	Backslash
<code>\\$</code>	Dollar sign
<code>\uabcd</code>	Unicode character $u + abcd$ (where a , b , c , and d are hex digits)
<code>\abc</code>	Unicode character $u + abc$ (where a , b , and c are octal digits, and b and c are optional)
<code>\'</code>	Single quote
<code>\"</code>	Double quote



GString

Une *GString* contient des placeholders exprimés par ***`${expression}`*** ou tout simplement ***`$reference`***.

```
def me = 'Tarzan'
def you = 'Jane'
def line = "me $me - you $you"
assert line == 'me Tarzan - you Jane'
```



Autres exemples *GString*

// Expression

```
TimeZone.default = TimeZone.getTimeZone('GMT')
def date = new Date(0)
def dateMap = [y:date[YEAR]-1900, m:date[MONTH], d:date[DAY_OF_MONTH]]
def out = "Year $dateMap.y Month $dateMap.m Day $dateMap.d"
assert out == 'Year 70 Month 0 Day 1'
```

// Expression complète

```
def tz = TimeZone.getTimeZone('GMT')
def format = 'd MMM YYYY HH:mm:ss z'
out = "Date is ${date.format(format, tz)} !"
assert out == 'Date is 1 Jan 1970 00:00:00 GMT !'
```



Méthodes additionnelles des *String*

```
String greeting = 'Hello Groovy!'
// Utilisation de range
assert greeting[6..11] == 'Groovy'
// Opérateur -
assert 'Hi' + greeting - 'Hello' == 'Hi Groovy!'
// Comptage de caractère
assert greeting.count('o') == 3
// Ajout de caractère à droite ou gauche, centrage
assert 'x'.padLeft(3) == ' x'
assert 'x'.padRight(3, '_') == 'x__'
assert 'x'.center(3) == ' x '
// Opérateur *
assert 'x' * 3 == 'xxx'
```



StringBuffer

Certains méthodes sur les String
retournent des *StringBuffer*

Cela permet de contourner le caractère
immuable des *String* Java

```
def greeting = 'Hello'  
greeting <<= ' Groovy'  
assert greeting instanceof java.lang.StringBuffer
```




regexp

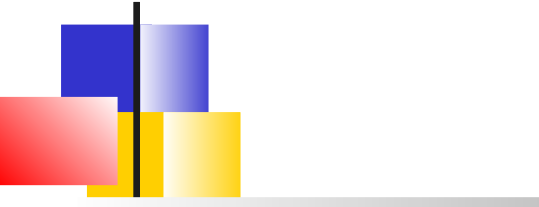
Groovy s'appuie sur Java pour les expressions régulières et ajoute 3 opérateurs :

- *find* , *=~*
- *match* , *==~*
- *pattern* , *~string*
(définition d'un pattern réutilisable)

L'écriture de *regexp* est facilitée via le marqueur */*

```
assert "\\d" == /\d/
```

Symbols regexp



Symbol	Meaning
.	Any character
^	Start of line (or start of document, when in single-line mode)
\$	End of line (or end of document, when in single-line mode)
\d	Digit character
\D	Any character except digits
\s	Whitespace character
\S	Any character except whitespace
\w	Word character
\W	Any character except word characters
\b	Word boundary
()	Grouping
(x y)	x or y, as in (Groovy Java Ruby)
\1	Backmatch to group one; for example, find doubled characters with (.)\1
x *	Zero or more occurrences of x
x +	One or more occurrences of x
x ?	Zero or one occurrence of x
x { m , n }	At least <i>m</i> and at most <i>n</i> occurrences of x
x { m }	Exactly <i>m</i> occurrences of x
[a-f]	Character class containing the characters a, b, c, d, e, f
[^a]	Character class containing any character except a
(?i s : x)	Switches mode when evaluating x; i turns on ignoreCase, s means single-line mode



Usage

Pour une *String* et un motif donnés, *Groovy* permet :

- D'indiquer si le motif correspond à la string complète.
- D'indiquer si il y a une occurrence du motif dans la chaîne.
- Compter le nombre d'occurrences
- Effectuer un traitement avec chaque occurrence.
- Remplacer toutes les occurrences avec un texte
- Diviser la chaîne en plusieurs chaînes en utilisant le motif comme séparateur.



Exemples

```
def twister = 'she sells sea shells at the sea shore of seychelles'
// Opérateur find : twister doit contenir une sous-chaîne de taille 3
// démarrant avec s et terminant avec a
assert twister =~ /s.a/
// Les expressions find sont évaluées comme des Matcher
def finder = (twister =~ /s.a/)
assert finder instanceof java.util.regex.Matcher
// Opérateur match : twister doit contenir seulement des mots délimités par des espaces simples
assert twister ==~ /(\w+ \w+)* /
// Les expressions de Match sont des booléens
def WORD = /\w+/
matches = (twister ==~ /($WORD $WORD)* /)
assert matches instanceof java.lang.Boolean
// Replace et split prennent en arguments un regexp
def wordsByX = twister.replaceAll(WORD, 'x')
assert wordsByX == 'x x x x x x x x x x'
def words = twister.split(/ /)
assert words.size() == 10
```



Exemples

boucle sur les occurrence

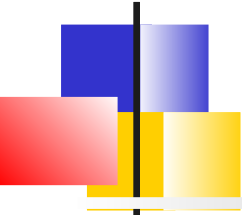
```
def myFairStringy = 'The rain in Spain stays mainly in the plain!'
def wordEnding = /\w*ain/
def rhyme = /\b$wordEnding\b/
def found = ''
myFairStringy.eachMatch(rhyme) { match ->
  found += match + ' '
}
assert found == 'rain Spain plain '
found = ''
(myFairStringy =~ rhyme).each { match ->
  found += match + ' '
}
assert found == 'rain Spain plain '
```



Règles de conversion des nombres

Pour les opérations + , - et *

- Si chaque opérande est *Float* ou un *Double* , le résultat est *Double*.
(A la différence de Java qui si les 2 opérandes sont des *Float* le résultat est *Float*)
- Sinon, si un des opérandes est *BigDecimal* , le résultat est un *BigDecimal*
- Sinon, si un des opérandes est *BigInteger* , le résultat est un *BigInteger*
- Sinon, si un des opérandes est *Long*, le résultat est *Long*
- Sinon le résultat est *Integer*



Méthodes additionnelles sur les nombres

```
def store = ''  
// Répétition  
10.times{ store += 'x' }  
assert store == 'xxxxxxxxxx'  
// Parcours  
store = ''  
1.upto(5) { number -> store += number }  
assert store == '12345'  
  
store = ''  
2.downto(-2) { number -> store += number + ' '}  
assert store == '2 1 0 -1 -2 '  
// Parcours avec un pas  
store = ''  
0.step(0.5, 0.1){ number -> store += number + ' '}  
assert store == '0 0.1 0.2 0.3 0.4 '
```



Types de données
Collections
Orientation objet
Closures



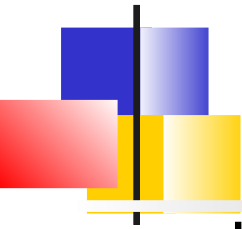
Introduction

Groovy facilite l'utilisation des collections

- Les *List*, *Map* et *Range* peuvent être déclarés par des littéraux.
- Ils ont des opérateurs spécialisés et de nombreuses améliorations par rapport au JDK

La notation utilisée est différente de Java mais la sémantique reste la même

Une collection n'est pas disponible en Java : les intervalles (*Range*)



Range

Un collection de type **Range** a une limite inférieure et supérieure ainsi qu'une stratégie de parcours

Elle permet :

- d'itérer sur chaque élément (*each*)
- De déterminer si un élément appartient à l'intervalle

Les éléments peuvent être de n'importe quel type du moment qu'ils :

- Implémentent les méthodes *next* et *previous* (surcharge des opérateurs ++ et --).
- Implémentent *java.lang.Comparable* ; méthode *compareTo* surchargeant l'opérateur <=>



Exemples (1)

```
assert (0..10).contains(0)
assert (0..10).contains(10)
assert (0..<10).contains(10) == false
// Références à un intervalle
def a = 0..10
a = new IntRange(0,10)
// Intersection d'intervalle
assert (0.0..1.0).contains(1.0)
// Dates
def today = new Date()
def yesterday = today - 1
assert (yesterday..today).size() == 2
// Caractère
assert ('a'..'c').contains('b')
```



Exemples (2)

// Boucle for in

```
def log = ''  
for (element in 5..9){ log += element }  
assert log == '56789'  
log = ''  
for (element in 9..5){ log += element }  
assert log == '98765'
```

// Méthode each

```
log = ''  
(9..<5).each { element -> log += element }  
assert log == '9876'
```



Listes

Les listes Groovy sont par défaut de type ***java.util.ArrayList***

Elles peuvent également être déclarées explicitement avec un constructeur approprié (ex : *new LinkedList(myList)*)

- Les listes peuvent être déclarées et initialisées avec

```
List myList = [1, 2, 3]
```

- Les éléments peuvent être accédés via

```
MyList[0]
```

- Un intervalle peut être converti en *List*

```
List longList = (0..1000).toList()
```

GDK étend tous les tableaux, les collections et les Strings avec *toList()* qui retourne une nouvelle *List*.



Opérateurs

GDK surcharge la méthode ***getAt*** en acceptant un argument de type pour accéder à un sous-ensemble de la collection.

La même stratégie est appliquée à la méthode ***putAt***

```
myList = ['a','b','c','d','e','f']
assert myList[0..2] == ['a','b','c']
assert myList[0,2,4] == ['a','c','e']
myList[0..2] = ['x','y','z']
assert myList == ['x','y','z','d','e','f']
myList[3..5] = []
assert myList == ['x','y','z']
myList[1..1] = [0, 1, 2]
assert myList == ['x', 0, 1, 2, 'z']
```



Opérateurs (2)

Les opérateurs *plus(Object)* , *plus(Collection)* , *leftShift(Object)* , *minus(Collection)* , et *multiply* sont supportés par les listes

```
myList = []  
myList += 'a'  
assert myList == ['a']  
myList += ['b', 'c']  
assert myList == ['a', 'b', 'c']  
myList = []  
myList << 'a' << 'b'  
assert myList == ['a', 'b']  
assert myList - ['b'] == ['a']  
assert myList * 2 == ['a', 'b', 'a', 'b']
```



Structure de contrôle

Les listes peuvent être utilisées dans des *if* , *switch* et *for*

```
myList = ['a', 'b', 'c']
// Switch value
assert myList.isCase('a')
assert 'b' in myList
def candidate = 'c'
switch(candidate){
  case myList : assert true; break
  default : assert false
}
assert ['x','a','z'].grep(myList) == ['a']
myList = []
if (myList) assert false
def expr = ''
for (i in [1,'*',5]){
  expr += i
}
assert expr == '1*5'
```




Map (1)

Les Maps sont par défaut de type ***java.util.LinkedHashMap***.

Elles peuvent également être déclarées explicitement en appelant un constructeur approprié

Elles peuvent être initialisées comme suit :

```
[key1:value1, key2:value2, key3:value3]
```



Map (2)

Les *Maps* permettent de récupérer ou de mettre à jour des valeurs en indiquant la clé :

```
def myMap = [a:1, b:2, c:3]
assert myMap['a']== 1
def emptyMap = [:]
assert emptyMap.size() == 0
```

L'opérateur de propagation *** permet de référencer toutes les valeurs d'une map

```
def myMap = [a:1, b:2, c:3]
def composed = [x:'y', *:myMap]
assert composed == [x:'y', a:1, b:2, c:3]
```



Exemples

```
def myMap = [a:1, b:2, c:3]
// Récupérer un élément existant
assert myMap['a'] == 1
assert myMap.a == 1
assert myMap.get('a') == 1
assert myMap.get('a',0) == 1
// Essayer de récupérer un élément manquant
assert myMap['d'] == null
assert myMap.d == null
assert myMap.get('d') == null
// Valeur par défaut
assert myMap.get('d',0) == 0
assert myMap.d == 0
// Mise à jour
myMap['d'] = 1
assert myMap.d == 1
myMap.d = 2
assert myMap.d == 2
```



Méthodes GDK

GDK ajoute les méthodes ***any*** et ***every*** qui retournent un *Boolean* indiquant si une ou toute les entrées de la Map satisfont une *closure* donnée.

```
assert myMap.any {entry -> entry.value > 2 }  
assert myMap.every {entry -> entry.key< 'd'}
```



Boucles

Il est possible d'itérer sur les entrées ou sur les clés et valeurs séparément

```
def myMap = [a:1, b:2, c:3]
// Boucle sur les entrées
def store = ''
myMap.each { entry ->
  store += entry.key
  store += entry.value
}
assert store == 'a1b2c3'
// Boucle sur les clés et valeurs
store = ''
myMap.each { key, value -> store += key ; store += value}
assert store == 'a1b2c3'
// Boucle sur les clés
store = ''
for (key in myMap.keySet()) {store += key}
assert store == 'abc'
// Boucle sur les valeurs
store = ''
for (value in myMap.values()) {store += value}
assert store == '123'
```



Types de données
Collections
Orientation objet
Closures



Classes et scripts

La définition de classe en Groovy est quasiment identique à Java :

- Elles sont déclarées via le mot-clé **class**
- Elles peuvent contenir des **attributs**, des **initialiseurs**, des **constructeurs** et des **méthodes**
- Les constructeurs et méthodes peuvent utiliser des **variables locales**

Les Scripts sont différents et ajoutent de la flexibilité mais également des restrictions.

Ils peuvent contenir du code, des définitions de variables, de méthodes ou de classes.



Variables

Les variables ou attributs doivent être déclarés avant d'être utilisés

- Sauf pour les scripts : une variable non déclarée est ajoutée au **binding** (map permettant d'échanger des arguments avec l'appelant)

Groovy utilise les mêmes qualifieurs que Java : *private*, *protected*, *public*, *static*

- La visibilité par défaut définit une propriété d'un bean ;
les accesseurs sont alors automatiquement générés par Groovy

Les variables sont typées ou précédées du mot-clé **def**

En plus de la notation `.`, les attributs sont accessibles via la notation `[]`

```
class Counter { public count = 0 }  
def fieldName = 'count'  
counter[fieldName] = 2
```




Méthodes et paramètres

- Les qualifieurs Java peuvent être utilisés
- Déclarer un type de retour est optionnel (utilisation de *def*)
- L'instruction *return* est également optionnelle même si on a déclaré un type de retour. La dernière ligne est retournée.
- La visibilité par défaut est *public*
- La déclaration explicite des types des paramètres est optionnelle
- Lorsque la déclaration n'est pas précisée, le type *Object* est utilisé
- Les appels de méthodes peuvent respecter l'ordre de déclaration des paramètres ou utiliser des paramètres nommés avec une Map
- Les paramètres peuvent avoir une valeur par défaut



Parenthèses

Les parenthèses peuvent également être omises pour les expressions de haut-niveau :

```
println "Hello"  
method a, b  
=>  
println("Hello")  
method(a, b)
```

Dans le cas d'une closure :

```
list.each( { println it } )  
list.each(){ println it }  
list.each { println it }
```

Attention les parenthèses sont obligatoires lors d'appels imbriqués :

```
def foo(n) { n }  
def bar() { 1 }  
println foo 1 // Ne marche pas  
def m = bar   // Ne marche pas
```



Exemples

```
class Summer {  
  def sumWithDefaults(a, b, c=0){ // Valeur par défaut de c  
    return a + b + c  
  }  
  def sumWithList(List args){  
    // Closure avec Initialisation de sum à 0  
    return args.inject(0){sum,i -> sum += i}  
  }  
  def sumWithOptionals(a, b, Object[] optionals){  
    return a + b + sumWithList(optionals.toList())  
  }  
}  
  
def summer = new Summer()  
assert 2 == summer.sumWithDefaults(1,1) // Appel sans paramètre c  
assert 3 == summer.sumWithDefaults(1,1,1)  
assert 3 == summer.sumWithList([1,1,1])  
assert 2 == summer.sumWithOptionals(1,1) // Dernier paramètre non renseigné  
assert 3 == summer.sumWithOptionals(1,1,1)
```



Map et paramètres nommés

Le passage des paramètres lors des appels de méthodes peuvent se faire :

- En utilisant la position de déclaration
- En utilisant une *Map* permettant de nommer les paramètres

```
class Summer {  
  def sumNamed(Map args){  
    ['a','b','c'].each{args.get(it,0)}  
    return args.a + args.b + args.c  
  }  
}  
def summer = new Summer()  
assert 2 == summer.sumNamed(a:1, b:1)
```



Opérateur ?.

Groovy fournit l'opérateur **?.** permettant de se protéger des NPEs (*Elvis Operator*)

Lorsque la référence devant l'opérateur est *null*, l'évaluation de l'expression s'arrête et retourne *null*

```
def map = [a:[b:[c:1]]]  
assert map.a.b.c == 1  
assert map?.a?.x?.c == null
```



Constructeurs

Si aucun constructeur n'est défini, un constructeur implicite est fourni par le compilateur

L'appel à un constructeur peut se faire de 3 façons :

```
class VendorWithCtor {  
    String name, product  
    VendorWithCtor(name, product) {  
        this.name = name  
        this.product = product  
    }  
}  
  
def first = new VendorWithCtor('Canoo','ULC')  
def second = ['Canoo','ULC'] as VendorWithCtor  
VendorWithCtor third = ['Canoo','ULC']
```



Constructeurs avec paramètres nommés

```
class SimpleVendor {  
    String name, product  
}  
new SimpleVendor()  
new SimpleVendor(name: 'Canoo')  
new SimpleVendor(name: 'Canoo', product: 'ULC')  
def vendor = new SimpleVendor(name: 'Canoo')  
assert 'Canoo' == vendor.name
```



Relations entre classes, scripts et fichiers

Les règles suivantes s'appliquent sur un fichier Groovy :

- Si il ne contient pas de déclaration de classe, il est considéré comme un script.
 - Une classe de type *Script* est générée avec le nom du fichier et le contenu est placé dans une méthode *run*.
 - Une méthode *main* permet de lancer le script
- Si il ne contient qu'une déclaration de classe, identique à Java
- Si il contient plusieurs déclarations. Le compilateur crée un fichier *.class* pour chaque déclaration.
 - Si la première classe contient une méthode *main*, le fichier est exécutable via *groovy*.
 - Si lors d'une exécution dynamique, la classe correspondant au nom de fichier est chargé alors toutes les classes du fichier sont chargées
- Si il mélange des déclarations et du code de script. Le script devient la classe principale



Packages et classpath

Groovy suit l'approche Java pour organiser les fichiers sources en packages hiérarchiques.

- La structure en package est utilisée pour trouver les fichiers *.class* sur le système de fichier.
- Comme Java, les classes Groovy doivent spécifier leur package avant la définition de classe (sinon *default package*).
- Groovy permet également les instructions ***import***
- Groovy utilise le classpath pour recherche les fichiers **.groovy*.
- Lors de la recherche d'une classe, si Groovy trouve un **.class* et un **.groovy*, il utilise le plus récent.



Héritage et interface

Les classes *Groovy* peuvent étendre/implémenter des classes/interfaces *Groovy* et Java.

- Groovy supporte complètement le mécanisme d'interface de Java.
- Mais comme Groovy permet un typage dynamique, on peut appeler des méthodes d'une interface sur une classe qui ne l'implémente pas !

=> On peut donc choisir son style de codage



Multi-méthodes

Le mécanisme de Groovy pour rechercher la méthode à appeler prend en compte le type dynamique des arguments.

Cet aspect est appelé **multi-méthodes**

```
def oracle(Object o) { return 'object' }
def oracle(String o) { return 'string' }
// Static types : Object mais runtime types Integer, String
Object x = 1
Object y = 'foo'
// La résolution de la méthode est différente
assert 'object' == oracle(x)
assert 'string' == oracle(y)
```



Usage multi-méthodes

// Surchage sélective de la méthode equals

```
class Equalizer {  
    boolean equals(Equalizer e){  
        return true  
    }  
}  
Object same = new Equalizer()  
Object other = new Object()  
assert new Equalizer().equals( same )  
assert ! new Equalizer().equals( other )
```



traits

Les **traits** permettent de composer les « capacités » d'un objet.

- Les capacités étant la présence d'un attribut ou d'une méthode (~ classe abstraite Java)

Les classes peuvent alors implémenter un trait pour indiquer qu'elles offrent les capacités associées

- L'implémentation peut être déclarative (statique, intrusive)
- Dynamique (non intrusif)

Les classes héritent des implémentations par défaut mais peuvent les surcharger

- Cela ressemble aux méthodes par défaut de Java8 avec en plus les attributs par défaut



Exemple

```
trait HasId { // Un trait avec un état
  long id
}
trait Persistent { // Un trait avec une méthode
  boolean save() { println "saving ${this.dump()}" }
}
trait Entity implements Persistent, HasId { // Composition de traits
  boolean save() {
    Persistent.super.save()
  }
}
class Publication implements Entity { // Implémentation déclarative
  String title
}
class Book extends Publication {
  String isbn
}
Entity gina = new Book(id:1, title:"gina", isbn:"111111")
gina.save()
```



Implémentation dynamique

```
class Publication {  
    String title  
}  
class Book extends Publication {  
    String isbn  
}  
// Attention gina n'est plus de type Book !!  
Entity gina = new Book(title:"gina", isbn:"111111") as Entity  
gina.id = 1
```



Types de données
Collections
Orientation objet
Closures



Introduction

Une ***closure*** est un bloc de code qui est dynamiquement encapsulé dans un objet.

- Il se comporte comme une méthode : Il a des paramètres et retourne une valeur
- Il a accès aux variables de son contexte d'utilisation
- C'est un objet : une variable peut référencer la closure

Groovy fournit un moyen facile de créer des objets closure avec les accolades :

```
Closure envelope = { person -> new Letter(person).send() }  
addressBook.each (enveloppe)
```

Ressemble au Lambda expressions de Java8



Cas d'utilisation

Une *closure* répond au besoin d'exécuter de la logique identique mais arbitraire pour tous les cas

Exemples

- Effectuer des tâches sur des collections.
Dans ce cas le code utile est le corps de l'itérateur
- Utiliser des ressources avec ouverture, fermeture, traitement des exceptions.
Dans ce cas, le code utile est l'utilisation de la ressource après qu'elle ait été acquise et avant qu'elle ne soit rendue

En termes plus généraux, ce type de mécanisme utilise une méthodes de call-back.

Les closure sont une façon de fournir des cibles de callbacks de façon transparente



Déclaration

3 façons de déclarer une *Closure*

- Déclaration simple
- Affectation à une variable
- Référence à une méthode

La syntaxe utilisée par les 2 premières façons est :

```
{ [closureParameters ] -> statements }
```



Déclaration simple

Lorsqu'il n'y a qu'un seul paramètre passé à la *closure*, la déclaration est optionnelle. La variable *it* peut alors être utilisée.

```
log = ''  
(1..10).each{ log += it }
```

Cette syntaxe est une abréviation car l'objet *Closure* est le dernier paramètre de la méthode *each* ; les parenthèses peuvent alors être omises

// Version longue

```
log = ''  
(1..10).each({ counter -> log += counter })
```

// Déclaration par affectation

```
def printer = { line -> println line }
```



Référence à une méthode

Une closure peut faire référence à une méthode en utilisant l'opérateur **.&** sur une référence

```
class SizeFilter {  
  Integer limit  
  boolean sizeUpTo(String value) {  
    return value.size() <= limit  
  }  
}
```

```
SizeFilter filter6 = new SizeFilter(limit:6)
```

```
SizeFilter filter5 = new SizeFilter(limit:5)
```

```
Closure sizeUpTo6 = filter6.&sizeUpTo
```

```
def words = ['long string', 'medium', 'short', 'tiny']
```

```
assert 'medium' == words.find (sizeUpTo6)
```

```
assert 'short' == words.find (filter5.&sizeUpTo)
```



Appel d'une closure

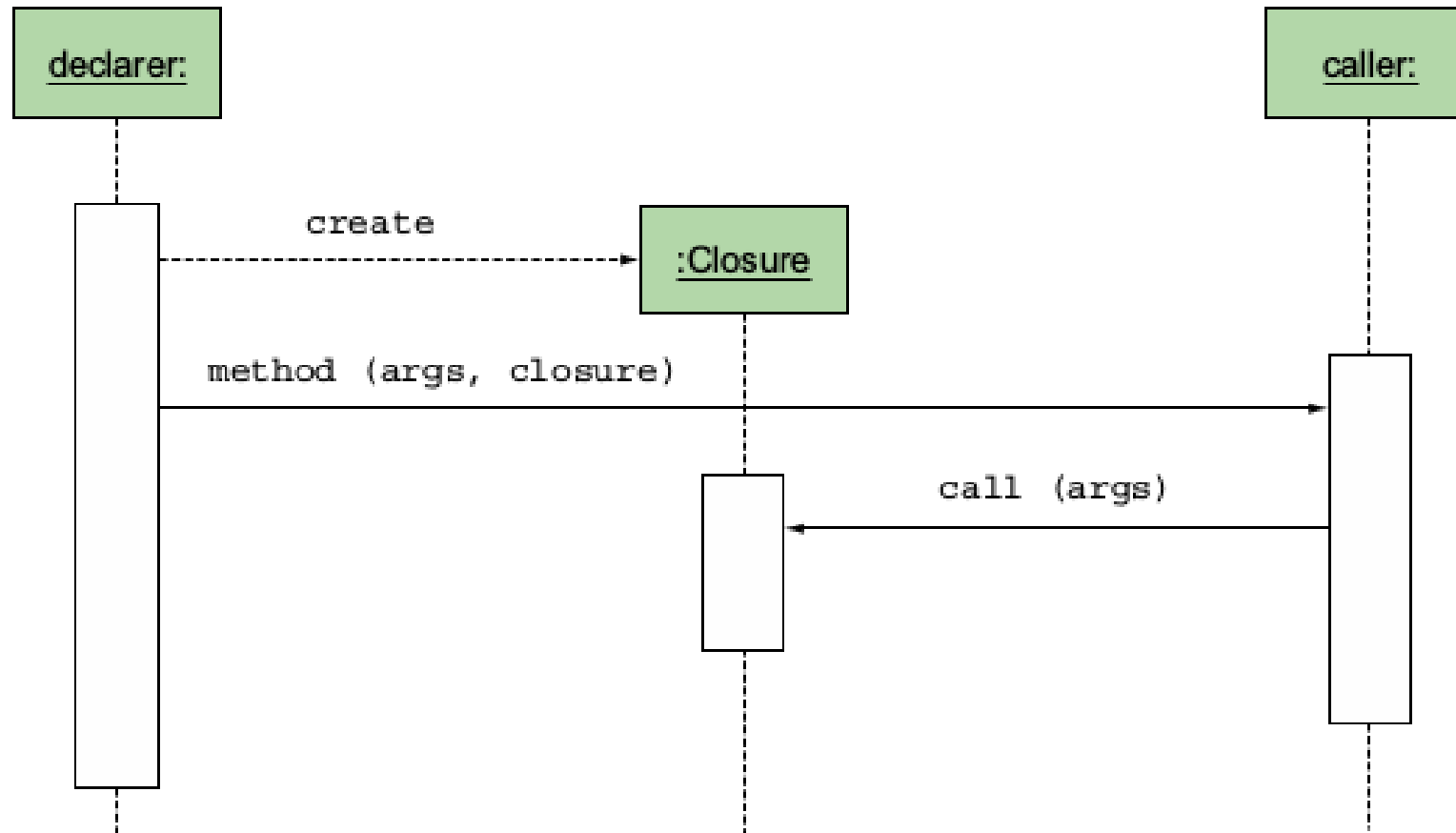
On peut appeler une closure `x` par `x.call()` ou simplement `x()`

```
def adder = { x, y -> return x+y }  
assert adder(4, 3) == 7  
assert adder.call(2, 6) == 8
```

On peut appeler une closure passée en paramètre à l'intérieur d'une méthode

```
def benchmark(int repeat, Closure worker) {  
  def start = System.nanoTime() // ~Ouverture de la ressource  
  repeat.times { worker(it) }  
  def stop = System.nanoTime() // ~Fermeture de la ressource  
  return stop - start  
}  
def slow = benchmark(10000) { (int) it / 2 }  
def fast = benchmark(10000) { it.intdiv(2) }
```

Diagramme de séquence





Contexte

```
def x = 0
10.times {
  x++
}
assert x == 10
```

Dans l'exemple précédent, La *closure* est passée à la méthode *times*

Cette méthode effectue un call-back vers la *closure* mais la méthode *times* ne connaît pas *x* et donc ne peut pas transmettre *x* à la *closure*.

L'exemple fonctionne car la *closure* se souvient du contexte de sa naissance et le préserve tout au long de son existence (d'objet)



Détails sur les objets

Le *Script* crée la *Closure* et devient son ***propriétaire (owner)***.

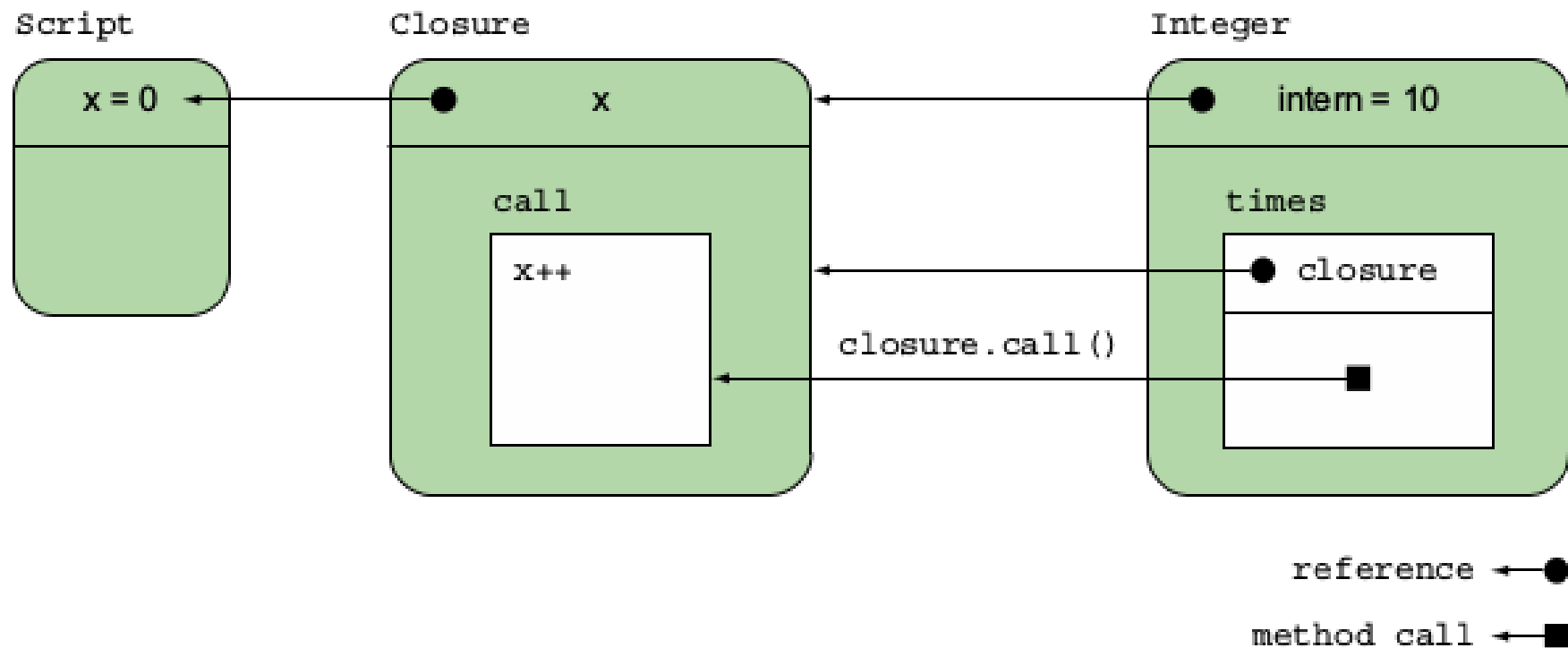
La *Closure* a une référence vers *x* qui est dans le contexte local de son propriétaire

Le *Script* appelle la méthode *times* sur l'entier 10 passant la closure déclarée en paramètre

La méthode *times* utilise cette référence pour exécuter la méthode de la *Closure* en lui passant sa variable locale (pas utilisée dans ce cas)

Closure.call ne travaille qu'avec *x* la référence vers la variable locale *x* de *Script* .

Contextes d'exécution





Propriétaire et délégué

Avec Groovy, le développeur peut contrôler comment les références sont résolues à l'intérieur d'une closure.

Même si il n'est pas possible de directement positionner la valeur de *this*, on peut définir une stratégie de résolution et une classe **déléguée** qui pourra être utilisée pour la résolution de références

- Par défaut l'objet délégué est l'objet propriétaire

L'attribut ***resolveStrategy*** peut prendre les valeurs :

- *Closure.OWNER_FIRST* (valeur par défaut), *Closure.OWNER_ONLY*
- *Closure.DELEGATE_FIRST*, *Closure.DELEGATE_ONLY*
- *Closure.TO_SELF*



Exemple

```
class Mother {
  def prop = 'prop'
  def method(){ 'method' }
  Closure birth (param) {
    def local = 'local'
    def closure = {
      [ this, prop, method(), local, param ]
    }
    return closure
  }
}

Mother julia = new Mother()
def closure = julia.birth('param')
def context = closure.call()
assert context[0] == julia
assert context[1 .. 4] == ['prop', 'method', 'local', 'param' ]
// Propriétés read-only
assert closure.thisObject == julia
assert closure.owner == julia
// Délégué et stratégie par défaut
assert closure.delegate == julia
assert closure.resolveStrategy == Closure.OWNER_FIRST
```



Mise à jour du délégué

Il est possible de mettre à jour l'objet délégué, la méthode ***with*** permet d'exécuter une closure en positionnant au préalable le délégué au récepteur de la méthode *with* :

```
def map = [:]  
map.with { // Le délégué devient map  
  a = 1 // équivalent à map.a = 1  
  b = 2 // équivalent à map.b = 2  
}  
assert map == [a:1, b:2]
```



Valeurs de retour

Il y a 2 façons de retourner de l'exécution d'une Closure :

- La dernière expression de la closure. L'utilisation du mot clé *return* est optionnelle.
- Le mot clé **return** peut être utilisé pour retourner prématurément.

```
[1, 2, 3].collect{ it * 2 }  
[1, 2, 3].collect{ return it * 2 }  
[1, 2, 3].collect{  
    if (it%2 == 0) return it * 2  
    it  
}
```



Closure vs Lamda

A la différence des lambda expressions de Java 8, les closures sont des instances de la classe *Closure*.

La délégation est un concept clé dans les Closures Groovy qui n'a pas d'équivalent dans les lambdas.

- Par exemple : la possibilité de changer le délégué ou de changer la stratégie de délégation des fermetures permet de concevoir des DSLs dans Groovy.

On peut utiliser une *Closure* à la place d'une lambda

```
list.stream().filter { true }
```



Méta-programming

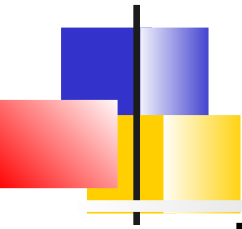
Programmation dynamique
Transformations AST, annotations



Introduction

La programmation dynamique permet d'ajouter des capacités à des objets sans modifier la classe originale (Classe Groovy ou Java) du moment qu'elle soit appelée par Groovy

Autrement dit, elle permet d'ajouter des propriétés ou des méthodes dynamiquement



Usages

La programmation dynamique a un large éventail d'application :

- Mettre au point un *Domain Specific Language*
- Implémenter des builders
- Le logging, tracing, debugging et profiling
- Les tests automatisés
- Compléter une librairie
- Organiser le code.
 - Ex : Fournir une abstraction qui encapsule de la collaboration entre des sous-classes dispersées



Mécanisme

Le **MOP (Méta-Object Protocol)** est l'ensemble des règles régissant l'appel à une méthode par le runtime Groovy incluant des appels à des méthodes de ***hooks***

=> Lorsque *Groovy* appelle une méthode, il ne l'appelle pas directement mais passe par une couche intermédiaire fournissant des *hooks*.



InvokeHelper

En fait,

```
println 'Hello' // Groovy
```

Devient

```
InvokerHelper.invokeMethod(this, "println", {"Hello"}); // Java
```

Dans ce cas, *InvokerHelper* (partie prenante du MOP) :

- Recherche la méthode nommée "*println*" avec un argument *String*
- Trouve que le runtime Groovy a enregistré cette méthode pour *java.lang.Object*
- Appelle alors cette implémentation.



Méta-class et méthodes de hook

Le MOP doit accéder à de nombreuses informations pour trouver la bonne méthode à appeler.

Cette information est stockée dans les **méta-classes**.

- Ces méta-classes ne sont pas fixes. La programmation dynamique permet de changer le contenu des méta-classes

Le MOP appelle également des méthodes spéciales : les **hooks**.

- La personnalisation des méthodes de hook permet une programmation dynamique sans modifier les méta-classes



Exemple *methodMissing*

MOP gère le cas où il ne trouve pas de méthode cible en appelant la méthode :

Object methodMissing(String name, Object arguments)

- Le comportement par défaut est de lancer une *MissingMethodException* mais il peut être surchargé

Un exemple d'implémentation

```
class Pretender {  
    def methodMissing(String name, Object args) {  
        "called $name with $args"  
    }  
}  
  
def bounce = new Pretender()  
assert bounce.hello('world') == 'called hello with [world]'
```



Autre Exemple

```
class MiniGorm {  
  def db = []  
  def methodMissing(String name, Object args) {  
    db.find { it[name.toLowerCase()-'findby'] == args[0] }  
  }  
}  
def people = new MiniGorm()  
def dierk = [first: 'Dierk', last:'Koenig']  
def paul= [first: 'Paul', last:'King']  
people.db << dierk << paul  
assert people.findByFirst('Dierk') == dierk  
assert people.findByLast('King')== paul
```



propertyMissing

propertyMissing est l'équivalent de *methodMissing* pour les accès aux propriétés.

Object *propertyMissing*(String name)

– Comportement par défaut : *MissingPropertyException*

Exemple d'implémentation :

// Calcul de nombre binaire, (~ DSL)

```
def propertyMissing(String name) {  
  int result = 0  
  name.each {  
    result <= 1  
    if (it == 'I') result++  
  }  
  return result  
}
```

// Comparaison de 2 entiers

```
assert II0I + I0I == I00I0
```




Hook dynamiques

Les méthodes de hook peuvent travailler avec l'état de l'instance, il est donc possible de changer leur comportement pendant une exécution.

```
class DynamicPretender {  
  Closure whatToDo = { name -> "accessed $name"}  
  def propertyMissing(String name) {  
    whatToDo(name)  
  }  
}  
  
def one = new DynamicPretender()  
assert one.hello == 'accessed hello'  
one.whatToDo = { name -> name.size() }  
// La méthode hello ne fait plus la même chose !!  
assert one.hello == 5
```



Interface *GroovyObject*

Toutes les classes compilées par Groovy implémentent l'interface ***GroovyObject***

```
public interface GroovyObject {  
    Object invokeMethod(String methodName, Object args);  
    Object getProperty(String propertyName);  
    void setProperty(String propertyName, Object newValue);  
    MetaClass getMetaClass();  
    void setMetaClass(MetaClass metaClass);  
}
```

Il est possible d'implémenter soi-même ces méthodes, sinon le compilateur insère une implémentation par défaut



Implémentation par défaut

L'implémentation par défaut relaie les appels de méthodes à la méta-classe

```
public abstract class GroovyObjectSupport implements GroovyObject {
    public Object invokeMethod(String name, Object args) {
        return getMetaClass().invokeMethod(this, name, args);
    }
    public Object getProperty(String property) {
        return getMetaClass().getProperty(this, property);
    }
    public void setProperty(String property, Object newValue) {
        getMetaClass().setProperty(this, property, newValue);
    }
    // more here...
}
```



Interface *GroovyObject*

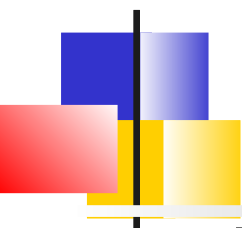
Le comportement des classes implémentant *GroovyObject* est le suivant :

- Chaque accès à une propriété appelle la méthode *getProperty()*
- Chaque modification, la méthode *setProperty()*
- Chaque appel à une méthode inconnue appelle *invokeMethod()*
- Si la méthode est connue, *invokeMethod()* n'est appelé seulement si la classe implémente interface marqueur *GroovyInterceptable* .



Exemple de surcharge

```
class NoParens {
    def getProperty(String propertyName) {
        if (metaClass.hasProperty(this, propertyName)) {
            return metaClass.getProperty(this, propertyName)
        }
        invokeMethod propertyName, null
    }
}
class PropUser extends NoParens {
    boolean existingProperty = true
}
def user = new PropUser()
assert user.existingProperty
assert user.toString() == user.toString
```



Modification de méta-classe

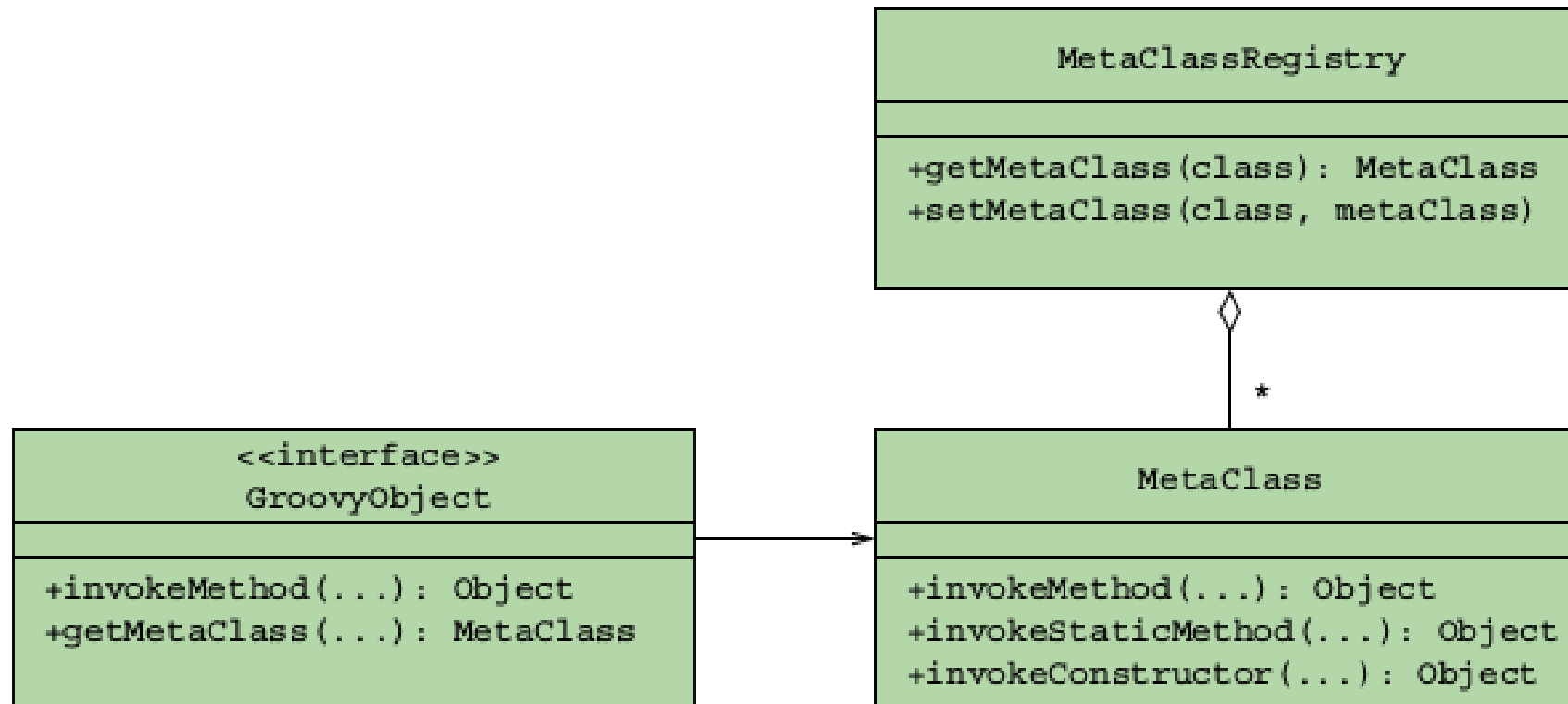
Pour chaque classe chargée, *Groovy* maintient un objet de type *MetaClass* dans un registre (*MetaClassRegistry*) .

- La méta-classe maintient l'ensemble des méthodes et des propriétés de la classe associée
- En général, toutes les instances d'une classe partagent la même méta-classe

Mais, une méta-classe peut changer au runtime et une instance peut changer sa méta-classe.

- C'est le 2^{ème} aspect de la programmation dynamique

Diagramme





Informations d'une méta-classe

Il est aisé de récupérer les informations des meta-classes :

```
MetaClass mc = String.metaClass
final Object[] NO_ARGS = []
assert 1 == mc.respondsTo("toString", NO_ARGS).size()
assert 3 == mc.properties.size()
assert 74 == mc.methods.size()
assert 176 == mc.metaMethods.size()
assert "" == mc.invokeMethod("", "toString", NO_ARGS)
assert null == mc.invokeStaticMethod(String, "println", NO_ARGS)
assert "" == mc.invokeConstructor(NO_ARGS)
```




Méta-classes Groovy

Groovy fournit différentes méta-classes :

- La méta-classe par défaut : ***MetaClassImpl*** utilisée dans la plupart des cas
- ***ExpandoMetaClass*** , qui peut étendre le statut et le comportement d'une classe
- ***ProxyMetaClass*** , qui permet de décorer une classe avec des intercepteurs
- D'autres utilisées en interne



Exemple intercepteur non-intrusif

```
class InspectMe {
  int outer(){
    return inner()
  }
  private int inner(){
    return 1
  }
}

def tracer = new TracingInterceptor(writer: new StringWriter())
def proxyMetaClass = ProxyMetaClass.getInstance(InspectMe)
proxyMetaClass.interceptor = tracer
InspectMe inspectMe = new InspectMe()
inspectMe.metaClass = proxyMetaClass
assert 1 == inspectMe.outer()
assert "\n" + tracer.writer.toString() == ""
before InspectMe.outer()
  before InspectMe.inner()
  after InspectMe.inner()
after InspectMe.outer()
""
```



Expando

La classe **Expando** est un bean dynamique. A l'exécution, on peut y ajouter des propriétés et des méthodes via des Closures

```
def user = new Expando(username: 'mrhaki')
assert 'mrhaki' == user.username
```

// Ajout de propriété

```
user.email = 'email@host.com'
assert 'email@host.com' == user.email
```

// Ajout de méthode via une closure

```
user.printInfo = { writer ->
    writer << "Username: $username"
    writer << ", email: $email"
}
```

```
def sw = new StringWriter()
user.printInfo(sw)
assert 'Username: mrhaki, email: email@host.com' == sw.toString()
```



ExpandoMetaClass

La classe ***ExpandoMetaClass*** est une metaclass fonctionnant comme *Expando*.

On peut enregistrer de nouvelles propriétés et méthodes dans la métaclasse avec des affectations de propriétés

- Il n'est pas nécessaire de positionner explicitement *ExpandoMetaClass*.

Groovy le fait automatiquement aussitôt que l'on ajoute de nouvelles méthodes ou propriétés dans la métaclasse

```
assert String.metaClass =~ /MetaClassImpl/  
String.metaClass.low = { -> delegate.toLowerCase() }  
assert String.metaClass =~ /ExpandoMetaClass/  
assert "DiErK".low() == "dierk"
```



Modifications de méta-classe

La méta-classe peut être modifiée à plusieurs niveaux :

- Au niveau de la classe Groovy : Toutes les nouvelles instances utiliseront la nouvelle méta-classe
- Au niveau d'une instance Groovy : Seule l'instance de cette classe utilisera la nouvelle méta-classe
- Au niveau d'une instance d'une classe Java



Exemple instance Java

```
def myJava = new String()  
myJava.metaClass.myProp = "MyJava prop"  
myJava.metaClass.test = {-> myProp }
```

```
try {  
    new String().test()  
    assert false, "should throw MME"  
} catch(mme) { }
```

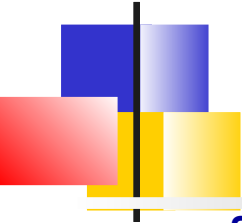
```
assert myJava.test() == "MyJava prop"
```



Meta-class builder

Il est possible d'utiliser le style builder pour effectuer plusieurs changements à la fois

```
def move(string, distance) {  
  string.collect { (it as char) + distance as char }.join()  
}  
  
String.metaClass {  
  shift = -1  
  encode {-> move delegate, shift }  
  decode {-> move delegate, -shift }  
  getCode {-> encode() }  
  getOrig {-> decode() }  
}  
  
assert "IBM".encode() == "HAL"  
assert "HAL".orig == "IBM"  
def ibm = "IBM"  
ibm.shift = 7  
assert ibm.code == "PIT"
```



Ajout d'opérateur et hook de méthode

```
String.metaClass {
  rightShiftUnsigned = { prefix ->
    delegate.replaceAll(~/\w+/) { prefix + it }
  }
  methodMissing = { String name, args->
    delegate.replaceAll name, args[0]
  }
}

def people = "Dierk,Guillaume,Paul,Hamlet,Jon"
people >>>= "\n"
"

people = people.Dierk('Mittie').Guillaume('Mr.G')
assert people == '''
Mittie,
Mr.G,
Paul,
Hamlet,
Jon'''
```




Résumé

- Tous les appels de méthodes en Groovy s'exécutent via une méta-classe
- Les méta-classes peuvent être changées pour toutes les instances ou pour une instance spécifique
- Les changements de métaclasses affectent toutes les futures instances dans toutes les threads
- Les méta-classes permettent des changements non intrusifs aux classes Groovy et Java du moment que l'appelant est Groovy
- Les changements peuvent prendre la forme d'accesseurs, d'opérateurs, de méthodes de *GroovyObject* ou des méthodes de hook
- *ExpandoMetaClass* permet de faciliter les modifications
- Il est préférable d'effectuer les changements sur les méta-classes une fois au démarrage de l'application.



Changements temporaires

ExpandoMetaClass n'est pas conçue pour des changements temporaires.

Groovy fournit les classes **category** qui permettent d'appliquer de la programmation dynamique seulement pour la thread courante et une petite partie de code

L'utilisation consiste à utiliser la méthode **use** (rajoutée par Groovy sur *java.lang.Object*)

La méthode prend 2 paramètres

- Une classe category (ou plusieurs)
- Une closure

```
use CategoryClass, {  
// new methods are available  
}  
// new methods are no longer available
```

Groovy fournit des classes *category* et il est possible de créer ses propres classes



Exemple

```
import groovy.time.TimeCategory
def janFirst1970 = new Date(0)
use TimeCategory, {
    Date xmas = janFirst1970 + 1.year - 7.days
    assert xmas.month == Calendar.DECEMBER
    assert xmas.date == 25
}
```

A l'intérieur de la closure de nouvelles propriétés sont disponibles sur les nombres (*1.year*), un nouvel opérateur pour calculer les dates même si *janFirst1970* a été construit avant la closure

A l'extérieur du bloc, ses caractéristiques ne sont pas disponibles.



Implémentation de *category*

Les classes ***Category*** ne doivent pas implémenter une interface ni étendre une classe particulière.

Elles ne sont pas non plus configurées ou enregistrées

La seule obligation est de contenir des méthodes ***static*** avec au moins un paramètre

Lorsqu'une classe est utilisée comme argument de la méthode *use*, elle devient une classe *category* et chaque méthode

```
static ReturnType methodName(Receiver self, optionalArgs) {...}
```

devient disponible sur le récepteur comme si il avait la méthode d'instance :

```
ReturnType methodName(optionalArgs) {...}
```



Example

```
class Marshal {  
  static String marshal(Integer self) {  
    self.toString()  
  }  
  static Integer unMarshal(String self) {  
    self.toInteger()  
  }  
}  
  
use Marshal, {  
  assert 1.marshal() == "1"  
  assert "1".unMarshal() == 1  
  [Integer.MIN_VALUE, -1, 0, Integer.MAX_VALUE].each {  
    assert it.marshal().unMarshal() == it  
  }  
}
```



Modules d'extensions

Les modules d'extensions peuvent être vus comme des catégories toujours visibles

Il est alors possible d'enrichir les classes du JDK ou de Groovy avec des méthodes personnalisées

Il est possible de packager ses extensions dans ses propres fichiers JAR et les rendre disponibles à d'autres programmes



Mise en place

Convertir une *category* en un module d'extension consiste à :

- Écrire la classe contenant les méthodes *static* dans un fichier source
- Écrire un descripteur d'extension et le positionner dans le classpath



Descripteur

Le descripteur doit s'appeler ***org.codehaus.groovy.runtime.ExtensionModule*** et se placer dans *META-INF/services* du JAR .

Il consiste de 4 entrées :

- ***moduleName*** : Impossible de charger 2 modules du même nom
- ***moduleVersion*** :
- ***extensionClasses*** : Listes des classes Category
- ***staticExtensionClasses*** : Listes des classes ajoutant des méthodes statiques à des classes existantes



@Category

L'annotation **@Category** permet d'écrire sa classe *Category* comme une classe d'instance normale

L'annotation l'ajuste pour rendre les méthodes *static* et ajouter le paramètre *self*

```
@Category(Integer)
class IntegerMarshal {
    String marshal() {
        toString()
    }
}

use IntegerMarshal {
    assert 1.marshal() == "1"
}
```



Méta-programming

Programmation dynamique
Transformations AST, annotations



Introduction

AST (abstract syntax tree) est une représentation du code sous forme d'arbre disponible après le parsing du code source

Les transformations permettent de modifier le code généré sans passer par le code source

- Aucun code source n'est généré, seulement du bytecode

Groovy fournit des transformations *AST* via des **annotations** et il est possible de définir ses propres transformations



Classification des transformations

Les principales transformations AST peuvent être classifiées comme suit :

- Annotations de génération de code
- Annotations de design de Class ou pattern
- Améliorations du Logging
- Déclaration de concurrence
- Facilitation du cloning ou de l'externalisation
- Support pour le Scripting
- ...



Génération de code

@ToString : Implémente *toString()* qui affiche le nom de la classe et la valeur de tous ses attributs

@EqualsAndHashCode : Implémente *equals()* et *hashCode()* à partir des propriétés. (Possibilité d'exclure des propriétés via les attributs de l'annotation)

@TupleConstructor : Implémente des constructeurs à partir des propriétés

@Canonical : combine *@ToString*, *@EqualsAndHashCode* et *@TupleConstructor*

@Lazy : retarde l'instanciation d'un champ jusqu'à sa première utilisation

@Indexed-Property : Ajoute des méthodes permettant d'accéder à une propriété de type *List* par un index.

@InheritConstructors : Ajoute les constructeurs de la classe mère

@Builder : Permet de faciliter la construction d'une instance en appliquant le pattern *builder*

@Sortable : Implémente les méthodes requises par *Comparable* et *Comparator*. On précise les propriétés à prendre en compte dans les attributs

@NullCheck : Ajoute des vérifications d'arguments *null* pour les constructeurs et les méthodes



Exemple *@Builder*

```
import groovy.transform.builder.Builder
@Builder
class Chemist {
    String first
    String last
    int born
}
def builder = Chemist.builder()
def c = builder.first("Marie").last("Curie").born(1867).build()
assert c.first == "Marie"
assert c.last == "Curie"
assert c.born == 1867
```



Design

@Immutable : Une instance ne peut pas être modifiée après sa construction.

L'annotation apporte également : *@ToString*,
@EqualsAndHashCode, *@TupleConstructor*

@Delegate : Permet de spécifier une instance déléguée et d'implémenter toutes ses méthodes en appelant la méthode déléguée.

@Singleton : Une seule instance possible

@Memoized : Permet de cacher les résultats d'une méthode stateless, i.e. une fonction

@TailRecursive : Rendre un code récursif simplement itératif



Exemple *@Memoized*

```
import groovy.transform.Memoized
class Calc {
    def log = []
    @Memoized
    int sum(int a, int b) {
        log << "$a+$b"
        a + b
    }
}
new Calc().with {
    assert sum(3, 4) == 7
    assert sum(4, 4) == 8
    // Utilisation du cache
    assert sum(3, 4) == 7
    assert log.join(' ') == '3+4 4+4'
}
```




Logging

Les annotations **@Log**, **@Log4j**, **@Log4j2**, **@Slf4j** et **@Commons** crée un logger basé sur le nom de la classe et encapsule les méthodes de *logging* dans un test vérifiant le niveau de trace autorisé.

```
import groovy.util.logging.Log
@Log
class Database {
    def search() {
        log.fine(runLongDatabaseQuery())
    }
    def runLongDatabaseQuery() {
        println 'Calling database'
        /* ... */
        return 'query result'
    }
}
new Database().search()
```



Concurrence

@Synchronized : Permet d'avoir un grain plus fin que le mot clé *synchronized* de java

@WithReadLock et **@WithWriteLock** :
Implémentation de *ReentrantReadWriteLock* de Java.
Un seul writer et plusieurs readers



Cloning et Externalisation

@AutoClone : Implémentation de
Clonable avec différentes stratégies
possibles

@AutoExternalize : Implémentation de
Externalizable



Scripting

@Field : Permet de rendre accessible de l'extérieur une variable de script. (Voir partie sur intégration)

@BaseScript : Permet de personnaliser une classe script parente

@TimedInterrupt : Une exception *TimeoutException* est lancée si le script dure trop longtemps

@ThreadInterrupt : le script vérifie le flag *isInterrupted()* et lance une *InterruptedException* si besoin

@ConditionalInterrupt : Permet de spécifier la condition d'interruption



Le GDK

Librairies cœur

Groovy SQL
Json et Groovlets



Introduction

L'éco-système groovy est constitué :

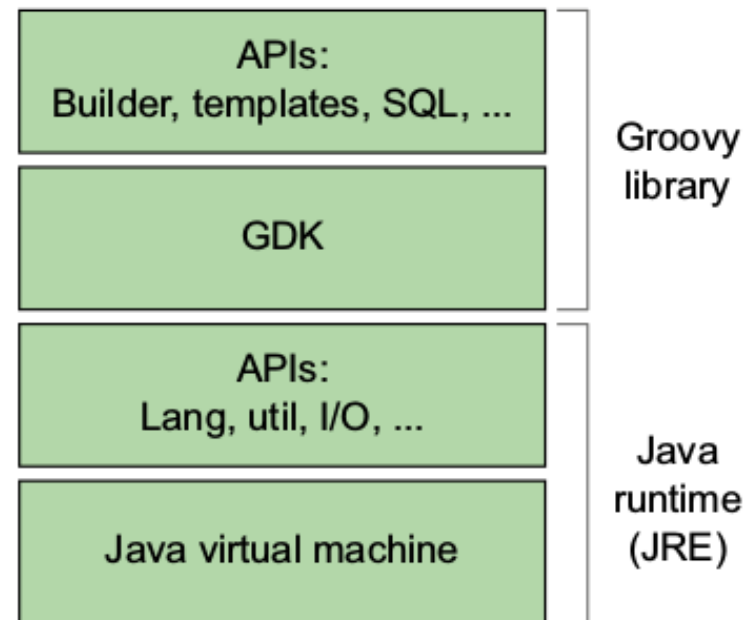
- Du GDK
- De librairies additionnelles (BD, XML, ...)
- De frameworks complétant les fonctionnalités de Groovy, (Tests unitaires, Couverture de code, ...)
- D'outils utilisant Groovy et ses facilités pour définir des DSLs (*Gradle, Jenkins*)
- Des outils utilisant Groovy comme langage de script (*JMeter, JasperReport, ...*)

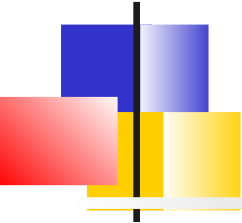


Groovy Development Kit

Groovy fournit une extension à Java : **GDK**

Le GDK inclut de nouvelles classes et librairies utilitaires et s'intègre de façon cohérente avec les classes Java cœur





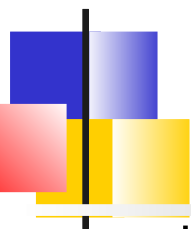
Ajout à *Object*

dump retourne la valeur de ses champs

inspect essaie de fournir une
représentation en code source

properties : Accesseurs aux propriétés
et leurs valeurs

hasProperty*, *respondTo : Permettent
de tester si un objet contient une
propriété, une méthode



Méthodes pour objets itératifs

```
boolean any {closure}
Collection List collect {closure}
Object each {closure}
Object eachWithIndex {closure}
boolean every {closure}
Object find {closure}
Collection findAll {closure}
int findIndexOf {closure}
List findIndexValues {closure}
int findLastIndexOf {closure}
Object findResult {closure}
List grep(Object filter)
Object inject {closure}
Collection split {closure}
```



I/O et fichiers

Groovy facilite énormément la gestion des fichiers :

Parcourir le système de fichier : *eachDir, eachDirMatch, eachDirRecurse, eachFile, eachFileMatch* et *eachFileRecurse* qui prennent une *Closure* en paramètre

Lecture : Propriété *text*, *readLines, eachLine, eachByte, splitEachLine, withReader*

Ecriture : Propriété *text*, *write, append, withWriter, withWriterAppend, withPrintWriter, writeLine*

Filtre, transformations : *transformChar, transformLine, filterLine*

Répertoires temporaires : *createTempDir()*

Scripts Ant : *AntBuilder* permet de créer des scripts *Ant* via *Groovy*. Utilisé par *Gradle*



Threads

Closure implémente ***Runnable***

// Simple thread

```
t = new Thread() { /* Closure body */ }  
t.start()
```

// Démon

```
Thread.startDaemon { /* Closure body */ }
```

// Démarrage après 1000 ms

```
new Timer().runAfter(1000){ /* Closure body */}
```



Processus

Le GDK propose un type ***Process*** permettant de démarrer des processus en dehors de la JVM.

```
Process proc = myCommandString.execute()
```

Les méthodes disponibles sont nombreuses en voici quelques unes :

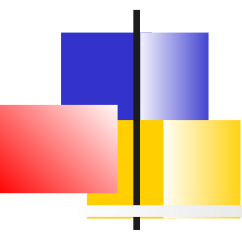
*getErr(), getIn(), getOut(), pipeTo(Process),
waitForOrKill(long),
withOutputStream(Closure),
withWriter(Closure)*



Builders

Groovy propose des classes *Builder* facilitant la création d'arbres spécifiques (XML, JSON, ...)

Il propose également les classes utilitaires ***BuilderSupport*** et ***FactoryBuilderSupport*** facilitant l'implémentation de ses propres Builders



Exemple document à balises

```
def builder = new groovy.xml.MarkupBuilder()
builder.numbers {
    description 'Squares and factors of 10..15'
    for (i in 10..15) {
        number (value: i, square: i*i) {
            for (j in 2..<i) {
                if (i % j == 0) {
                    factor (value: j)
                }
            }
        }
    }
}
```



Librairies

Groovy fournit de nombreuses autres librairies, permettant :

- De générer du contenu Web, template et groovlets
- Interagir avec les BD ou du NoSQL
- Travailler avec XML, JSON
- Interagir avec des services web (REST ou SOAP)



Le GDK

Librairies cœur
Groovy SQL
Json et Groovlets



Introduction

Le module ***groovy-sql*** fournit une abstraction de plus haut niveau que *JDBC*

La classe la plus fréquemment utilisée est la classe *groovy.sql.Sql* qui permet :

- Ouvrir des Connections vers une BD
- Récupérer une connection d'une DataSource
- Exécuter du code SQL
- Effectuer les opérations CRUD
- Délimiter des transactions
- ...



Connections BD

```
@Grab('org.hsqldb:hsqldb:2.3.2')
@GrabConfig(systemClassLoader=true) // Du au chargement des Driver jdbc
import groovy.sql.Sql

def url = 'jdbc:hsqldb:mem:GinA'
def user = 'sa'
def password = ''
def driver = 'org.hsqldb.jdbcDriver'
def sql = Sql.newInstance(url, user, password, driver)

// Utilisation de l'instance 'sql'
sql.close()
```

OU

```
Sql.withInstance(url, user, password, driver) { sql ->
    // Utilisation de l'instance 'sql'
}
```



Datasource

```
import groovy.sql.Sql
import org.hsqldb.jdbc.JDBCDataSource

def dataSource = new JDBCDataSource(
    database: 'jdbc:hsqldb:mem:marathon',
    user: 'sa',
    password: '')
def sql = new Sql(dataSource)
// Utilisation de l'instance
sql.close()
```



Exécution SQL

Des instructions SQL peuvent être exécutées via la méthode ***execute()***

```
// ... create 'sql' instance
sql.execute '''
    CREATE TABLE Author (
        id            INTEGER GENERATED BY DEFAULT AS
        IDENTITY,
        firstname     VARCHAR(64),
        lastname      VARCHAR(64)
    );
'''
// close 'sql' instance ...
```



Insertion

Pour l'insertion, on peut utiliser
executeInsert() à la place *execute()*

- La méthode renvoie un tableau de clés primaires
- Elle peut prendre des paramètres via la notation ?

```
def insertSql = 'INSERT INTO Author (firstname, lastname) VALUES  
  (?,?)'  
def params = ['Jon', 'Skeet']  
def keys = sql.executeInsert insertSql, params  
assert keys[0] == [1]
```



Lecture

Les méthodes ***query()***, ***eachRow()***, ***firstRow()*** et ***rows()*** permettent d'exécuter des requêtes.

```
rowNum = 0
sql.eachRow('SELECT firstname, lastname FROM Author') {
  row ->
  def first = row[0]
  def last = row.lastname
  assert expected[rowNum++] == "$first $last"
}
```



Mise à jour

La méthode ***executeUpdate()*** renvoie le nombre de lignes mises à jour.

```
def updateSql = "UPDATE Author SET lastname='Pragt' where  
    lastname='Thorvaldsson'"  
def updateCount = sql.executeUpdate updateSql  
assert updateCount == 1  
  
def row = sql.firstRow "SELECT * FROM Author where firstname =  
    'Erik'"  
assert "${row.firstname} ${row.lastname}" == 'Erik Pragt'
```



Suppression

Pas de méthode spécifique pour la suppression

```
assert sql.firstRow('SELECT COUNT(*) as num FROM Author').num == 3
sql.execute "DELETE FROM Author WHERE lastname = 'Skeet'"
assert sql.firstRow('SELECT COUNT(*) as num FROM Author').num == 2
```




Transactions

Pour délimiter une transaction, il suffit d'inclure ses opérations SQL dans la closure ***withTransaction***

Si une erreur survient : rollback

```
assert sql.firstRow('SELECT COUNT(*) as num FROM Author').num == 0
sql.withTransaction {
  sql.execute "INSERT INTO Author (firstname, lastname) VALUES ('Dierk',
    'Koenig')"
  sql.execute "INSERT INTO Author (firstname, lastname) VALUES ('Jon',
    'Skeet')"
}
assert sql.firstRow('SELECT COUNT(*) as num FROM Author').num == 2
```



Batch

Pour exécuter plusieurs instructions SQL en lot, utiliser la closure ***withBatch*** et la méthode ***addBatch***

```
def qry = 'INSERT INTO Author (firstname, lastname) VALUES (?,?)'
sql.withBatch(3, qry) { ps ->
  ps.addBatch('Dierk', 'Koenig')
  ps.addBatch('Paul', 'King')
  ps.addBatch('Guillaume', 'Laforge')
  ps.addBatch('Hamlet', "D'Arcy")
  ps.addBatch('Cedric', 'Champeau')
  ps.addBatch('Erik', 'Pragt')
  ps.addBatch('Jon', 'Skeet')
}
```



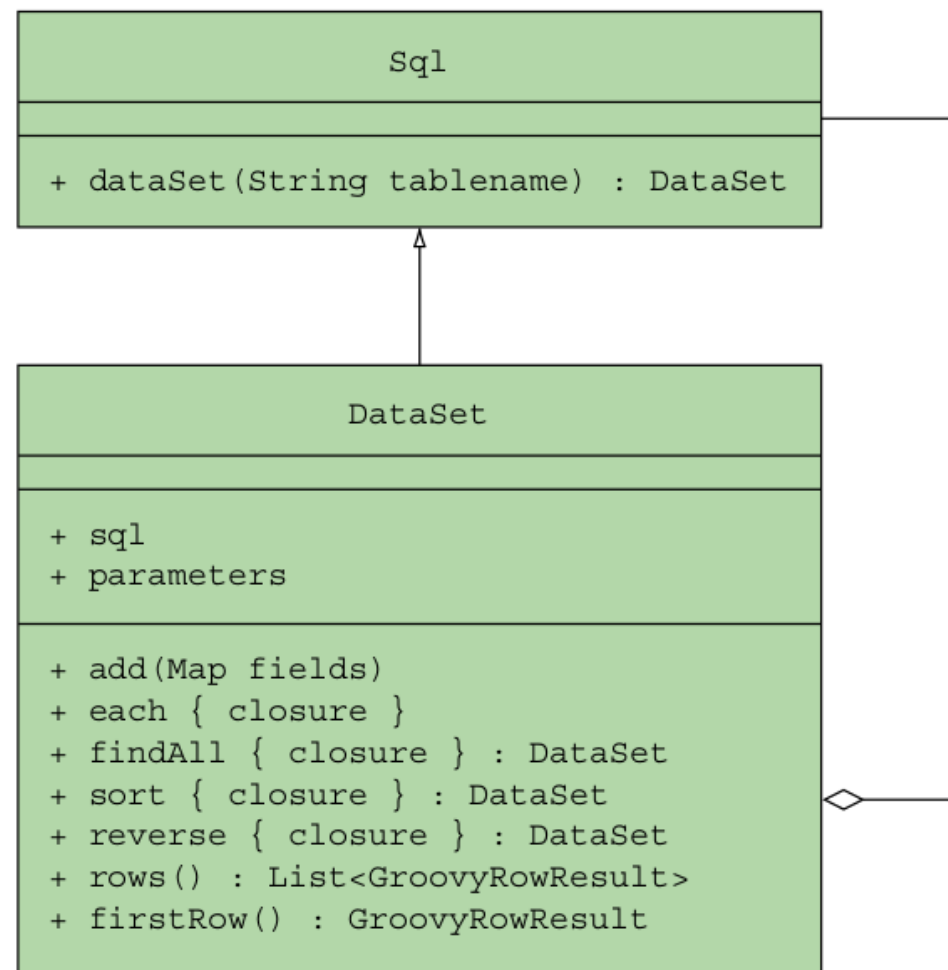
DataSet

Groovy permet de travailler avec des BD sans utiliser SQL.

Avec les ***datasets***, on peut sans utiliser SQL :

- Ajouter des lignes dans une table
- Travailler sur tous les enregistrements d'une table ou d'une vue
- Sélectionner des enregistrements avec des expressions simples

Diagramme de classe





API

```
// Passage de la table dans la factory
athletes = sql.dataSet('Athlete')
// Insertion
athletes.add(firstname:'Paula',lastname:'Radcliffe', Date: '1973-12-17')
// Parcours complet
athletes.each({ println it.firstname })
// Sélection
athletes.findAll{ it.dateOfBirth > '1970-1-1' }
// Affichage du sql et des paramètres
println athletes.sql
println athletes.parameters
```



Le GDK

Librairies cœur
Groovy SQL
Json et Groovlets



Introduction

Le module ***groovy.json*** permet de convertir des Objets en JSON via ses 2 classes :

- ***JsonSlurper*** pour le parsing
- ***JsonOutput*** pour la sérialisation



JsonSlurper

JsonSlurper est la classe qui parse un texte JSON ou un reader et qui le convertit dans des structures de données comme des maps, des lists et des types primitifs Integer, Double, Boolean et String

Elle propose la méthode ***parse()*** plus d'autres plus spécialisées comme *parseText*, *parseFile*, ...



Exemple *parseText*

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

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

def object = jsonSlurper.parseText('{ "myList": [4, 8,
  15, 16, 23, 42] }')

assert object instanceof Map
assert object.myList instanceof List
assert object.myList == [4, 8, 15, 16, 23, 42]
```



JsonOutput

JsonOutput sérialise des objets en des String JSON.

Il propose des méthodes statiques ***toJson***

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

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

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

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

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



JsonGenerator

Pour contrôler la sérialisation, on peut utiliser un ***JsonGenerator***

Le builder ***JsonGenerator.Options*** permet de spécifier différentes options

- Positionner les options
- Appeler la méthode *build()* pour créer un générateur spécialisé



Example

```
class Person {  
    String name  
    String title  
    int age  
    String password  
    Date dob  
    URL favoriteUrl  
}
```

```
Person person = new Person(name: 'John', title: null, age: 21, password: 'secret',  
                             dob: Date.parse('yyyy-MM-dd', '1984-12-15'),  
                             favoriteUrl: new URL('http://groovy-lang.org/'))
```

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

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



Convertisseur

Une *closure* utilisée pour transformer un type particulier peut être précisée dans les options de génération

- Le premier paramètre requis est un objet correspondant au type
- Le 2nd paramètre optionnel est une String correspondant au nom de la clé



Example

```
class Person {  
    String name  
    URL favoriteUrl  
}
```

```
Person person = new Person(name: 'John', favoriteUrl: new URL('http://groovy-  
lang.org/json.html#_jsonoutput'))
```

```
def generator = new JsonGenerator.Options()  
    .addConverter(URL) { URL u, String key ->  
        if (key == 'favoriteUrl') {  
            u.getHost()  
        } else {  
            u  
        }  
    }  
    .build()
```

```
assert generator.toJson(person) == '{"favoriteUrl":"groovy-  
lang.org","name":"John"}'
```



JsonBuilder, StreamingJsonBuilder

JsonBuilder propose un DSL pour spécifier une String JSON

// Un constructeur prenant un JsonGenerator en argument existe également

```
JsonBuilder builder = new JsonBuilder()
```

// Pas d'erreur de syntaxe dans le code ci-dessous !

```
builder.records {
```

```
  car {
```

```
    name 'HSV Maloo'
```

```
    make 'Holden'
```

```
    year 2006
```

```
    country 'Australia'
```

```
    record {
```

```
      type 'speed'
```

```
      description 'production pickup truck with speed of 271kph'
```

```
    }
```

```
  }
```

```
}
```

// prettyPrint formate joliment le JSON

```
String json = JsonOutput.prettyPrint(builder.toString())
```



API Rest

Pas de support particulier pour les services web

=> Framework *Grails*

OU

- Côté serveur : Utiliser le support servlet et les *Groovlets*
- Côté client
 - Utiliser l'API *java.net.URL* de bas niveau
 - Utiliser des modules Groovy d'extension ou tierces :
 - `org.codehaus.groovy.modules.http-builder`
 - `groovy-wslite`



Support Servlet

Il est tout à fait possible d'écrire des Servlets en Groovy, mais Groovy fournit la classe ***GroovyServlet*** qui simplifie la mise au point de petites applications Web.

GroovyServlet

- Compile automatiquement les fichiers *.groovy* sous la racine d'un serveur web, charge les classes et les cache tant que le fichier source n'est pas modifié.
- Fournit des variables implicites (request, response, ...) directement disponible dans le fichier source



Mise en place

1) Typiquement, il faut déclarer le servlet dans le ***web.xml*** :

```
<servlet>
  <servlet-name>Groovy</servlet-name>
  <servlet-class>groovy.servlet.GroovyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Groovy</servlet-name>
  <url-pattern>*.groovy</url-pattern>
</servlet-mapping>
```

2) Placer les librairies dans ***WEB-INF/lib***

3) Placer les fichiers Groovy sur le serveur



Exemple fichier *Groovy*

```
if (!session) {
    session = request.getSession(true)
}

if (!session.counter) {
    session.counter = 1
}

html.html { // Variable implicite html = new MarkupBuilder(out)
    head {
        title('Groovy Servlet')
    }
    body {
        p("Hello, ${request.remoteHost}: ${session.counter}! ${new Date()}")
    }
}
session.counter = session.counter + 1
```



Exemple client REST

```
def httpConnection = new URL(base + key).openConnection()
assert httpConnection.responseCode == httpConnection.HTTP_OK
def result = slurper.parse(httpConnection.inputStream.newReader())
// result contient le JSON parsé
-----
@Grab('org.codehaus.groovy.modules.http-builder:http-builder:0.7.2')
import groovyx.net.http.RESTClient
def base = 'https://issues.apache.org/jira/rest/api/latest/'
def jira = new RESTClient(base)
jira.get(path: 'issue/GROOVY-5999') { resp, json ->
    assert resp.status == 200
    json.fields.with {
        assert summary == "Make @Delegate work with @DelegatesTo"
        assert fixVersions.name == ['2.1.1']
        assert resolutiondate.startsWith('2013-02-14')
    }
}
```



Exemple Client REST (2)

```
@Grab('com.github.groovy-wslite:groovy-wslite:1.1.3')
RESTClient client = new RESTClient(BASE_URL)
client.authorization = new HTTPBasicAuthorization("user", "password")
client.defaultAcceptHeader = ContentType.JSON
def path = "/post"
def params = ["foo":1, "bar":2]
def response = client.post(path: path) {
    type ContentType.JSON
    json params
}
assert response.json?.data == params
```



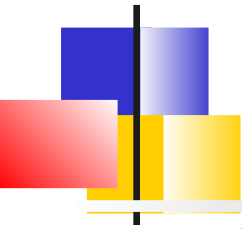
Cas d'usage

Tests

Intégrer Groovy

DSLs

Projets Connexes



Introduction

Groovy facilite le test unitaire :

- Groovy embarque ***JUnit***
=> Pas besoin de spécifier une dépendance
- Groovy propose une **classe de test** qui ajoute plein de méthodes d'assertions
- Groovy propose des facilités pour créer des **mocks ou des stubs** permettant d'isoler une classe
- Les tests écrits en Groovy peuvent facilement être exécutés par **Gradle, Maven** ou un IDE .



assert

Le mot clé ***assert*** est directement disponible. Il peut être utilisé à tout moment.

Pas besoin d'autoriser les assertions comme en Java avec `-ea`

En cas d'erreur, le reporting est beaucoup plus précis



Exemple

```
def x = [1,2,3,4,5]
assert (x << 6) == [6,7,8,9,10]
```

```
// Output:
```

```
//
```

```
// Assertion failed:
```

```
// assert (x << 6) == [6,7,8,9,10]
```

```
//
```

```
| | |
```

```
//
```

```
| | false
```

```
//
```

```
| [1, 2, 3, 4, 5, 6]
```

```
//
```

```
[1, 2, 3, 4, 5, 6]
```



Mock et Stub

La création de mocks personnalisés est beaucoup plus facile dans Groovy que dans Java.

- Simple maps ou closures peuvent suffirent

Il est toujours possible d'utiliser des frameworks spécialisés comme *Mockito*

Groovy propose les classes *MockFor* et *StubFor*



Map et Closures

En utilisant des *Maps* ou des expandos, on peut très facilement spécifier le comportement d'un collaborateur

```
class TranslationService {  
  String convert(String key) {  
    return "Complex service"  
  }  
}  
  
// as contraint la map à TranslationService.  
// Les clés sont interprétées comme des noms de méthode  
// les valeurs, (Closure), sont interprétées comme des blocs de code  
def service = [convert: { String key -> 'Mock' }] as TranslationService  
assert 'Mock' == service.convert('key.text')
```



Stub et Mock

Groovy propose les classes ***StubFor*** et ***MockFor*** qui permettent de facilement spécifier le comportement des collaborateurs d'une classe à tester (isolation des tests unitaires)

- *StubFor* vérifie optionnellement le nombre d'appels aux méthodes des collaborateurs
- *MockFor* permet en plus de vérifier la séquence des appels



Exemple StubFor

```
import groovy.mock.interceptor.StubFor

class Person {
    String first, last
}

class Family {
    Person mother, father
    def nameOfFather() { "$father.first $father.last" }
}

def stub = new StubFor(Person)
stub.demand.with {
    getLast{ 'name' }
    getFirst{ 'dummy' }
}
stub.use {
    def john = new Person(first:'John', last:'Smith')
    def f = new Family(father:john)
    assert f.nameOfFather() == 'dummy name'
}
// Appel optionnel, vérifie que getLast et getFirst ont été appelé 1 fois
stub.expect.verify()
```



Exemple *MockFor*

```
import groovy.mock.interceptor.MockFor
```

```
class Person {  
    String first, last  
}
```

```
class Family {  
    Person father, mother  
    def nameOfMother() { "$mother.first $mother.last" }  
}
```

```
def mock = new MockFor(Person)  
mock.demand.getFirst{ 'dummy' }  
mock.demand.getLast{ 'name' }  
mock.use {  
    def mary = new Person(first:'Mary', last:'Smith')  
    def f = new Family(mother:mary)  
    assert f.nameOfMother() == 'dummy name'  
}
```

```
// Appel obligatoire, vérifie que la séquence d'appel et getFirst puis getLast  
mock.expect.verify()
```



Génération de données de test

Groovy donne des facilité pour générer des données de tests via ses méthodes ***Iterable#combinations*** qui permet d'obtenir des combinaisons à partir de sous-liste

```
void testCombinations() {
    def combinations = [[2, 3],[4, 5, 6]].combinations()
    assert combinations == [[2, 4], [3, 4], [2, 5], [3, 5], [2, 6], [3, 6]]
}
// eachCombination permet d'ajouter une Closure
void testEachCombination() {
    [[2, 3],[4, 5, 6]].eachCombination { println it[0] + it[1] }
}
```



GroovyTestCase (JUnit3)

La classe **GroovyTestCase** permet :

- De nouvelles méthodes d'assertions
ex : *void shouldFail(Closure code)*
- D'exécuter des scripts Groovy comme si c'était des cas de test

```
class SimpleUnitTest extends GroovyTestCase {  
    void testSimple() {  
        assertEquals("Groovy should add correctly", 2, 1 + 1)  
    }  
    void testInvalidIndexAccess2() {  
        def numbers = [1,2,3,4]  
        shouldFail IndexOutOfBoundsException, {  
            numbers.get(4)  
        }  
    }  
}  
  
> groovy SimpleUnitTest
```




JUnit4 et 5

La classe ***GroovyAssert*** détient des méthodes statiques remplaçant celles de *GroovyTestCase*

```
import org.junit.Test
import static groovy.test.GroovyAssert.shouldFail

class JUnit4ExampleTests {

    @Test
    void indexOutOfBoundsAccess() {
        def numbers = [1,2,3,4]
        shouldFail {
            numbers.get(4)
        }
    }
}
```



Test Suites

GroovyTestSuite est une classe Java permettant d'ajouter des cas de test Groovy dans des suites *JUnit*

```
import junit.framework.*
import junit.textui.TestRunner
static Test suite() {
    def suite = new TestSuite()
    def gts = new GroovyTestSuite()
    suite.addTestSuite(gts.compile("Listing_17_02_CounterTest.groovy"))
    suite.addTestSuite(gts.compile("Listing_17_03_HashMapTest.groovy"))
    return suite
}
TestRunner.run(suite())
```

AllTestSuite permet de spécifier un répertoire de base et un motif de nom de fichier. Tous les scripts y répondant sont alors ajoutés à la suite.

(S'intègre plus facilement dans Eclipse car permet des exécutions *JUnit*)

```
def suite = AllTestSuite.suite(".", "Listing_17_*Counter*Test.groovy")
junit.textui.TestRunner.run(suite)
```



Autres aspects des tests

Certains frameworks Java liés aux tests s'intègrent avec Groovy.

Citons

- **Cobertura** qui permet de calculer la couverture des tests en prenant en compte les sources *Groovy*
- **Spock** supporte Groovy, permet de faire du BDD (*Behaviour Driven Development*)
- **Geb** orienté tests fonctionnels basé sur Selenium
- **Maven** et **Gradle** permettent de lancer les tests dans les phases de build



Cas d'usage

Tests
Intégrer Groovy
DSLs
Projets Connexes



Introduction

Groovy propose plusieurs façons de s'intégrer dans des applications Java ou même *Groovy* au moment de l'exécution.

- Exécution de code/script simple
- Intégration plus complète avec la mise en cache et la personnalisation du compilateur par exemple.



Eval

La classe ***Eval*** est le moyen le plus simple d'exécuter du code *Groovy* dynamiquement à l'exécution.

Il suffit d'appeler la méthode ***me***

```
import groovy.util.Eval
```

```
assert Eval.me('33*3') == 99
```

```
assert Eval.me('"foo".toUpperCase()') == 'F00'
```



GroovyShell

La classe ***GroovyShell*** est le moyen recommandé d'évaluer les scripts avec la possibilité de mettre en cache l'instance de script résultante.

GroovyShell offre plus d'options que Eval

```
def shell = new GroovyShell()
def result = shell.evaluate '3*5'
def result2 = shell.evaluate(new StringReader('3*5'))
assert result == result2
def script = shell.parse '3*5'
assert script instanceof groovy.lang.Script
assert script.run() == 15
```



Binding

Il est possible d'échanger des données entre l'application et le script en utilisant la classe ***Binding***¹

```
// Dans le sens appli => script
def sharedData = new Binding()
def shell = new GroovyShell(sharedData)
def now = new Date()
sharedData.setProperty('text', 'I am shared data!')
sharedData.setProperty('date', now)

String result = shell.evaluate('"At $date, $text"')

assert result == "At $now, I am shared data!"
```

1. Attention l'objet *Binding* n'est pas *ThreadSafe*



Binding

Dans le sens script => application, le script doit mettre à jour une variable **non déclarée**

```
def sharedData = new Binding()
def shell = new GroovyShell(sharedData)

shell.evaluate('foo=123')
// shell.evaluate('int foo=123') ne fonctionnerait pas
assert sharedData.getProperty('foo') == 123
```



CustomScript

GroovyShell peut utiliser une classe script personnalisée (renvoyée par la méthode *parse()*)

- Implémenter une classe qui descend de *Script*
- Positionner des options de compilation à la création de *GroovyShell*



Custom script

```
abstract class MyScript extends Script {
    String name

    String greet() {
        "Hello, $name!"
    }
}

-----
import org.codehaus.groovy.control.CompilerConfiguration

def config = new CompilerConfiguration()
config.scriptBaseClass = 'MyScript'

def shell = new GroovyShell(this.class.classLoader, new Binding(), config)
def script = shell.parse('greet()')
assert script instanceof MyScript
script.setName('Michel')
assert script.run() == 'Hello, Michel!'
```



GroovyClassLoader

Il est également possible de charger des classes et de les exécuter via ***GroovyClassLoader¹***

```
import groovy.lang.GroovyClassLoader

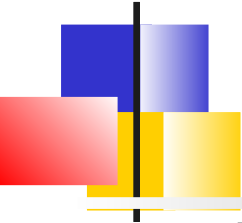
def gcl = new GroovyClassLoader()
def clazz = gcl.parseClass('class Foo { void doIt() { println
    "ok" } }')
assert clazz.name == 'Foo'
def o = clazz.newInstance()
o.doIt()
```



GroovyScriptEngine

La classe ***GroovyScriptEngine*** ajoute une couche au-dessus *GroovyClassLoader* pour gérer les problématique de dépendances des scripts et de rechargement de classes

```
def binding = new Binding()
def engine = new
    GroovyScriptEngine([tmpDir.toURI().toURL()] as URL[])
def result = engine.run('AScript.groovy', binding)
```



Bean Scripting Framework et JSR-223

Le ***Bean Scripting Framework*** fut une tentative de créer une API pour permettre d'appeler n'importe quel langages de script à partir de Java.

Il a ensuite été remplacé par l'API **JSR-223**

Groovy supporte les 2 standards



Exemple JSR-223

```
import javax.script.Invocable;

...
// Initialisation de l'API JSR-223 pour parler à Groovy
ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine engine = factory.getEngineByName("groovy");
// Exécution de script et partage de variables
engine.put("first", "HELLO");
engine.put("second", "world");
String result = (String) engine.eval("first.toLowerCase() + ' ' + second.toUpperCase()");
assertEquals("hello WORLD", result);
// Invocation de fonctions
String fact = "def factorial(n) { n == 1 ? 1 : n * factorial(n - 1) }";
engine.eval(fact);
Invocable inv = (Invocable) engine;
Object[] params = {5};
Object result = inv.invokeFunction("factorial", params);
assertEquals(new Integer(120), result);
```



Cas d'usage

Tests
Intégrer Groovy
DSLs
Projets Connexes



Introduction

Les langages structurés ne sont pas compris par les experts métiers

Les **DSLs** (*Domain Specific Languages*) sont donc mis au point pour représenter le plus naturellement possible un domaine métier et pour augmenter la qualité de communication entre l'expert métier et l'expert technique

Groovy permet de séparer les règles métier du code d'infrastructure requis pour l'exécution et d'adopter une syntaxe plus lisible et plus proche du domaine métier visé.



Chaîne de commandes

Groovy permet d'omettre les parenthèses autour des arguments d'un appel de méthode.

La fonction "**chaîne de commande**" étend cette fonctionnalité en permettant de chaîner des appels de méthode sans parenthèses, ni points entre les appels chaînés.

`a b c d <=> a(b).c(d)`

Ceci est possible avec des méthodes avec plusieurs arguments, des closures ou même des arguments nommés



Exemples

```
// Équivalent à: turn(left).then(right)
turn left then right
```

```
// Équivalent à: take(2.pills).of(chloroquine).after(6.hours)
take 2.pills of chloroquine after 6.hours
```

```
// Équivalent à: paint(wall).with(red, green).and(yellow)
paint wall with red, green and yellow
```

```
// Avec de paramètres nommés
// Équivalent à: check(that: margarita).tastes(good)
check that: margarita tastes good
```

```
// Avec des closures
// Équivalent à: given({}).when({}).then({})
given { } when { } then { }
```

```
// Méthodes sans arguments nécessite des parenthèses
// Équivalent à: select(all).unique().from(names)
select all unique() from names
```

```
// Si la chaine nombre un nombre impair, le dernier est considéré comme une propriété
// Équivalent à: take(3).getCookies()
take 3 cookies
```



Autres mécanismes

D'autres mécanismes sont utiles lors de l'élaboration d'un DSL

- Surcharge des opérateurs
- Les classes de base utilisées par les scripts
- Ajout des propriétés à des nombres
`1.minute`
- Les transformations AST
- Le mécanisme de délégation
Ex : *JsonBuilder*



Problématique

Supposons que nous aimerions créer un DSL permettant de commander les mouvements d'un robot.

Nous aimerions pouvoir spécifier les déplacements du robot par ce type de code :

move right by 3.m at 5.km/h



Apports de Groovy pour un DSL

Groovy par sa syntaxe apporte déjà plus de clarté :

- Les parenthèses peuvent en général être omises
- Les ; ne sont pas obligatoires

```
package v01
import static Direction.*
enum Direction {
    left, right, forward, backward
}
class Robot {
    void move(Direction dir) {
        println "robot moved $dir"
    }
}
def robot = new Robot()
robot.move left // <=> robot.move(left) ;
```



Séparer le code d'infrastructure

Pour améliorer le DSL, il faut séparer le code d'infrastructure du code métier :

- Le code du robot, les directions, l'instanciation du robot sont du code d'infrastructure
- Les ordres de déplacement font partie du DSL

La classe *GroovyShell* permettant d'exécuter des scripts sera utiliser pour évaluer les règles métier



Classes du domaine

```
// Les classes Robot et l'énumération des directions
// sont dans leurs propres fichiers inclus dans le classpath
package v02.model
enum Direction {
    left, right, forward, backward
}
...
package v02.model
class Robot {
    void move(Direction dir) {
        println "robot moved $dir"
    }
}
```




GroovyShell

```
package v02
def shell = new GroovyShell(this.class.classLoader)
shell.evaluate '''
import v02.model.Robot
import static v02.model.Direction.*
def robot = new Robot()
robot.move left
'''
```

Il nous reste à :

- Se débarrasser des instructions imports
- Injecter l'instance de robot
- Faciliter les envois d'ordre aux robots

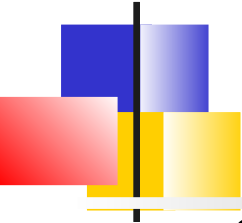


Binding

Tout script Groovy contient une map dans lequel des variables dynamiques peuvent être stockées : le ***binding***

C'est le moyen pour échanger des données avec l'extérieur

```
def binding = new Binding()
def shell = new GroovyShell(binding)
binding.setVariable('x',1)
binding.setVariable('y',3)
shell.evaluate 'z=2*x+y'
assert binding.getVariable('z') == 5
```



Application du Binding pour notre exemple

Grâce au binding, on se débarrasse dans notre fichier DSL
d'un import et de l'instanciation de la classe Robot

```
package v02
import v02.model.Robot
def binding = new Binding(
    robot: new Robot(),
    *: Direction.values().collectEntries { [(it.name()): it] }
}
def shell = new GroovyShell(this.class.classLoader, binding)
shell.evaluate '''
robot.move left
'''
```



Script de base

Pour éviter d'avoir à préfixer les appels de méthodes par *robot*, il faudrait que la methode *move* soit implémentée dans la classe *Script*

Un solution plus élégante consiste à utiliser une classe script de base personnalisée qui positionne l'objet délégué à l'instance de robot

```
package v02.integration
import v02.model.Direction
abstract class RobotBaseScript extends Script {
// @Lazy est nécessaire car les bindings s'appliquent
// après l'instanciation
@Delegate @Lazy Robot robot = this.binding.robot
}
```



@BaseScript

L'annotation **@BaseScript** permet de définir la classe de Script utilisée.

```
package v02
import v02.model.*
def binding = new Binding(
    robot: new Robot(),
    *: Direction.values().collectEntries { [(it.name()): it] }
)
def shell = new GroovyShell(this.class.classLoader, binding)
shell.evaluate '''
@BaseScript(v02.integration.RobotBaseScript)
import groovy.transform.BaseScript
move left
'''
```



Ajout automatique d'imports

Le constructeur de *GroovyShell* prend un paramètre de type

CompilerConfiguration

Ce paramètre permet de définir

- des personnalisations du compilateur :
 - En particulier, l'ajout automatique d'import
 - ...
- Une classe de Script de base



Ajout automatique d'import et script de base

```
package v02

import org.codehaus.groovy.control.CompilerConfiguration
import org.codehaus.groovy.control.customizers.*
import v02.integration.RobotBaseScript
import v02.model.*

def binding = new Binding(robot: new Robot())
def importCustomizer = new ImportCustomizer()
importCustomizer.addStaticStars Direction.name
def config = new CompilerConfiguration()
config.addCompilationCustomizers importCustomizer
config.scriptBaseClass = RobotBaseScript.name

def shell = new GroovyShell(this.class.classLoader, binding, config)
shell.evaluate '''
move left
'''
```



Amélioration

Nouvel objectif :

- Ajout d'un paramètre sur la méthode *move*
`move right, 3.meters`
- Ou même
`move right, by : 3.meters`

=> Ajouter une propriété aux nombres !

Solution :

- les *Category* : modifier la meta-class de number pour une petite portion de code
- Utilisation des paramètres nommés



Nouvelle classe du modèle

--- Classe DistanceUnit

```
enum DistanceUnit {  
    centimeter('cm', 0.01),  
    meter( 'm',1),  
    kilometer ('km', 1000)  
    String abbreviation  
    double multiplier  
    DistanceUnit(String abbr, double mult) {  
        this.abbreviation = abbr  
        this.multiplier = mult  
    }  
    String toString() { abbreviation }  
}
```

--- Classe Distance

```
import groovy.transform.TupleConstructor  
@TupleConstructor  
class Distance {  
    Number amount  
    DistanceUnit unit  
    String toString() { "$amount$unit" }  
}
```

--- Classe Robot avec paramètres nommés Map pouvant contenir les clés *by* et *at*

```
void move(Map m, Direction dir) {  
    println "robot moved $dir by $m.by at ${m.at ?: '1 km/h'}"  
}
```



Classe *Category*

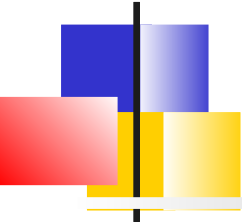
```
package v03.integration
import v03.model.*
class DistanceCategory {
    static Distance getCentimeters(Number num) {
        new Distance(num, Unit.centimeter)
    }
    static Distance getMeters(Number num) {
        new Distance(num, Unit.meter)
    }
    static Distance getKilometers(Number num) {
        new Distance(num, Unit.kilometer)
    }
    static Distance getCm(Number num) { getCentimeters(num) }
    static Distance getM(Number num) { getMeters(num) }
    static Distance getKm(Number num) { getKilometers(num) }
}
```



Résultat

```
package v02
import org.codehaus.groovy.control.*
import v02.integration.*
import v02.model.*
def binding = new CustomBinding(robot: new Robot())
def config = new CompilerConfiguration()
config.scriptBaseClass = CaseRobotBaseScript.name
Specifies script
base class
def shell = new GroovyShell(this.class.classLoader, binding, config)

use(DistanceCategory) {
shell.evaluate '''
    move left
    move right, by: 3.meters, at: 5.km/h
'''
}
```



Cas d'usage

Tests
Intégrer Groovy
DSLs
Projets Connexes



Eco-système Groovy

Groovy Grapes : Ajoute des dépendances Maven au classpath

Spock : Framework de test à partir de spécification

Scriptom : Permet de manipuler facilement des objets COM et ActiveX.

GroovyServ : Groovy en mode client serveur pour accélérer l'exécution de scripts

Graddle : Système de build utilisant les conventions de build et des tâches spécifiques

CodeNarc : Analyse de code Groovy, détection de bugs

GContracts : Apporte le concepts de design-by-contract à Groovy

Grails : Applications web en Groovy (MVC, Spring, Hibernate, SiteMesh)

Griffon : Framework de développement d'application desktop

Gaelyk : Framework pour exécuter des Groovlets et des templates Groovy en utilisant Google App Engine



Merci!!!

❖ MERCI DE VOTRE ATTENTION

« **Groovy in Action** »,
Dierk König, Paul King
Manning Publications