





Groovy pour les pipeline Jenkins

David THIBAU - 2024

david.thibau@gmail.com



Agenda

- Présentation

- Groovy et Java
- Un bref tour de Groovy
- Installation

- Les bases de Groovy

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

- Pipeline Jenkins

- Approche et concepts
- Syntaxe déclarative
- Groovy et Syntaxe script
- Bibliothèques partagées



Groovy et Java

Un bref tour de Groovy

Installation



Groovy et la JVM

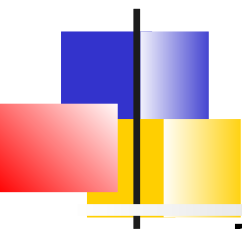
Groovy est un langage agile et dynamique pour la JVM.
Il ressemble à Java mais simplifie certains éléments de syntaxe

Au final, le code Groovy est transformé en bytecode Java

- Soit dynamiquement à l'exécution
- Soit statiquement à la compilation

=> S'intègre à toutes les classes et bibliothèques Java existantes

=> Courbe d'apprentissage presque nulle pour développeurs Java



Cas d'usage

Facilite l'écriture de shell et de scripts.
Équivalent à Python

Intégration de script dans des outils Java.
Exemple JMeter, JasperReport, ...

Permet le développement rapide de petites applications Web
(UI, BD, ...)

Supporte les syntaxes compactes et permet la mise en place
de DSLs.
Exemple : *Jenkinsfile*, *build.gradle*



Ressemblances Java

- Les commentaires : // OU /* ...*/
- Les packages : Organisation des sources en répertoires
- Syntaxe : Instructions, Structures de contrôle, Opérateurs, Expressions et Affectations
- Orientation Objet : Classes, Interfaces, les *enum*, attributs et méthodes, Classes imbriquées.
- Le traitement des exceptions : Construction try, catch, finally
- Les génériques et les annotations : <> ET @
- L'instanciation d'objet : *new*



GDK

Le ***GDK*** est une extension du JDK :

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

Le *GDK* est organisé en modules (Web, BD, XML, JSON, YML...)



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 |



Optionnellement typé

Groovy est optionnellement typé

Le mot clé **def** est utilisé pour définir une variable ou une fonction non typée.

Groovy décide des types lors de l'exécution en fonction des valeurs attribuées.

```
// listOfCountries est une ArrayList
```

```
def listOfCountries = ['USA', 'UK', 'FRANCE', 'INDIA']
```

```
// La fonction multiply peut renvoyer n'importe quel objet
```

```
// En fonction des paramètres x et y
```

```
def multiply(x, y) {  
    return x*y  
}
```



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.



Définition de classes

Une classe comporte des **champs** et des **propriétés**, i.e. champ accessible par des méthodes *get/set*, ainsi que des méthodes.

Les constructeurs sont des méthodes particulières permettant d'instancier la classe

On peut appliquer des *modifiers* de visibilité sur les composants de la classe. (*private*, *public*)

- les classes et les méthodes sans *modifier* sont *public* par défaut
- Les champs sans *modifiers* deviennent des propriétés
- 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 ;  
  
    // Constructeur  
    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
- 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'
```

```
def getTitleBackwards(book) {  
    String title = book.getTitle()  
    title.reverse()  
}
```




Gstring et regexp

En *Groovy*, les littéraux String peuvent utiliser les simples, doubles ou triples 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
```

```
// Boucle for sur un intervalle
```

```
def clinks = 0
```

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

```
    clinks += remainingGuests
```

```
}
```

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



Structures de contrôle (2)

// Boucle for sur une liste

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

// Appel de la méthode 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.



Commandes d'exécution

3 commandes permettent d'exécuter du code *Groovy* :

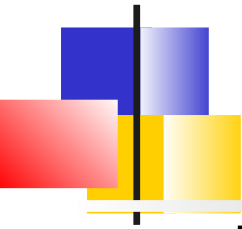
- **groovy** : Exécution d'un script *Groovy*.
- **groovysh** : Shell groovy permettant l'exécution interactive.
- **groovyConsole** : Interface graphique pour exécution interactive et chargement de script



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

Compléments



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> | |

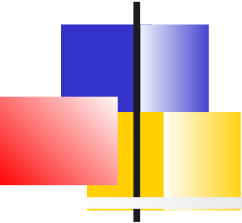


Type safe

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`



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')
```



Chaînes de caractères

Groovy permet différentes options pour les littéraux :

- Les **simple-quotes** ' : équivalent à *String* Java
- Les **double-quotes** " , permettent l'utilisation d'expression via **\$** (placeholder). Ce sont des *GString*
- Les **triples (simple ou double) quotes** ''' ou """" permettent du texte sur plusieurs lignes¹ .
- Le **/** ou **\$/** permet de ne pas échapper les caractères particulier (comme le backslash par exemple)
=> Pratique pour les expressions régulières



Examples

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



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'
```



regexp

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

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

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

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



Usage

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

- D'indiquer si le motif correspond à la chaîne 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 '
```



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 '
```




Les bases de Groovy

Types de données

Collections

Orientation objet

Closures

Compléments



Introduction

Groovy hérite des Collections Java, en particulier :

- **List** accessible via un index
- **Map** accessible via une clé

Il ajoute **Range** définissant un intervalle.

Il facilite leur utilisation

- Les *List*, *Map* et *Range* peuvent être déclarés par des littéraux.
- Ils ont des opérateurs spécialisés



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* (Opérateurs ++ et --).
- Implémentent *compareTo* (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)
// Test des limites
assert (0.0..1.0).contains(1.0)
assert (0.0..1.0).containsWithinBounds(0.5)
// 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 sur une Range

```
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 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 permet de facilement lire ou écrire un sous-ensemble de la collection.

```
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']
```




Map (1)

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

Elles peuvent être initialisées comme
suit :

```
def MyMap = [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 condition donnée indiquée via une *closure* .

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



Boucles

```
def myMap = [a:1, b:2, c:3]

// Boucle sur les entrees
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'
```



Les bases de Groovy

Types de données

Collections

Orientation objet

Closures

Compléments



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 le programme l'appelant)

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

- La visibilité par défaut définit une propriété ;
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 }
```

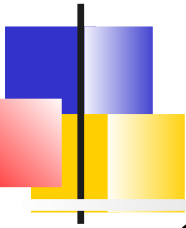
Attention les parenthèses sont obligatoires lors d'appels imbriqués :

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



Exemples : Usage des paramètres

```
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

Si la méthode prend une *Map* en paramètre, les appels peuvent se faire en nommant les paramètres qui deviennent les cls de la Map

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



Opérateur Elvis ?.

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 sources en packages

- 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 rechercher 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



Les bases de Groovy

Types de données
Collections
Orientation objet
Closures
Compléments



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 Closure : les accolades :

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

Ressemble au Lambda expressions de Java8



Variable *it*

Lorsqu'il n'y a qu'un seul paramètre passé à la *closure*, sa déclaration est optionnelle.

La variable *it* peut alors être utilisée.

```
log = ''
```

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

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

// Version longue

```
log = ''
```

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

// Déclaration par affectation

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



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  
}
```



Appel d'une closure

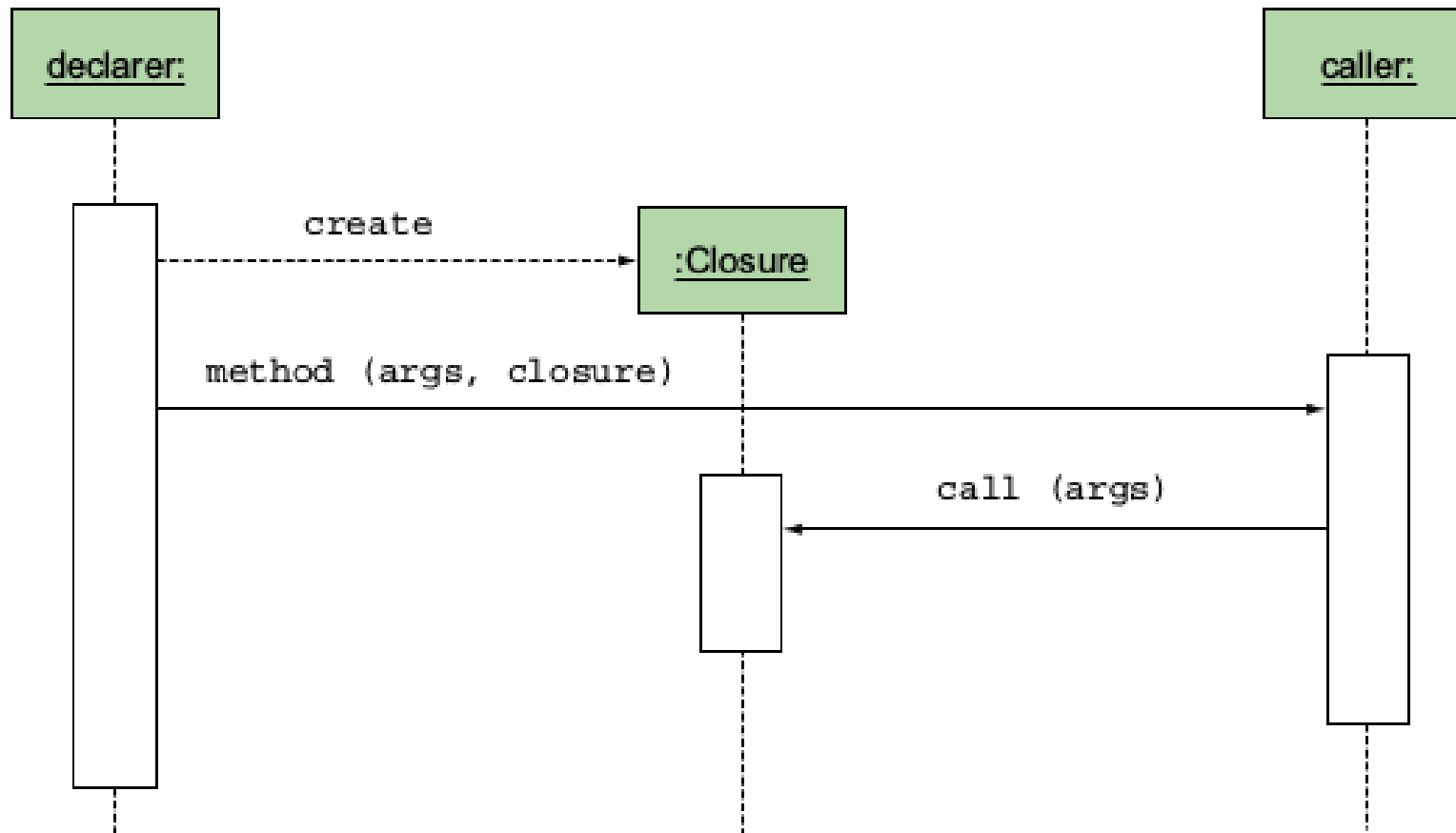
On peut appeler une closure `x` par `x.call()` ou simplement `x()`

```
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 en argument 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 le transmettre 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.



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) : Si une propriété/méthode existe dans le propriétaire, elle est utilisé ; sinon le délégué est utilisé
- *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]
```



Les bases de Groovy

Types de données

Collections

Orientation objet

Closures

Compléments



Annotations

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

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

Les annotations permettent principalement l'ajout de méthodes ou de bloc de code



Exemple *@Grab*

@Grab permet de charger des librairies externes.

A la compilation ou à l'exécution, la librairie est téléchargée si besoin puis 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)
```



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*



Exemples

```
// Parcours de répertoire
def srcDir = new File('./src')
dirs = []
directory names
topDir.eachDirRecurse { dirs << it.name }
assert dirs.containsAll(['gradle', 'src', 'main'])
assert dirs.containsAll(['groovy', 'services', 'wrapper'])
// Lecture de petit fichier
println new File('data/example.txt').text
// Ecriture de fichier
def outFile = new File('data/example.txt')
def lines = ['line one', 'line two', 'line three']
outFile.write(lines[0..1].join("\n"))
outFile.append("\n"+lines[2])
// Transformation et filtre
reader = new File('../data/example.txt').newReader()
writer = new StringWriter()
reader.transformLine(writer) { it - 'line' }
assert " one${\n} two${\n} three${\n}" == writer.toString()
...
```



Intégration

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
```



Pipelines

Approche et concepts

Syntaxe déclarative

Groovy et Syntaxe script

Librairies partagées



Introduction

Jenkins Pipeline est une **suite de plugins** qui permettent d'implémenter et d'intégrer des pipelines de CI/CD.

Grâce à Pipeline, les processus de build sont modélisés **via du code et un langage spécifique (DSL)**



Définir une pipeline

Une Pipeline peut être créée :

- En saisissant un script directement dans l'interface utilisateur de Jenkins.
- En créant un fichier ***Jenkinsfile*** qui peut alors être enregistré dans le SCM.

Approche recommandée

Quelque soit l'approche, 2 syntaxes sont disponibles :

- Syntaxe déclarative
- Syntaxe script



Avantages de l'approche

Pipeline offre de nombreux avantages :

- *Build As Code* : Les pipelines implémentées par du code sont gérées par le SCM
=> Historique des révisions, branches, ...
- Le DSL supporte des pattern de workflow complexes (fork/join, boucle, ...)
- Les Pipelines peuvent s'arrêter et attendre une approbation manuelle, survivent au redémarrage de Jenkins
- Il est possible de profiter de toute la puissance d'un langage de script comme Groovy en particulier de librairies partagées
- Les plugins Jenkins permettent d'étendre le DSL en proposant de nouvelles fonctions facilitant l'intégration



Termes du DSL

Le DSL introduit plusieurs termes et concepts :

- **node ou agent** : Les travaux d'une pipeline sont exécutés dans le contexte d'un nœud ou agent.
 - Plusieurs nœuds peuvent être déclarés dans une pipeline.
 - Chaque nœud a son propre espace de travail
- **stage (phase)** : Une phase définit un sous ensemble de la pipeline.
Par exemple : "Build", "Test", et "Deploy".
Les phases améliorent la compréhension et la visualisation de l'avancement de la pipeline
- **step (étape)** : Une tâche unitaire Jenkins.
Par exemple, exécuter un shell, publier les résultats de test



Exemple *Jenkinsfile*

```
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
        sh './mvnw -Dmaven.test.failure.ignore=true clean test'
      } post {
        always { junit '**/target/surefire-reports/*.xml' }
      }
    }
    stage('Parallel Stage') {
      parallel {
        stage('Intégration test') {
          agent any
          steps {
            sh './mvnw clean integration-test'
          }
        }
        stage('Quality analysis') {
          agent any
          steps {
            sh './mvnw clean verify sonar:sonar'
          }
        }
      }
    }
  }
}
```



Jobs pipeline

Le plugin Pipeline ajoute de nouveaux types de Jobs :

- **Pipeline** : Définition d'une pipeline in-line ou dans un Jenkinsfile
- **Multi-branch pipeline** : On indique un dépôt et Jenkins scanne toutes les branches à la recherche de fichier Jenkinsfile. Un job est démarré pour chaque branche
- **Bitbucket/Team, Github** : On indique un compte et Jenkins scanne toutes les branches de tous les projet du serveur Bitbucket ou Github à la recherche de Jenkinsfile. Il démarre un job pour chaque Jenkinsfile trouvé



Variables d'environnement additionnelles

Les builds d'une pipeline multi-branches ont accès à des variables additionnelles :

- **BRANCH_NAME** : Nom de la branche pour laquelle la pipeline est exécutée
- **CHANGE_ID** : Identifiant permettant d'identifier le changement ayant provoqué le build

Les builds d'une pipeline multi-projet ont accès à des variables additionnelles identifiant le projet Github ou Bitbucket



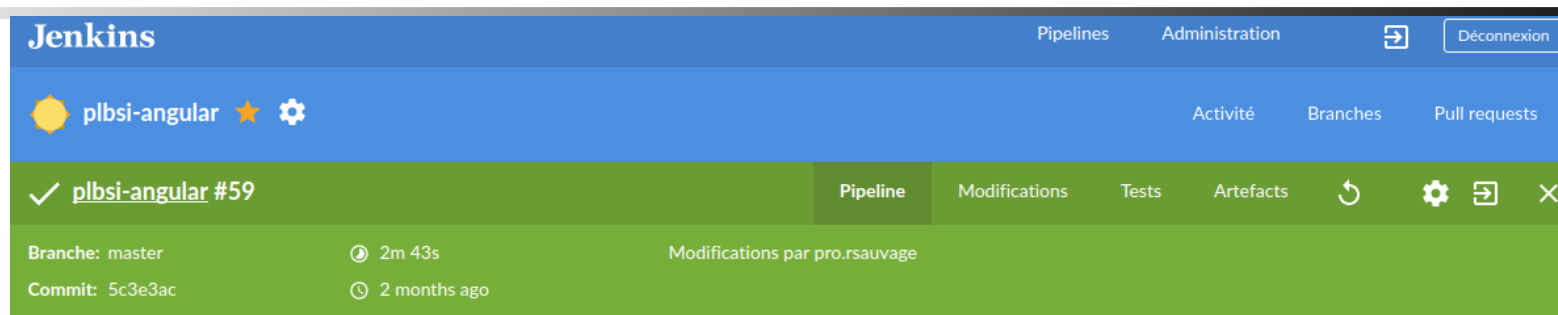
Blue Ocean

Le plugin ***Blue Ocean*** propose une interface utilisateur dédiée aux pipelines :

- Visualisation graphique des pipelines
- Éditeur graphique de pipeline
- Intégration des branches et pull-request

Cette interface cohabite avec l'interface classique

Exemple : Détail d'un build



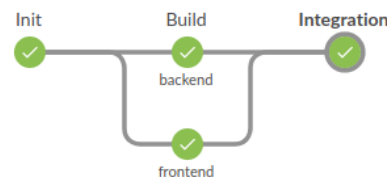
The Jenkins UI header shows the 'plbsi-angular' project. Below it, the build details for 'plbsi-angular #59' are displayed. The build is successful (green checkmark) and is on the 'master' branch. It was committed by 'pro.rsauvage' 2 months ago. The build duration is 2m 43s. The build is triggered by 'Modifications par pro.rsauvage'.

plbsi-angular #59

✓ plbsi-angular #59

Branche: master 2m 43s Modifications par pro.rsauvage

Commit: 5c3e3ac 2 months ago



Étapes - Integration

| | | |
|---|------------------------------------|-----|
| ✓ | > Restore files previously stashed | <1s |
| ✓ | > Shell Script | 27s |
| ✓ | > Shell Script | <1s |
| ✓ | > Shell Script | <1s |
| ✓ | > Shell Script | <1s |
| ✓ | > Shell Script | <1s |
| ✓ | > Shell Script | 2s |



Pipelines

Approche et concepts

Syntaxe déclarative

Groovy et Syntaxe script

Librairies partagées



Documentation

La documentation est incluse dans Jenkins. Elle est accessible à *localhost:8080/pipeline-syntax/*

Les utilitaires « ***Snippet Generator*** » et « ***Declarative Directive Generator*** » sont des assistants permettant de générer des fragments de code

- *Declarative Directive Generator* n'est valable que pour la syntaxe déclarative
- *Snippet Generator*, permettant de générer les steps, est valable pour les 2 syntaxes



Declarative Directive Generator

Sample Directive

agent: Agent

See [the online documentation](#) for more information on the agent directive.

Agent

label: Run on an agent matching a label

Label

jdk8

Generate Declarative Directive



Snippet Generator

Sample Step

archiveArtifacts: Archive the artifacts

archiveArtifacts

Files to archive

Generate Pipeline Script



Références Variables globales

En plus des générateurs, Jenkins fournit un lien vers le « ***Global Variable Reference*** » qui est également mis à jour en fonction des plugins installés.

Le lien documente les variables directement utilisables dans les pipelines

Par défaut, *Pipeline* fournit les variables suivantes :

- ***env*** : Variables d'environnement.
Par exemple : *env.PATH* ou *env.BUILD_ID*.
- ***params*** : Tous les paramètres de la pipeline dans une Map.
Par exemple : *params.MY_PARAM_NAME*.
- ***currentBuild*** : Encapsule les données du build courant.
Par exemple : *currentBuild.result*, *currentBuild.displayName*



Généralités

La syntaxe déclarative est recommandée car plus simple d'accès et plus lisible

Toutes les pipelines déclaratives sont dans un bloc ***pipeline***.

Elles contiennent toujours les **sections** suivantes :

- ***stages*** : Bloc contenant un ou plusieurs bloc *stage*
- ***stage*** : Bloc contenant un bloc *steps* et éventuellement un bloc *post*
- ***steps*** : Bloc contenant une succession de step unitaire

Des **directives** sont placées soit au niveau pipeline soit au niveau d'un stage. Par exemple : *agent*

Éventuellement, des sections ***post*** spécifient les actions de post build



Structure

```
pipeline {  
  // Directives s'appliquant à toute la pipeline  
  stages {  
    stage('Compile et tests') {  
      // Directives ne s'appliquant qu'au stage  
      steps {  
        // Steps unitaires  
        echo 'Unit test et packaging'  
  
      } post { // Les actions de post-build  
        // Les différentes issues du build (stable, unstable, success, ...)  
      }  
    }  
    stage('Une autre phase') {  
      ....  
    }  
  }  
}
```



Stages et steps

Les sections stages et steps ne font que délimiter un bloc

stages :

- pas de paramètres spécifiques.
- Englobe plusieurs blocs *stage*

stage :

- Un nom
- des directives appliquées au *stage*
- un bloc steps

steps :

- Pas de paramètre
- A l'intérieur de chaque *stage*
- Englobe plusieurs actions unitaires



Section *post*

La section ***post*** définit une ou plusieurs *steps* qui sont exécutées en fonction du statut du build

- *always* : Étapes toujours exécutées
- *changed* : Seulement si le statut est différent du run précédent
- *failure* : Seulement si le statut est *échoué*
- *success* : Seulement si statut est *succès*
- *unstable* : : Seulement si statut *instable* (Tests en échec, Violations qualité, porte perf., ..)
- *aborted* : Build avorté



Exemple

```
stage('Compile et tests') {
    agent any
    steps {
        echo 'Unit test et packaging'
        sh 'mvn -Dmaven.test.failure.ignore=true clean package'
    }
    post {
        always {
            junit '**/target/surefire-reports/*.xml'
        }
        success {
            archiveArtifacts artifacts: 'application/target/*.jar', followSymlinks: false
        }
        failure {
            mail bcc: '', body: 'http://localhost:8081/job/multi-branche/job/dev', cc: '',
from: '', replyTo: '', subject: 'Packaging failed', to: 'david.thibau@gmail.com'
        }
    }
}
```



Directives

Les directives se placent généralement soit

- Sous le bloc pipeline
=> Il s'applique à tous les *stages* de la pipeline
- Sous un bloc stage
=> Il ne s'applique qu'au *stage* concerné



Directive *agent*

La directive ***agent*** supporte les paramètres suivants :

- ***any*** : N'importe quel agent.
- ***label*** : Agent ayant été labellisé par l'administrateur
- ***none*** : Aucun.
 - Permet de s'assurer qu'aucun agent ne sera alloué inutilement.
 - Placer au niveau global, force à définir un agent au niveau de stage
- ***docker, dockerfile*** : Si plugin docker présent
- ***kubernetes*** : Si plugin Kubernetes présent



Directive *tools*

tools permet d'indiquer les outils à installer sur l'exécuteur ou agent.

La section est ignorée si *agent none*

Les outils supportés sont ceux installés par l'administrateur Jenkins



Exemples

```
// Directive,  
// agent avec le label jdk8  
agent {  
    label 'jdk8'  
}
```

```
// Mise à disposition d'un outil sur l'agent  
  
agent any  
tools {  
    maven 'Maven 3.5'  
}
```



Directive *environment*

La directive ***environment*** spécifie une séquence de paires clé-valeur qui seront définies comme variables d'environnement pour le stage .

- La directive supporte la méthode ***credentials()*** utilisée pour accéder aux crédits définis dans Jenkins.

```
pipeline {  
    agent any  
  
    environment {  
        NEXUS_CREDENTIALS = credentials('jenkins_nexus')  
        NEXUS_USER = "${env.NEXUS_CREDENTIALS_USR}"  
        NEXUS_PASS = "${env.NEXUS_CREDENTIALS_PSW}"  
    }  
}
```



Directive *options*

La directive ***options*** permet de configurer des options globale à la pipeline.

Par exemple, *timeout*, *retry*, *buildDiscarder*, ..

Exemple :

```
pipeline {  
  agent any  
  options { timeout(time: 1, unit: 'HOURS') }  
  stages {  
    stage('Example') {  
      steps { echo 'Hello World'}  
    }  
  }  
}
```




Directives

input et *parameters*

La directive ***input*** permet de stopper l'exécution d'une pipeline et d'attendre une saisie manuelle d'un utilisateur

Via la directive ***parameters***, elle peut définir une liste de paramètres à saisir.

Chaque paramètre est défini par :

- Un type : String ou booléen, liste, ...
- Une valeur par défaut
- Un nom (Le nom de la variable disponible dans le script)
- Une description



Example : input

```
input {  
  message "Should we continue?"  
  ok "Yes, we should."  
  submitter "alice,bob"  
  parameters {  
    string(name: 'PERSON', defaultValue: 'Mr Jenkins', description:  
    'Who should I say hello to?')  
  }  
}  
steps {  
  echo "Hello, ${PERSON}, nice to meet you."  
}
```



Directive *triggers*

La directive ***triggers*** définit les moyens automatique par lesquels la pipeline sera redéclenchée.

Les valeurs possibles sont *cron*, *pollSCM* et *upstream*



Directive *when*

La directive ***when*** permet à Pipeline de déterminer si le stage doit être exécutée

La directive doit contenir au moins une condition.

Si elle contient plusieurs conditions, toutes les conditions doivent être vraies. (Équivalent à une condition *allOf* imbriquée)



Conditions imbriquées disponibles

branch : Exécution si la branche correspond au pattern fourni

environnement : Si la variable d'environnement spécifié à la valeur voulue

expression : Si l'expression Groovy est vraie

not : Si l'expression est fausse

allOf : Toutes les conditions imbriquées sont vraies

anyOf : Si une des conditions imbriquées est vraie



Example

```
stage('Example Deploy') {  
  when {  
    allof {  
      branch 'production'  
      environment name: 'DEPLOY_TO',  
                   value: 'production'  
    }  
  }  
  steps {  
    echo 'Deploying'  
  }  
}
```



Directive *parallel*

Avec la directive ***parallel***, les *stages* peuvent déclarer des *stages* imbriqués qui seront alors exécutés en parallèle

- Les *stages* imbriqués ne peuvent pas contenir à leur tour de *stages* imbriqués
- Le *stage* englobant ne peut pas définir *d'agent* ou de *tools*



Example

```
// Declarative //
pipeline {
  agent any
  stages {
    stage('Parallel Stage') {
      when {branch 'master' }
      parallel {
        stage('Branch A') {
          agent { label "for-branch-a" }
          steps { echo "On Branch A" }
        }
        stage('Branch B') {
          agent { label "for-branch-b" }
          steps { echo "On Branch B" }
        }
      }
    }
  }
}
```




Steps

Les steps disponibles sont extensibles en fonction des plugins installés.

Voir la documentation de référence à :
<https://jenkins.io/doc/pipeline/steps/>

A noter que la version déclarative a une step ***script*** qui peut inclure un bloc dans la syntaxe script



Steps

// Exécuter un script

sh, bat

// Copier des artefacts

```
copyArtifacts(projectName: 'downstream', selector: specific("$  
    {built.number}"));
```

// Archive the build output artifacts.

```
archiveArtifacts artifacts: 'output/*.txt', excludes: 'output/*.md'
```

// Step basiques

stash, unstash : Mettre de côté puis reprendre

dir, deleteDir, pwd : Positionner le répertoire courant, supprimer un répertoire

fileExists, writeFile, readFile

mail, git, build, error

sleep, timeout, waitUntil, retry

withEnv, credentials, tools

cleanWs() : Nettoyer le workspace

Voir : <https://jenkins.io/doc/pipeline/steps/>



Lint

La syntaxe autorisée dans Jenkinsfile dépend des plugins installés sur le serveur.

Pour valider la syntaxe d'un Jenkinsfile, Jenkins propose un Linter accessible via Jenkins CLI ou l'API Rest

Exemple :

```
# ssh (Jenkins CLI)
# JENKINS_SSHD_PORT=[sshd port on master]
# JENKINS_HOSTNAME=[Jenkins master hostname]
ssh -p $JENKINS_SSHD_PORT $JENKINS_HOSTNAME declarative-linter <
Jenkinsfile
```

Des extensions d'IDE proposent une validation du Jenkinsfile via l'API Rest de Jenkins



Rejouer une pipeline

Sur un build exécuté, le lien "**Replay**" permet de le rejouer en apportant des modifications (sans changer la configuration de la pipeline et sans committer)

Après plusieurs essais, il est possible de récupérer les modifications pour les committer



Pipelines

Approche et concepts

Syntaxe déclarative

Groovy et Syntaxe script

Librairies partagées



Introduction

Une pipeline scriptée est un DSL construit avec Groovy.

- => La plupart des fonctionnalités du langage Groovy sont disponibles rendant l'outil très flexible et extensible

La syntaxe script n'est pas recommandée car moins lisible.

On l'utilise cependant assez souvent dans un bloc ***script*** à l'intérieur d'une pipeline déclarative



Exemple bloc script

```
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
                script {
                    def browsers = ['chrome', 'firefox']
                    for (int i = 0; i < browsers.size(); ++i) {
                        echo "Testing the ${browsers[i]} browser"
                    }
                }
            }
        }
    }
}
```



Déclaration de variables

En Groovy, il n'est pas nécessaire de préciser le type d'une variable lors de sa déclaration.

Il suffit d'utiliser le mot-clé ***def***

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




String

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

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

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



Collections

Groovy facilite la manipulation des collections en ajoutant des opérateurs, des instanciations via des littéraux

L'accès aux éléments est cohérent quelque soit le type de la collection (*List*, *Map*, ...)

Il introduit également un nouveau type : *Range* avec la notation ..



Exemples Collection

```
// 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 facilement instanciées
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.

Un objet de type *Closure* travaille en sous-main

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

```
// Variable implicite it
```

```
[1, 2, 3].each { println it }
```



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

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



Limitations Groovy

Les pipeline, pour pouvoir survivre à des redémarrage, doivent sérialiser leurs données vers le master.

Ainsi certaines constructions Groovy ne sont pas supportées. Par exemple :

```
collection.each {  
    item → /* perform operation */  
}
```



Exécuteur

En mode script, la fonction *node()* permet de provisionner un agent

```
// Script
// Un nœud esclave taggé 'Windows'
node('Windows') {
    // some block
}
```



Examples

```
node {  
  stage('Example') {  
    if (env.BRANCH_NAME == 'master') {  
      echo 'I only execute on the master branch'  
    } else {  
      echo 'I execute elsewhere'  
    }  
  }  
}
```




Checkout du dépôt

A la différence de la directive `agent`, *node* ne provoque pas le checkout du dépôt

// Checkout branche master de Git

```
checkout([$class: 'GitSCM', branches: [[name: '*/master']],
  doGenerateSubmoduleConfigurations: false, extensions: [],
  submoduleCfg: [], userRemoteConfigs: [[url:
    '/home/dthibau/Formations/MavenJenkins/MyWork/weather-
    project']]])
```

// Plus simple, clone du repo :

```
git '/home/dthibau/Formations/MavenJenkins/MyWork/weather-
project'
```

// Positionner la clé de hash dans une variable

```
gitCommit = sh(returnStdout: true, script: 'git rev-parse
HEAD').trim()
```



Plugin Utility Steps

Le plugin ***Pipeline Utility Steps*** ajoute plein d'étapes utilitaires comme :

- Trouver un fichier dans le workspace
- Créer ou vérifier un SHA-1
- Créer des tar gz, des zip
- Lire des fichiers CSV, properties, YAML, JSON
- ...



Exécution //

// Exemple script

```
stage('Test') {  
  parallel linux: {  
    node('linux') {  
      checkout scm  
      try {  
        unstash 'app'  
        sh 'make check'  
      }  
      finally {  
        junit '**/target/*.xml'  
      }  
    }  
  },  
  windows: {  
    node('windows') {  
      /* .. snip .. */  
    }  
  }  
}
```



Exemple complet

```
#!/groovy

stage('Init') {
    node {
        git 'file:///home/dthibau/Formations/MyWork/MyProject/.git'
        echo 'Pulling...' + env.BRANCH_NAME
        sh(returnStdout: true, script: 'git checkout '+ env.BRANCH_NAME)
        gitCommit = sh(returnStdout: true, script: 'git rev-parse HEAD').trim()
    }
}

stage('Build') {
    parallel frontend : {
        node {
            checkout([$class: 'GitSCM',branches: [[name: gitCommit ]],userRemoteConfigs: [[url:
'file:///home/dthibau/Formations/MyWork/MyProject/']]])
            dir("angular") {
                sh 'nvm v9.5.0'
                sh 'ng build -prod' }
            dir ("angular/dist") {
                stash includes: '**', name: 'front'}
        }}, backend : {
        node {
            checkout([$class: 'GitSCM', branches: [[name: gitCommit ]], userRemoteConfigs: [[url:
'file:///home/dthibau/Formations/MyWork/MyProject']]])
            sh 'mvn clean install'
        } } }
}
```



Pipelines

Approche et concepts
Syntaxe déclarative
Groovy et Syntaxe script
Librairies partagées



Introduction

Pipeline permet la création de **librairies partagées** pouvant être définies dans des dépôts de sources externes et chargées lors de l'exécution d'une Pipelines.

Une librairie est constituée de fichiers Groovy



Étapes de mise en place

La mise en place consiste en :

- 1) Créer les scripts groovy en respectant une arborescence projet et committer dans un dépôt
- 2) Définir la librairie dans Jenkins
Administrer Jenkins → Shared Libraries :
 - Un nom
 - Une méthode de récupération
 - Une version par défaut
- 3) L'importer dans un projet en utilisant l'annotation ***@Library***



Code groovy

Différents types de codes peuvent être développés dans une librairie :

- Classes groovy classique, définissant des structures de données, des méthodes.
Pour les utiliser, il faudra les instancier ;
Pour interagir avec les variables de la pipeline (env par exemple), il faudra les passer en paramètre.
Utilisable dans pipeline script
- Définir des variables globales. Jenkins les instancie automatiquement comme singleton et elles apparaissent dans l'aide.
Utilisable dans pipeline script
- Définir des nouvelles steps. Idem variable globale + mise à disposition de la méthode *call()*
Dans ce cas, utilisable en script et déclaratif



Structure projet

```
(root)
+- src                                     # Classes Groovy classiques
|   +- org
|       +- foo
|           +- Bar.groovy                # Classe org.foo.Bar
+- vars
|       +- foo.groovy                    # Variable globale 'foo'
|       +- foo.txt                       # Aide pour la variable 'foo'
+- resources                             # Fichiers ressources
|       +- org
|           +- foo
|               +- bar.json              # Données pour org.foo.Bar
```



Exemple Code classique

Fichier *src/org/foo/Zot.groovy*

```
package org.foo
```

```
def checkoutFrom(repo) {  
    git url: "git@github.com:jenkinsci/${repo}"  
}  
return this
```

Utilisation dans une pipeline

```
def z = new org.foo.Zot()  
z.checkoutFrom('myRepo')
```



Variable globale

Fichier ***vars/log.groovy***. Le nom du fichier doit être en *camelCase*.

```
def info(message) {  
    echo "INFO: ${message}"  
}  
  
def warning(message) {  
    echo "WARNING: ${message}"  
}
```

Utilisation dans pipeline déclarative :

```
@Library('utils') _  
  
pipeline {  
    agent none  
    stage ('Example') {  
        steps {  
            script {  
                log.info 'Starting'  
                log.warning 'Nothing to do!'  
            } } } }  
}
```



Nouvelle step

Fichier *vars/sayHello.groovy*

```
def call(String name = 'human') {  
    // N'importe quelle steps peut être appelé dans ce bloc  
    // Scripted Pipeline  
    echo "Hello, ${name}."  
}
```

Utilisation

```
sayHello 'Joe'
```



Définition de librairies

Les librairies une fois développées peuvent être installées de différentes façons :

- **Global** Pipeline Libraries :
Manage Jenkins → Configure System → Global Pipeline Libraries
- **Folder** : Une librairie peut être définie au niveau d'un dossier
- Certains **plugins** ajoutent des façons de définir des librairies.
Ex : *github-branch-source*



Utilisation des librairies

Les librairies marquées « ***Load Implicitly*** » sont directement disponibles.

=> Les classes et les variables définies sont directement utilisables

Pour les autres, le Jenkinsfile doit explicitement les charger en utilisant l'annotation ***@Library***

Depuis la version 2.7, le plugin *Shared Groovy Libraries* permet de définir une ***step*** « ***library*** » qui charge dynamiquement la librairie.


- Avec cette méthode, les erreurs ne sont pas détectées à la compilation.



Option Load Implicitly

Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.



Library

Name

my-shared-library

?

Default version

master

?

Load implicitly

☐

?

Allow default version to be overridden

☒

?

Retrieval method

☒ Modern SCM

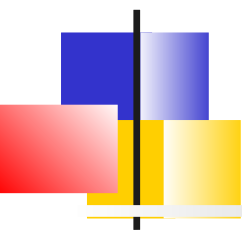
?



Exemples d'usage

-- Annotations

```
@Library('my-shared-library') _  
/* Avec une version, branch, tag, ou autre */  
@Library('my-shared-library@1.0') _  
/* Plusieurs librairies */  
@Library(['my-shared-library', 'otherlib@abc1234'])  
/* Typiquement devant la classe importée */  
@Library('somelib')  
import com.mycorp.pipeline.somelib.UsefulClass  
  
-- Plugin  
library('my-shared-  
    library').com.mycorp.pipeline.Utls.someStaticMethod()
```

Récupération de la librairie

La meilleure façon de référencer la librairie est d'utiliser un plugin de SCM supportant l'API **Modern SCM** (Supporté par Git, SVN)

Cela se fait :

- via la page d'administration pour les librairies globales
- Via les options de *@Library*
- Ou dynamiquement :

```
library identifier: 'custom-lib@master', retriever:  
modernSCM( [$class: 'GitSCMSource', remote:  
'git@git.mycorp.com:my-jenkins-utils.git',  
credentialsId: 'my-private-key'] )
```



Librairies de tiers

Il est possible également de charger une librairie à partir du dépôt Maven Central en utilisant l'annotation **@Grab**

```
@Grab('org.apache.commons:commons-math3:3.4.1')
import org.apache.commons.math3.primes.Primes
void parallelize(int count) {
    if (!Primes.isPrime(count)) {
        error "${count} was not prime"
    }
    // ...
}
```