# JENKINS: Structure Groovy

**Table of Contents**


- JENKINS: Structure Groovy
- Noms des paquets
- Importations
  - Importations par défaut
  - Importation simple
  - Importation d'étoile normale
  - Importation statique
  - Aliasing d'importation statique
  - Importation d'étoile statique
  - Alias d'importation
- Scripts versus classes
  - Méthode principale de l'application vs script
  - Classe de script
  - Les méthodes
- Variables


Ce chapitre couvre la structure de programme du langage de programmation Groovy.

# Noms des paquets

Les noms de paquetages jouent exactement le même rôle qu'en Java. Ils permettent de séparer la base de code sans aucun conflit. Les classes Groovy doivent spécifier leur package avant la définition de classe, sinon le package par défaut est utilisé.

Définir un paquet est très similaire à Java:

```
// defining a package named com.yoursite
package com.yoursite
```

Pour faire référence à une classe Foo dans le package com.yoursite.com , il faut utiliser le nom complet qualifié com.yoursite.com.Foo , sinon on peut utiliser une instruction d import comme indiqué ci-dessous.

# Importations

Pour faire référence à une classe, il faut utiliser une référence qualifiée à son package. Groovy suit la

notion de Java autorisant l'instruction d'import à résoudre les références de classe.

Par exemple, Groovy fournit plusieurs classes de générateur, telles que MarkupBuilder . MarkupBuilder est dans le paquet groovy.xml donc pour utiliser cette classe, on doit spécifier l'import comme suit:

```
// importing the class MarkupBuilder
import groovy.xml.MarkupBuilder
// using the imported class to create an object
def xml = new MarkupBuilder() assert xml != null
```

# Importations par défaut

Les importations par défaut sont les importations fournies par défaut par la langue Groovy. Par exemple, regardez le code suivant:

```
new Date()
```

Le même code en Java nécessite une instruction d'importation dans la classe Date comme suit: import java.util.Date. Groovy par défaut importe ces classes pour vous.

Les importations ci-dessous sont ajoutées par groovy :

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

Ceci est fait parce que les classes de ces paquets sont les plus utilisées. En important ces codes passe-partout, le code est réduit.

# Importation simple

Une importation simple est une instruction d'importation dans laquelle on définit entièrement le nom de la classe avec le package. Par exemple, l'instruction d'importation import groovy.xml.MarkupBuilder dans le code ci-dessous est une importation simple qui fait directement référence à une classe dans un package.

```
// importing the class MarkupBuilder
import groovy.xml.MarkupBuilder
// using the imported class to create an object
def xml = new MarkupBuilder() assert xml != null
```

# Importation d'étoile normale

Groovy, comme Java, offre un moyen spécial d'importer toutes les classes d'un package en utilisant * , ce que l'on appelle l'importation en étoile.

Par exemple **MarkupBuilder** est une classe qui se trouve dans le paquet **groovy.xml** , à côté d'une autre classe appelée **StreamingMarkupBuilder**, pour importer les deux classes, on peut faire:

```
import groovy.xml.MarkupBuilder
import groovy.xml.StreamingMarkupBuilder
def markupBuilder = new MarkupBuilder()
assert markupBuilder != null
assert new StreamingMarkupBuilder() != null
```

C'est un code parfaitement valide. Mais avec une importation *, on peut obtenir le même effet avec une seule ligne. L'étoile importe toutes les classes sous package groovy.xml :

```
import groovy.xml.*
def markupBuilder = new MarkupBuilder()
assert markupBuilder != null
assert new StreamingMarkupBuilder() != null
```

Un problème avec * imports est qu'ils peuvent encombrer votre espace de noms local. Mais avec les types de repliement fournis par Groovy, cela peut être résolu facilement.

# Importation statique

La fonction d'importation statique de Groovy permet de référencer les classes importées comme s'il s'agissait de méthodes statiques dans une classe privée:

```
import static Boolean.FALSE
assert !FALSE
//use directly, without Boolean prefix!
```

Cette fonctionnalité est similaire à la capacité d'importation statique de Java, mais est plus dynamique que Java en ce sens qu'elle vous permet de définir des méthodes portant le même nom qu'une méthode importée, à condition que vous disposiez de types différents:

```
import static
java.lang.String.format  (1)
class SomeClass { String format(Integer i)
  {  (2) i.toString() }
  static void main(String[] args) {
    assert format('String') == 'String'  (3)
    assert new SomeClass().format(Integer.valueOf(1)) == '1'
    }
  }
```

1 importation statique de la méthode 2 déclaration de méthode portant le même nom que la méthode importée statiquement ci-dessus, mais avec un type de paramètre différent 3 erreur de compilation en java, mais code valide pour groovy

Si on utilise les mêmes types, la classe importée est prioritaire.

# Aliasing d'importation statique

Les importations statiques avec le mot-clé as fournissent une solution élégante aux problèmes d’espace de noms. Supposons que vous souhaitiez obtenir une instance de Calendar aide de sa méthode getInstance() . C'est une méthode statique, nous pouvons donc utiliser une importation statique. Mais au lieu d'appeler chaque fois getInstance() , ce qui peut être trompeur lorsqu'il est séparé de son nom de classe, nous pouvons l'importer avec un alias pour améliorer la lisibilité du code:

```
import static
Calendar.getInstance as now
assert now().class == Calendar.getInstance().class
```

Maintenant, c'est propre!

# Importation d'étoile statique

Une importation d'étoile statique est très similaire à l'importation d'étoile normale. Toutes les méthodes statiques de la classe donnée seront importées.

Par exemple, supposons qu'on doive calculer des sinus et des cosinus pour une application. La classe java.lang.Math a des méthodes statiques nommées sin et cos qui répondent à ces besoins. Avec l'aide d'une importation d'étoile statique, on peut faire:

```
import static
java.lang.Math.*
assert sin(0) == 0.0
assert cos(0) == 1.0
```

On a pu accéder directement aux méthodes sin et cos , sans les préfixe **Math.**.

# Alias d'importation

Avec le type aliasing, on peut faire référence à un nom de classe complet en utilisant un nom de notre choix. Cela peut être fait avec le mot-clé as, comme auparavant.

Par exemple, on peut importer **java.sql.Date** en tant que **SQLDate** et l'utiliser dans le même fichier que **java.util.Date** sans avoir à utiliser le nom complet de l'une ou l'autre classe:

```
import java.util.Date
import java.sql.Date as SQLDate
Date utilDate = new Date(1000L)
SQLDate sqlDate = new SQLDate(1000L)
assert utilDate instanceof java.util.Date
assert sqlDate instanceof java.sql.Date
```

# Scripts versus classes

## Méthode principale de l'application vs script

Groovy prend en charge les scripts et les classes. Prenez le code suivant par exemple:

```
class Main {  (1)
   static void main(String... args) {  (2)
     println 'Groovy world!'  (3)
     }
   }
```

1 définir une classe Main , le nom est arbitraire 2 la méthode public static void main(String[]) définit la méthode principale de la classe 3 le corps principal de la méthode

Il s'agit d'un code typique de Java, dans lequel le code doit être intégré à une classe pour être exécutable. Groovy facilite les choses, le code suivant est équivalent:

```
println 'Groovy world!'
```

Un script peut être considéré comme une classe sans avoir à le déclarer, avec quelques différences.

## Classe de script

Un script est toujours compilé dans une classe. Le compilateur Groovy compilera la classe, le corps du script étant copié dans une méthode **run** . L'exemple précédent est donc compilé en java comme s'il était le suivant:

```
import org.codehaus.groovy.runtime.InvokerHelper
class Main extends Script {  (1)
   def run() {  (2)
     println 'Groovy world!'  (3)
     }
   static void main(String[] args) {  (4)
     InvokerHelper.runScript(Main, args)  (5)
     }
   }
```

1 La classe Main étend la classe groovy.lang.Script 2 groovy.lang.Script nécessite une méthode d’ run renvoyant une valeur 3 le corps du script va dans la méthode run 4 la méthode main est générée automatiquement 5 et délègue l'exécution du script sur la méthode d' run

Si le script est dans un fichier, le nom de base du fichier est utilisé pour déterminer le nom de la classe de script générée. Dans cet exemple, si le nom du fichier est Main.groovy , la classe de script sera Main .

## Les méthodes

Il est possible de définir des méthodes dans un script, comme illustré ici:

```
int fib(int n) { n < 2 ? 1 : fib(n-1) + fib(n-2) }
assert fib(10)==89
```

On peut également mélanger des méthodes et du code. La classe de script générée transportera toutes les méthodes dans la classe de script et assemblera tous les corps de script dans la méthode **run** :

```
println 'Hello'  (1)
int power(int n) { 2**n }  (2)
println "2^6==${power(6)}"  (3)
```

1. le script commence
2. une méthode est définie dans le corps du script
3. et le script continue

Ce code est converti en interne en:

```
import org.codehaus.groovy.runtime.InvokerHelper
class Main extends Script { int power(int n) { 2** n}  (1)
  def run() {
    println 'Hello'  (2)
    println "2^6==${power(6)}"  (3)
    }
  static void main(String[] args) {
    InvokerHelper.runScript(Main, args)
    }
}
```

1. la méthode **power** est copiée telle quelle dans la classe de script générée
2. la première instruction est copiée dans la méthode **run**
3. la deuxième instruction est copiée dans la méthode **run**

Même si Groovy crée une classe à partir de votre script, celle-ci est totalement transparente pour l'utilisateur. En particulier, les scripts sont compilés en bytecode et les numéros de ligne sont préservés. Cela implique que si une exception est levée dans un script, la trace de pile affichera les numéros de ligne correspondant au script d'origine,

pas le code généré que nous avons montré.

# Variables

Les variables d'un script ne nécessitent pas de définition de type. Cela signifie que ce script:

```
int x = 1 int y = 2 assert x+y == 3
```

se comportera comme:

```
x = 1 y = 2 assert x+y == 3
```

Cependant, il existe une différence sémantique entre les deux:

- si la variable est déclarée comme dans le premier exemple, il s'agit d'une variable locale . Il sera déclaré dans la méthode d' run que le compilateur générera et ne sera pas visible en dehors du corps principal du script. En particulier, une telle variable ne sera pas visible dans les autres méthodes du script
- si la variable est non déclarée, elle entre dans la liaison de script . La liaison est visible depuis les méthodes et est particulièrement importante si vous utilisez un script pour interagir avec une application et devez partager des données entre le script et l'application.
    - Si on veut qu'une variable devienne un champ de la classe sans entrer dans la Binding, on peut utiliser l' annotation @Field .