JENKINS: Syntaxe Groovy

Table of Contents

La grammaire du langage de programmation Groovy dérive de la grammaire Java, mais l'améliore avec des constructions spécifiques et permet certaines simplifications.

Commentaires

Commentaire sur une seule ligne

Les commentaires sur une seule ligne commencent par *et peuvent être trouvés à n'importe quelle position de la ligne. Les caractères suivant* jusqu'à la fin de la ligne sont considérés comme faisant partie du commentaire.

```
// a standalone single line comment
println "hello" // a comment till the end of the line
```

Commentaire multiligne

Un commentaire multiligne commence par /* et peut être trouvé à n'importe quel emplacement de la ligne. Les caractères suivants /* seront considérés comme faisant partie du commentaire, y compris les caractères de nouvelle ligne, jusqu'au premier */ fermant le commentaire. Les commentaires multilignes peuvent donc être placés à la fin d'une déclaration, voire à l'intérieur d'une déclaration.

```
 /* a standalone multiline comment
 spanning two lines */
 println "hello" /* a multiline comment
                   starting at the end of a statement */
 println 1 /* one */ + 2 /* two */
```

GroovyDoc commentaire

Comme pour les commentaires multilignes, les commentaires GroovyDoc sont multilignes, mais commencent par /** et se terminent par */ . Les lignes suivant la première ligne de commentaire GroovyDoc peuvent éventuellement commencer par une étoile * . Ces commentaires sont associés à:

```
définitions de type (classes, interfaces, enums, annotations),

définitions de champs et propriétés

définitions de méthodes
```

Bien que le compilateur ne se plaint pas du fait que les commentaires GroovyDoc ne sont pas associés aux éléments de langage ci-dessus, on doit ajouter ces commentaires avant le commentaire.

```
/**
```

```
* A Class description
*/
class Person {
   /** the name of the person */
   String name

    /**
    * Creates a greeting method for a certain person.
    *
    * @param otherPerson the person to greet
    * @return a greeting message
    */
    String greet(String otherPerson) {
      "Hello ${otherPerson}"
    }
}
```

GroovyDoc suit les mêmes conventions que le JavaDoc de Java. Vous pourrez donc utiliser les mêmes balises qu'avec JavaDoc.

Ligne Shebang

Outre le commentaire sur une ligne, il existe un commentaire spécial, souvent appelé ligne shebang comprise par les systèmes UNIX, qui permet d'exécuter les scripts directement à partir de la ligne de commande, à condition que vous ayez installé la distribution Groovy et que la commande groovy soit disponible à la page. PATH .

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

```
Le caractère # doit être le premier caractère du fichier. Toute indentation
produirait une erreur de compilation.
```

Mots-clés

La liste suivante représente tous les mots-clés de la langue Groovy:

Tableau 1. Mots-clés

as assert break case catch class const continue def default do else enum extends false finally for goto if implements import in instanceof interface new null package return super switch this throw throws trait true try while

Identifiants

Identifiants normaux

Les identifiants commencent par une lettre, un dollar ou un trait de soulignement. Ils ne peuvent pas commencer avec un numéro.

Une lettre peut être dans les plages suivantes:

```
'a' à 'z' (lettre ascii minuscule)
'A' à 'Z' (lettre majuscule asci)
'\ u00C0' à '\ u00D6'
'\ u00D8' à '\ u00F6'
'\ u00F8' à '\ u00FF'
'\ u0100' à '\ uFFFE'
```

Les caractères suivants peuvent alors contenir des lettres et des chiffres.

Voici quelques exemples d'identificateurs valides (ici, noms de variables):

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

Mais les suivants sont des identifiants invalides:

```
def 3tier
def a+b
def a#b
```

Tous les mots-clés sont également des identifiants valides à la suite d'un point:

```
foo.as
foo.assert
foo.break
foo.case
foo.catch
```

Identifiants cités

Les identificateurs cités apparaissent après le point d'une expression en pointillé. Par exemple, le name de l'expression person.name peut être cité avec person.“name” person.'name' person.“name” ou person.'name' . Ceci est particulièrement intéressant lorsque certains identifiants contiennent des caractères non autorisés interdits par la spécification du langage Java, mais autorisés par Groovy lorsqu'ils sont cités. Par exemple, des caractères tels qu'un tiret, un espace, un point d'exclamation,

etc.

```
def map = [:]

map."an identifier with a space and double quotes" = "ALLOWED"
map.'with-dash-signs-and-single-quotes' = "ALLOWED"

assert map."an identifier with a space and double quotes" == "ALLOWED"
assert map.'with-dash-signs-and-single-quotes' == "ALLOWED"
```

Comme nous le verrons dans la section suivante sur les chaînes , Groovy fournit différents littéraux de chaîne. Tous les types de chaînes sont autorisés après le point:

```
map.'single quote'
map."double quote"
map.'''triple single quote'''
map."""triple double quote"""
map./slashy string/
map.$/dollar slashy string/$
```

Il y a une différence entre les chaînes de caractères simples et les GStrings (chaînes interpolées) de Groovy, car dans ce dernier cas, les valeurs interpolées sont insérées dans la chaîne finale pour évaluer l'identificateur complet:

```
def firstname = "Homer"
map."Simpson-${firstname}" = "Homer Simpson"

assert map.'Simpson-Homer' == "Homer Simpson"
```

Chaînes

Les littéraux de texte sont représentés sous la forme de chaînes de caractères appelées chaînes. Groovy vous permet d’instancier des objets java.lang.String , ainsi que des chaînes GStrings (groovy.lang.GString), également appelées chaînes interpolées dans d’autres langages de programmation.

Chaîne entre guillemets

Les chaînes simples entre guillemets sont une série de caractères entourés de guillemets simples:

```
'a single quoted string'
```

Les chaînes à guillemets simples sont java.lang.String et ne prennent pas en charge l'interpolation.

Concaténation de chaînes

Toutes les chaînes Groovy peuvent être concaténées avec l'opérateur + :

```
assert 'ab' == 'a' + 'b'
```

Triple chaîne entre guillemets

Les chaînes triples simples sont une série de caractères entourés de triplets de guillemets simples:

```
'''a triple single quoted string'''
```

Les chaînes entre guillemets simples sont java.lang.String et ne supportent pas l'interpolation.

Les chaînes triples simples citées sont multilignes. Vous pouvez étendre le contenu de la chaîne au-delà des limites de la ligne sans avoir à diviser la chaîne en plusieurs parties, sans caractères de contaténation ou d'échappement de nouvelle ligne:

```
def aMultilineString = '''line one
line two
line three'''
```

Si le code est indenté, par exemple dans le corps de la méthode d'une classe, la chaîne contiendra le blanc de l'indentation. Le kit de développement Groovy contient des méthodes permettant de supprimer l'indentation avec la méthode **String#stripIndent()** et avec la méthode **String#stripMargin()** qui utilise un caractère délimiteur pour identifier le texte à supprimer au début d'une chaîne.

Lors de la création d'une chaîne comme suit:

```
def startingAndEndingWithANewline = '''
line one
line two
line three
'''
```

On remarquera que la chaîne résultante contient un caractère de nouvelle ligne en tant que premier caractère. Il est possible de supprimer ce caractère en échappant la nouvelle ligne avec une barre oblique inverse:

```
def strippedFirstNewline = '''\
line one
line two
line three
'''

assert !strippedFirstNewline.startsWith('\n')
```

Échapper des caractères spéciaux

On peut échapper des guillemets simples avec le caractère barre oblique inverse pour éviter de mettre fin au littéral de chaîne:

```
'an escaped single quote: \' needs a backslash'
```

Et on peut échapper un caractère d'échappement lui-même avec une double barre oblique inversée:

```
'an escaped escape character: \\ needs a double backslash'
```

Certains caractères spéciaux utilisent également la barre oblique inversée comme caractère d'échappement: Séquence d'échappement Personnage

- **\t** : tabulation
- **\b** : retour arrière
- **\n** : nouvelle ligne
- **\r** : retour chariot
- **\F** : formulaire d'alimentation
- **** : barre oblique inverse
- **\'** : guillemet simple (pour les chaînes guillemets simples et triples guillemets simples)
- **\“** : double guillemet (pour les guillemets doubles et triples doubles)

Séquence d'échappement Unicode

Pour les caractères qui ne sont pas présents sur votre clavier, on peut utiliser des séquences d'échappement Unicode: une barre oblique inverse, suivie de “u”, puis de 4 chiffres hexadécimaux.

Par exemple, le symbole de l’euro peut être représenté avec:

```
'The Euro currency symbol: \u20AC'
```

Chaîne entre guillemets

Les chaînes entre guillemets sont une série de caractères entourés de guillemets:

```
"a double quoted string"
```

Les chaînes double quote sont plain java.lang.String s'il n'y a pas d'expression interpolée, mais sont des instances groovy.lang.GString si une interpolation est présente.

Pour échapper une double quote, vous pouvez utiliser le caractère barre oblique inverse: “Une double citation: \” ”.

Interpolation de chaîne

Toute expression Groovy peut être interpolée dans tous les littéraux de chaîne, à l'exception des chaînes simples et triples simples. L'interpolation consiste à remplacer un espace réservé dans la chaîne par sa valeur lors de l'évaluation de la chaîne. Les expressions d'espace réservé sont entourées de ${} ou préfixées par $ pour les expressions en pointillé. La valeur de l'expression dans l'espace réservé est évaluée sous forme de chaîne lorsque GString est transmis à une méthode prenant un argument String en appelant toString() pour cette expression.

Ici, nous avons une chaîne avec un espace réservé référençant une variable locale:

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

Mais toute expression Groovy est valide, comme on peut le voir dans cet exemple avec une expression arithmétique:

```
def sum = "The sum of 2 and 3 equals ${2 + 3}"
assert sum.toString() == 'The sum of 2 and 3 equals 5'
```

Non seulement les expressions sont autorisées entre les caractères réservés ${} , mais il en va de même pour les instructions. Cependant, la valeur d'une instruction est simplement null . Ainsi, si plusieurs instructions sont insérées dans cet espace réservé, la dernière doit en quelque sorte renvoyer une valeur significative à insérer. Par exemple, “La somme de 1 et 2 est égale à $ {def a = 1; def b = 2; a + b}” est prise en charge et fonctionne comme prévu, mais une bonne pratique consiste généralement à s'en tenir à des expressions simples à l'intérieur de caractères génériques GString. .

En plus des espaces réservés ${} , nous pouvons également utiliser un seul signe $ préfixant une expression en pointillé:

```
def person = [name: 'Guillaume', age: 36]
assert "$person.name is $person.age years old" == 'Guillaume is 36 years
old'
```

Toutefois, seules les expressions en pointillés de la forme ab , abc , etc. sont valides, mais les expressions contenant des parenthèses, telles que des appels de méthode, des accolades pour les fermetures ou des opérateurs arithmétiques, seraient invalides. Étant donné la définition de variable suivante d'un nombre:

```
def number = 3.14
```

L'instruction suivante **groovy.lang.MissingPropertyException** une groovy.lang.MissingPropertyException car Groovy pense qu'on essaye d'accéder à la propriété toString de ce nombre, qui n'existe pas:

```
shouldFail(MissingPropertyException) { println "$number.toString()" }
```

On peut penser que “$number.toString()” est interprété par l'analyseur comme “${number.toString}()” .

Pour échapper les espaces réservés $ ou ${} dans un GString afin qu'ils apparaissent tels quels sans

interpolation, il suffit d'utiliser un caractère \ barre oblique inverse pour échapper au signe dollar:

```
assert '${name}' == "\${name}"
```

Cas particulier des expressions de fermeture interpolées

Jusqu'à présent, on a vu que nous pourrions interpoler des expressions arbitraires à l'intérieur de l'espace réservé ${} , mais il existe un cas spécial et une notation pour les expressions de fermeture. Lorsque le paramètre fictif contient une flèche, ${→} , l'expression est en fait une expression de fermeture. On peut la considérer comme une fermeture précédée d'un dollar:

```
def sParameterLessClosure = "1 + 2 == ${-> 3}"  (1)
assert sParameterLessClosure == '1 + 2 == 3'
def sOneParamClosure = "1 + 2 == ${ w -> w << 3}"  (2)
assert sOneParamClosure == '1 + 2 == 3'
```

1. La fermeture est une fermeture sans paramètre qui ne prend pas d'arguments.
2. Ici, la fermeture prend un seul argument java.io.StringWriter , auquel vous pouvez ajouter du contenu avec l'opérateur « leftShift. Dans les deux cas, les deux espaces réservés sont des fermetures intégrées.

En apparence, cela ressemble à une manière plus verbeuse de définir des expressions à interpoler, mais les fermetures ont un avantage intéressant par rapport à de simples expressions: l'évaluation paresseuse.

Considérons l'exemple suivant:

```
def number = 1  (1)
def eagerGString = "value == ${number}"
def lazyGString = "value == ${ -> number }"
assert eagerGString == "value == 1"  (2)
assert lazyGString == "value == 1"  (3)
number = 2  (4)
assert eagerGString == "value == 1"  (5)
assert lazyGString == "value == 2"  (6)
```

1. On définit une variable number contenant 1 que nous interpolons ensuite dans deux GStrings, en tant qu'expression dans **eagerGString** et en tant que fermeture dans **lazyGString**.
2. On s'attend donc à ce que la chaîne résultante contienne la même valeur de chaîne que 1 pour **eagerGString**.
3. De même pour **lazyGString**
4. Ensuite, on change la valeur de la variable en un nouveau nombre
5. Avec une expression interpolée simple, la valeur était liée au moment de la création de **GString**.
6. Mais avec une expression de fermeture, la fermeture est appelée à chaque contrainte de **GString** dans String, ce qui donne une chaîne mise à jour contenant la nouvelle valeur numérique.

Une expression de fermeture incorporée prenant plus d'un paramètre générera une exception lors de l'exécution. Seules les fermetures avec zéro ou un paramètre sont autorisées.

Interopérabilité avec Java

Lorsqu'une méthode (implémentée en Java ou Groovy) attend un java.lang.String , mais que nous passons une instance groovy.lang.GString , la toString() de GString est appelée automatiquement et de manière transparente.

```
String takeString(String message) {  (4)
assert message instanceof String  (5)
return message } def message = "The message is ${'hello'}"  (1)
assert message instanceof GString  (2)
def result = takeString(message)  (3)
assert result instanceof String
assert result == 'The message is hello'
```

1 Nous créons une variable GString 2 Nous vérifions qu'il s'agit d'une instance de GString 3 Nous passons ensuite GString à une méthode utilisant une chaîne comme paramètre. 4 La signature de la méthode takeString() indique explicitement que son seul paramètre est une chaîne. 5 Nous vérifions également que le paramètre est bien une chaîne et non un GString.

GString et String hashCodes

Bien que les chaînes interpolées puissent être utilisées à la place des chaînes Java standard, elles diffèrent d'une chaîne à une autre: leurs hashCodes sont différents. Les chaînes Java simples sont immuables, alors que la représentation String résultante d'un GString peut varier en fonction de ses valeurs interpolées. Même pour la même chaîne résultante, GStrings et Strings n'ont pas le même hashCode.

```
assert "one: ${1}".hashCode() != "one: 1".hashCode()
```

GString et Strings ayant différentes valeurs hashCode, l'utilisation de GString en tant que clés de carte doit être évitée, en particulier si nous essayons de récupérer une valeur associée avec une chaîne au lieu d'un GString.

```
def key = "a"
def m = ["${key}": "letter ${key}"]  (1)
assert m["a"] == null  (2)
```

1 La carte est créée avec une paire initiale dont la clé est un GString 2 Lorsque nous essayons d'extraire la valeur avec une clé String, nous ne la trouverons pas, car Strings et GString ont des valeurs hashCode différentes.

Chaîne entre triple guillemets

Les chaînes entre triples doubles quotes se comportent comme des chaînes doubles, avec en plus le fait qu'elles sont multilignes, comme les chaînes triples simples.

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

    You're the winner of the lottery!

    Yours sincerly,

    Dave
"""

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

Ni les guillemets doubles ni les guillemets simples ne doivent être échappés dans des chaînes triples doubles.

Chaîne Slashy

Au-delà des chaînes citées habituelles, Groovy propose des chaînes slashy, qui utilisent / comme délimiteurs. Les chaînes slashy sont particulièrement utiles pour définir des expressions régulières et des modèles, car il n'est pas nécessaire d'échapper aux barres obliques inverses.

Exemple de chaîne slashy:

```
def fooPattern = /.*foo.*/
assert fooPattern == '.*foo.*'
```

Seules les barres obliques doivent être échappées avec une barre oblique inverse:

```
def escapeSlash = /The character \/ is a forward slash/
assert escapeSlash == 'The character / is a forward slash'
```

Les chaînes slashy sont multilignes:

```
def multilineSlashy = /one
two
three/
assert multilineSlashy.contains('\n')
```

Les chaînes slashy peuvent également être interpolées (par exemple, un GString):

```
def color = 'blue'
```

```
def interpolatedSlashy = /a ${color} car/
assert interpolatedSlashy == 'a blue car'
```

Il y a quelques pièges à connaître.

Une chaîne slashy vide ne peut pas être représentée par une double barre oblique, car l'analyseur Groovy le comprend comme un commentaire de ligne. C'est pourquoi l'assertion suivante ne compilerait pas car elle ressemblerait à une déclaration non terminée:

```
 assert '' == //
```

Comme les chaînes slashy ont été principalement conçues pour faciliter l'expression rationnelle, quelques erreurs dans GStrings telles que $() fonctionneront avec des chaînes slashy.

Chaîne slashy dollar

Les chaînes dollar slashy sont des chaînes de caractères multilignes délimitées par une ouverture $/ et une clôture /$. Le caractère qui s'échappe est le signe dollar et peut échapper à un autre dollar ou à une barre oblique. Toutefois, il n'est pas nécessaire d'échapper les barres obliques dollar et forward, sauf si vous souhaitez échapper le dollar d'une sous-séquence de chaîne commençant par une séquence fictive GString, ou si vous devez échapper à une séquence commençant par un délimiteur de chaîne slashy dollar de clôture .

Voici un exemple:

```
def name = "Guillaume"
def date = "April, 1st"

def dollarSlashy = $/
    Hello $name,
    today we're ${date}.

    $ dollar sign
    $$ escaped dollar sign
    \ backslash
    / forward slash
    $/ escaped forward slash
    $$$/ escaped opening dollar slashy
    $/$$ escaped closing dollar slashy
/$

assert [
    'Guillaume',
    'April, 1st',
    '$ dollar sign',
    '$ escaped dollar sign',
    '\\ backslash',
    '/ forward slash',
    '/ escaped forward slash',
```

```
    '$/ escaped opening dollar slashy',
    '/$ escaped closing dollar slashy'
].every { dollarSlashy.contains(it) }
```

Tableau récapitulatif des chaînes

String name	String syntax	Interpolated	Multiline	Escape character
Single quoted	'…'			\
Triple single quoted	'''…'''		x	\
Double quoted	"…"	x		\
Triple double quoted	"""…"""	x	x	\
Slashy	/…/	x	x	\
Dollar slashy	$/…/$	x	x	$

Characters

Contrairement à Java, Groovy n'a pas de littéral de caractère explicite. Cependant, vous pouvez être explicite sur la transformation d'une chaîne Groovy en caractère réel, de trois manières différentes:

```
char c1 = 'A'  (1)
assert c1 instanceof Character
def c2 = 'B' as char  (2)
assert c2 instanceof Character
def c3 = (char)'C'  (3)
assert c3 instanceof Character
```

1. en étant explicite lors de la déclaration d'une variable contenant le caractère en spécifiant le type **char**
2. en utilisant la contrainte de type avec l'opérateur **as**
3. en utilisant une opération **cast to char**

La première option 1 est intéressante lorsque le caractère est contenu dans une variable, tandis que les deux autres (2 et 3) sont plus intéressantes lorsqu'une valeur de caractère doit être passée en tant qu'argument d'un appel de méthode.

Nombres

Groovy prend en charge différents types de littéraux intégraux et décimaux, pris en charge par les types Number habituels de Java.

Nombres entiers

Les types entiers sont les mêmes qu'en Java:

- byte
- char
- short
- int
- long
- java.lang.BigInteger

On peut créer des nombres entiers de ces types avec les déclarations suivantes:

```
// primitive types
byte b = 1
char c = 2
short s = 3
int i = 4
long l = 5
// infinite precision
BigInteger bi = 6
```

Lorsque'on utilise la saisie facultative à l'aide du mot clé def , le type du nombre entier varie: il s'adapte à la capacité du type pouvant contenir ce numéro.

Pour les nombres positifs:

```
def a = 1
assert a instanceof Integer
// Integer.MAX_VALUE
def b = 2147483647
assert b instanceof Integer
// Integer.MAX_VALUE + 1
def c = 2147483648
assert c instanceof Long
// Long.MAX_VALUE
def d = 9223372036854775807
assert d instanceof Long
// Long.MAX_VALUE + 1
def e = 9223372036854775808
assert e instanceof BigInteger
```

Ainsi que pour les nombres négatifs:

```
def na = -1
assert na instanceof Integer
// Integer.MIN_VALUE
def nb = -2147483648
assert nb instanceof Integer
// Integer.MIN_VALUE - 1
```

```
def nc = -2147483649
assert nc instanceof Long
// Long.MIN_VALUE
def nd = -9223372036854775808
assert nd instanceof Long
// Long.MIN_VALUE - 1
def ne = -9223372036854775809
assert ne instanceof BigInteger
```

Représentations alternatives non base 10

Les nombres peuvent également être représentés sous forme de bases binaires, octales, hexadécimales et décimales.

Nombres binaires

Les nombres binaires commencent par un préfixe 0b :

```
int xInt = 0b10101111
assert xInt == 175
short xShort = 0b11001001
assert xShort == 201 as short
byte xByte = 0b11
assert xByte == 3 as byte
long xLong = 0b101101101101
assert xLong == 2925l
BigInteger xBigInteger = 0b111100100001
assert xBigInteger == 3873g
int xNegativeInt = -0b10101111
assert xNegativeInt == -175
```

Nombres octal

Les nombres octaux sont spécifiés dans le format typique de 0 suivi de chiffres octaux.

```
int xInt = 077
assert xInt == 63
short xShort = 011
assert xShort == 9 as short
byte xByte = 032
assert xByte == 26 as byte
long xLong = 0246
assert xLong == 166l
BigInteger xBigInteger = 01111
assert xBigInteger == 585g
int xNegativeInt = -077
```

```
assert xNegativeInt == -63
```

Nombres hexadécimaux

Les nombres hexadécimaux sont spécifiés au format typique de 0x suivi de chiffres hexadécimaux.

```
int xInt = 0x77
assert xInt == 119
short xShort = 0xaa
assert xShort == 170 as short
byte xByte = 0x3a
assert xByte == 58 as byte
long xLong = 0xffff
assert xLong == 65535l
BigInteger xBigInteger = 0xaaaa
assert xBigInteger == 43690g
Double xDouble = new Double('0x1.0p0')
assert xDouble == 1.0d
int xNegativeInt = -0x77
assert xNegativeInt == -119
```

Nombres décimaux

Les types littéraux décimaux sont les mêmes qu'en Java:

- float
- double
- java.lang.BigDecimal

On peut créer des nombres décimaux de ces types avec les déclarations suivantes:

```
// primitive types
float f = 1.234
double d = 2.345
// infinite precision
BigDecimal bd = 3.456
```

Les décimales peuvent utiliser des exposants, avec la lettre de l'exposant e ou E , suivie d'un signe facultatif et d'un nombre entier représentant l'exposant:

```
assert 1e3 == 1_000.0
assert 2E4 == 20_000.0
assert 3e+1 == 30.0
assert 4E-2 == 0.04
assert 5e-1 == 0.5
```

Pour le calcul exact des nombres décimaux, Groovy choisit java.lang.BigDecimal comme type de nombre décimal. De plus, float et double sont pris en charge, mais nécessitent une déclaration de

type explicite, une contrainte de type ou un suffixe. Même si BigDecimal est la valeur par défaut pour les nombres décimaux, ces littéraux sont acceptés dans les méthodes ou les fermetures prenant les types float ou double comme type de paramètre.

Les nombres décimaux ne peuvent pas être représentés à l'aide d'une représentation binaire, octale ou hexadécimale.

Souligner en littéraux

Lors de l'écriture de longs nombres littéraux, il est plus difficile de comprendre comment certains nombres sont regroupés, par exemple avec des groupes de milliers, de mots, etc. En vous permettant de placer un soulignement dans des littéraux numériques, cela signifie plus facile de repérer ces groupes:

```
long creditCardNumber = 1234_5678_9012_3456L
long socialSecurityNumbers = 999_99_9999L
double monetaryAmount = 12_345_132.12
long hexBytes = 0xFF_EC_DE_5E
long hexWords = 0xFFEC_DE5E
long maxLong = 0x7fff_ffff_ffff_ffffL
long alsoMaxLong = 9_223_372_036_854_775_807L
long bytes = 0b11010010_01101001_10010100_10010010
```

Suffixes de type nombre

On peut forcer un nombre (y compris binaire, octal et hexadécimal) à avoir un type spécifique en donnant un suffixe (voir tableau ci-dessous), en majuscule ou en minuscule.

Type	
BigInteger	G ou g
Long	L ou l
Entier	I ou i
BigDecimal	G ou g
Double	D ou d
Flotteur	F ou f

Exemples:

```
assert 42I == new Integer('42')
assert 42i == new Integer('42') // lowercase i more readable
assert 123L == new Long("123") // uppercase L more readable
assert 2147483648 == new Long('2147483648') // Long type used, value too
large for an Integer
assert 456G == new BigInteger('456')
assert 456g == new BigInteger('456')
assert 123.45 == new BigDecimal('123.45') // default BigDecimal type used
assert 1.200065D == new Double('1.200065')
```

```
assert 1.234F == new Float('1.234')
assert 1.23E23D == new Double('1.23E23')
assert 0b1111L.class == Long // binary
assert 0xFFi.class == Integer // hexadecimal
assert 034G.class == BigInteger // octal
```

Opérations mathématiques

Bien que les opérateurs soient couverts plus tard, il est important de discuter du comportement des opérations mathématiques et des types qui en résultent.

Opérations binaires de division et de puissance mises à part (voir ci-dessous),

- les opérations binaires entre byte , char , short et int résultent en int
- les opérations binaires impliquant long avec byte , char , short et int résultat long
- les opérations binaires impliquant BigInteger et tout autre type intégral donnent BigInteger
- les opérations binaires impliquant BigDecimal avec byte , char , short , int et BigInteger donnent BigDecimal
- les opérations binaires entre float , double et BigDecimal résultat double
- opérations binaires entre deux résultats BigDecimal dans BigDecimal

Le tableau suivant résume ces règles:

	byte	**char**	**short**	**int**	**long**	**BigInteger**	**float**	**double**	**BigDecimal**
byte	int	int	int	int	long	BigInteger	double	double	BigDecimal
char		int	int	int	long	BigInteger	double	double	BigDecimal
short			int	int	long	BigInteger	double	double	BigDecimal
int				int	long	BigInteger	double	double	BigDecimal
long					long	BigInteger	double	double	BigDecimal
BigInteger						BigInteger	double	double	BigDecimal
float							double	double	double
double								double	double
BigDecimal									BigDecimal

Grâce à la surcharge d'opérateurs de Groovy, les opérateurs arithmétiques habituels fonctionnent également avec BigInteger et BigDecimal , contrairement à Java où vous devez utiliser des méthodes explicites pour exploiter ces nombres.

Le cas de l'opérateur de division

Les opérateurs de division / (et /= pour la division et l'affectation) produisent un résultat double si l'un des opérandes est un float ou un double , et un résultat BigDecimal sinon (lorsque les deux opérandes sont une combinaison quelconque d'un type entier short , char , byte , int , long , BigInteger ou BigDecimal).

BigDecimal division BigDecimal est effectuée avec la méthode divide() si la division est exacte (c'est-à-dire donnant un résultat pouvant être représenté dans les limites de la même précision et de la même échelle), ou en utilisant un MathContext avec une précision du maximum des deux opérandes.

précision plus une précision supplémentaire de 10, et une échelle du maximum de 10 et du maximum de l'échelle des opérandes.

Pour les divisions entières comme en Java, vous devez utiliser la méthode intdiv() , car Groovy ne fournit pas de symbole d'opérateur de division d'entiers dédié.

Le cas de l'opérateur électrique

Le fonctionnement en puissance est représenté par l'opérateur ** , avec deux paramètres: la base et l'exposant. Le résultat de l'opération d'alimentation dépend de ses opérandes et du résultat de l'opération (en particulier si le résultat peut être représenté sous forme de valeur intégrale).

Les règles suivantes sont utilisées par l'opération power de Groovy pour déterminer le type résultant:

- Si l'exposant est une valeur décimale
  - si le résultat peut être représenté sous la forme d'un Integer , alors retourne un Integer
  - sinon, si le résultat peut être représenté par Long , alors retourne Long
    - sinon retourne un Double
- Si l'exposant est une valeur intégrale
  - si l'exposant est strictement négatif, renvoyez un Integer , Long ou Double si la valeur du résultat correspond à ce type
    - si l'exposant est positif ou nul
      - si la base est un BigDecimal , alors retourne une valeur de résultat BigDecimal
      - si la base est un BigInteger , retourne une valeur de résultat BigInteger
      - si la base est un Integer , alors retourne un Integer si la valeur du résultat y correspond, sinon un BigInteger
      - si la base est un Long , alors retourne un Long si la valeur du résultat y correspond, sinon un BigInteger

Nous pouvons illustrer ces règles avec quelques exemples:

```
// base and exponent are ints and the result can be represented by an
Integer
assert    2    **   3    instanceof Integer    //  8
assert   10    **   9    instanceof Integer    //  1_000_000_000

// the base is a long, so fit the result in a Long
// (although it could have fit in an Integer)
assert    5L   **   2    instanceof Long       //  25

// the result can't be represented as an Integer or Long, so return a
BigInteger
assert  100    **  10    instanceof BigInteger //  10e20
assert 1234    ** 123    instanceof BigInteger //  17051580621272704287 5...

// the base is a BigDecimal and the exponent a negative int
// but the result can be represented as an Integer
assert    0.5  **  -2    instanceof Integer    //  4
```

```
// the base is an int, and the exponent a negative float
// but again, the result can be represented as an Integer
assert    1    **  -0.3f instanceof Integer    //  1

// the base is an int, and the exponent a negative int
// but the result will be calculated as a Double
// (both base and exponent are actually converted to doubles)
assert   10    **  -1    instanceof Double     //  0.1

// the base is a BigDecimal, and the exponent is an int, so return a
BigDecimal
assert    1.2  **  10    instanceof BigDecimal //  6.1917364224

// the base is a float or double, and the exponent is an int
// but the result can only be represented as a Double value
assert    3.4f **   5    instanceof Double     //  454.35430372146965
assert    5.6d **   2    instanceof Double     //  31.359999999999996

// the exponent is a decimal value
// and the result can only be represented as a Double value
assert    7.8  **   1.9  instanceof Double     //  49.542708423868476
assert    2    **   0.1f instanceof Double     //  1.0717734636432956
```

Booléens

Boolean est un type de données spécial utilisé pour représenter les valeurs de vérité: true et false . Utilisez ce type de données pour les indicateurs simples qui suivent les conditions vraies / fausses.

Les valeurs booléennes peuvent être stockées dans des variables, assignées dans des champs, comme tout autre type de données:

```
def myBooleanVariable = true
boolean untypedBooleanVar = false
booleanField = true
```

true et false sont les deux seules valeurs booléennes primitives. Mais des expressions booléennes plus complexes peuvent être représentées à l'aide d' opérateurs logiques .

De plus, Groovy a des règles spéciales (souvent appelées Groovy Truth) pour contraindre des objets non booléens à une valeur booléenne.

listes

Groovy utilise une liste de valeurs séparées par des virgules, entourées de crochets, pour désigner les listes. Les listes Groovy sont du JDK simple java.util.List , car Groovy ne définit pas ses propres classes de collection. L'implémentation concrète de la liste utilisée lors de la définition des littéraux de liste

est java.util.ArrayList par défaut, à moins que vous ne décidiez de spécifier le contraire, comme nous le verrons plus tard.

```
def numbers = [1, 2, 3]  (1)
assert numbers instanceof List  (2)
assert numbers.size() == 3  (3)
```

1. On définit une liste de nombres délimités par des virgules et entourés de crochets, et nous affectons cette liste à une variable
2. La liste est une instance de l'interface java.util.List de Java.
3. La taille de la liste peut être interrogée avec la méthode size() , et montre que notre liste contient 3 éléments

Dans l'exemple ci-dessus, nous avons utilisé une liste homogène, mais vous pouvez également créer des listes contenant des valeurs de types hétérogènes:

```
def heterogeneous = [1, "a", true]  (1)
```

1. Notre liste ici contient un nombre, une chaîne et une valeur booléenne

On a mentionné que, par défaut, les littéraux de liste sont en fait des instances de java.util.ArrayList , mais il est possible d'utiliser un type de support différent pour nos listes, grâce à l'utilisation de la contrainte de type avec l'opérateur as ou d'une déclaration de type explicite pour vos variables. :

```
def arrayList = [1, 2, 3]
assert arrayList instanceof java.util.ArrayList
def linkedList = [2, 3, 4] as LinkedList  (1)
assert linkedList instanceof java.util.LinkedList
LinkedList otherLinked = [3, 4, 5]  (2)
assert otherLinked instanceof java.util.LinkedList
```

1. On utilise la contrainte avec l'opérateur as pour demander explicitement une implémentation java.util.LinkedList
2. On peut dire que la variable contenant le littéral de liste est de type java.util.LinkedList

On peutaccéder aux éléments de la liste avec l'opérateur d'indice [] (pour la lecture et la définition de valeurs) avec des index positifs ou négatifs pour accéder aux éléments de la fin de la liste, ainsi qu'avec des plages, et utiliser l'opérateur « leftShift pour ajouter des éléments à une liste:

```
def letters = ['a', 'b', 'c', 'd']
assert letters[0] == 'a'  (1)
assert letters[1] == 'b'
assert letters[-1] == 'd'  (2)
assert letters[-2] == 'c' letters[2] = 'C'  (3)
assert letters[2] == 'C' letters << 'e'  (4)
assert letters[ 4] == 'e'
assert letters[-1] == 'e'
assert letters[1, 3] == ['b', 'd']  (5)
assert letters[2..4] == ['C', 'd', 'e']  (6)
```

1. Accéder au premier élément de la liste (comptage à base zéro)
2. Accéder au dernier élément de la liste avec un index négatif: -1 est le premier élément à partir

de la fin de la liste.
3. Utiliser une affectation pour définir une nouvelle valeur pour le troisième élément de la liste
4. Utiliser l'opérateur « leftShift pour ajouter un élément à la fin de la liste.
5. Accéder à deux éléments à la fois, en renvoyant une nouvelle liste contenant ces deux éléments
6. Utiliser une plage pour accéder à une plage de valeurs de la liste, allant du début à la fin d'un élément

Comme les listes peuvent être de nature hétérogène, les listes peuvent également contenir d'autres listes pour créer des listes multidimensionnelles:

```
def multi = [[0, 1], [2, 3]]  (1)
assert multi[1][0] == 2  (2)
```

1. Définir une liste de numéros
2. Accéder au deuxième élément de la liste la plus en haut et au premier élément de la liste interne

Tableaux

Groovy réutilise la notation de liste pour les tableaux, mais pour créer de tels tableaux littéraux, vous devez définir explicitement le type du tableau par la contrainte ou la déclaration du type.

```
String[] arrStr = ['Ananas', 'Banana', 'Kiwi']  (1)
assert arrStr instanceof String[]  (2)
assert !(arrStr instanceof List)
def numArr = [1, 2, 3] as int[]  (3)
assert numArr instanceof int[]  (4)
assert numArr.size() == 3
```

1. Définir un tableau de chaînes en utilisant une déclaration de type de variable explicite
2. Affirmer que nous avons créé un tableau de chaînes
3. Créer un tableau d'ints avec l'opérateur as
4. Affirmer que nous avons créé un tableau d'ints primitifs

On peut également créer des tableaux multidimensionnels:

```
def matrix3 = new Integer[3][3]  (1)
assert matrix3.size() == 3
Integer[][] matrix2  (2)
matrix2 = [[1, 2], [3, 4]]
assert matrix2 instanceof Integer[][]
```

1. On peut définir les limites d'un nouveau tableau
2. Ou déclarer un tableau sans spécifier ses limites

L'accès aux éléments d'un tableau suit la même notation que pour les listes:

```
String[] names = ['CÃ©dric', 'Guillaume', 'Jochen', 'Paul']
```

```
assert names[0] == 'CÃ©dric'  (1)
names[2] = 'Blackdrag'  (2)
assert names[2] == 'Blackdrag'
```

1. Récupérer le premier élément du tableau
2. Définit la valeur du troisième élément du tableau sur une nouvelle valeur

La notation d'initialiseur de tableau de Java n'est pas prise en charge par Groovy, car les accolades peuvent être mal interprétées avec la notation des fermetures de Groovy.

cartes

Parfois appelés dictionnaires ou tableaux associatifs dans d'autres langues, Groovy propose des cartes. Les cartes associent des clés à des valeurs, séparant les clés et les valeurs par des deux points, et chaque paire clé / valeur avec des virgules, ainsi que l'ensemble des clés et des valeurs entourées de crochets.

```
def colors = [red: '#FF0000', green: '#00FF00', blue: '#0000FF']  (1)
assert colors['red'] == '#FF0000'  (2)
assert colors.green == '#00FF00'  (3)
colors['pink'] = '#FF00FF'  (4)
colors.yellow = '#FFFF00'  (5)
assert colors.pink == '#FF00FF'
assert colors['yellow'] == '#FFFF00'
assert colors instanceof java.util.LinkedHashMap
```

1. On définit une mappe de noms de couleur de chaîne, associés à leurs couleurs html codées en hexadécimal
2. On utilise la notation en indice pour vérifier le contenu associé à la touche red
3. On peut également utiliser la notation de propriété pour affirmer la représentation hexadécimale de la couleur verte
4. De même, on peut utiliser la notation en indice pour ajouter une nouvelle paire clé / valeur
5. Ou la notation de propriété, pour ajouter la couleur yellow

Lors de l'utilisation de noms pour les clés, nous définissons en fait des clés de chaîne dans la carte.

Groovy crée des cartes qui sont en fait des instances de java.util.LinkedHashMap .

Si on essayed'accéder à une clé qui n'est pas présente dans la carte:

```
assert colors.unknown == null
```

On récupére un résultat null .

Dans les exemples ci-dessus, on a utilisé des clés de chaîne, mais on peut également utiliser des valeurs d'autres types en tant que clés:

```
def numbers = [1: 'one', 2: 'two']
assert numbers[1] == 'one'
```

Ici, on a utilisé des nombres en tant que clés, car les nombres peuvent être reconnus sans ambiguïté, donc Groovy ne créera pas de clé de chaîne comme dans nos exemples précédents. Mais on peut également passer une variable à la place de la clé, pour que la valeur de cette variable devienne la clé:

```
def key = 'name'
def person = [key: 'Guillaume']  (1)
assert !person.containsKey('name')  (2)
assert person.containsKey('key')  (3)
```

1. La key associée au nom 'Guillaume' sera en réalité la chaîne “key” , pas la valeur associée à la variable key
2. La carte ne contient pas la clé 'name'
3. Au lieu de cela, la carte contient une 'key' clé 'key'

On peut également passer des chaînes entre guillemets ainsi que des clés: [“name”: “Guillaume”]. Ceci est obligatoire si votre chaîne de clé n'est pas un identifiant valide, par exemple si vous souhaitez créer une clé de chaîne contenant un hachage, comme dans: [“nom-rue”: “Rue principale”].

Pour transmettre des valeurs de variable en tant que clés dans les définitions de carte, il faut entourer la variable ou l'expression de parenthèses:

```
person = [(key): 'Guillaume']  (1)
assert person.containsKey('name')  (2)
assert !person.containsKey('key')  (3)
```

1. Cette fois, on entoure la keyvariable de parenthèses pour indiquer à l’analyseur que l'on passe une variable plutôt que de définir une clé de chaîne.
2. La carte contient la nameclé
3. Mais la carte ne contient pas la keyclé comme avant