Les opérateurs

Un opérateur prend un ou plusieurs opérandes et produit une nouvelle valeur. Les opérateurs en Java sont très proches de ceux des langages C et C++ qui les ont inspirés.

L'opérateur d'affectation

L'affectation est réalisée grâce à l'opérateur **=**. Cet opérateur, copie la valeur du paramètre de droite (appelé *rvalue*) dans le paramètre de gauche (appelé *lvalue*). Java opère donc par copie. Cela signifie que si l'on change plus tard la valeur d'un des opérandes, la valeur de l'autre ne sera pas affectée.

```
int i = 1;
int j = i; // j reçoit la copie de la valeur de i


i = 10; // maintenant i vaut 10 mais j vaut toujours 1
```

Pour les variables de type objet, on appelle ces variables des **handlers** car la variable ne contient pas à proprement parler un objet mais la *référence d'un objet*. On peut dire aussi qu'elle pointe vers la zone mémoire de cet objet. Cela a plusieurs conséquences importantes.

```
Voiture v1 = new Voiture();
Voiture v2 = v1;
```

Dans, l'exemple ci-dessus, **v2** reçoit la copie de l'adresse de l'objet contenue dans **v1**. Donc ces deux variables référencent bien le même objet et nous pouvons le manipuler à travers l'une ou l'autre de ces variables. Si plus loin dans le programme, on écrit :

```
v1 = new Voiture();
```

v1 reçoit maintenant la référence d'un nouvel objet et les variables **v1** et **v2** référencent des instances différentes de **Voiture**. Si enfin, j'écris :

```
v2 = null;
```

Maintenant, la variable **v2** contient la valeur spéciale **null** qui indique qu'elle ne référence rien. Mais l'instance de *Voiture* que la variable **v2** référençait précédemment, n'a pas disparue pour autant. Elle existe toujours quelque part en mémoire. On dit que cette instance n'est plus

référencée.

> Le passage par copie de la référence vaut également pour les paramètres des méthodes.

> **=** est plus précisément l'opérateur d'initialisation et d'affectation. Pour une variable, l'initialisation se fait au moment de sa déclaration et pour un attribut, au moment de la création de l'objet.

```
// Initialisation
int a = 1;
```

L'affectation est une opération qui se fait, pour une variable, après sa déclaration et, pour un attribut, après la construction de l'objet.

```
int a;
// Affectation
a = 1;
```

Les opérateurs arithmétiques

Les opérateurs arithmétiques à deux opérandes sont :

Opérateurs arithmétiques

*	Multiplication
/	Division
%	Reste de la division
+	Addition
-	Soustraction

La liste ci-dessus est donnée par ordre de précédence. Cela signifie qu'une multiplication est effectuée avant une division.

```
int i = 2 * 3 + 4 * 5 / 2;
int j = (2 * 3) + ((4 * 5) / 2);
```

Les deux expressions ci-dessus donne le même résultant en Java : 16. Il est tout de même recommandé d'utiliser les parenthèses qui rendent l'expression plus facile à lire.

Les opérateurs arithmétiques unaires

Les opérateurs arithmétiques unaires ne prennent qu'un seul argument (comme l'indique leur nom), il s'agit de :

Opérateurs arithmétiques unaires

`expr++`	Incrément postfixé
`expr--`	Décrément postfixé
`++expr`	Incrément préfixé
`--expr`	Décrément préfixé
+	Positif
-	Négatif

```
int i = 0;
i++; // i vaut 1
++i; // i vaut 2
--i; // i vaut 1

int j = +i; // équivalent à int j = i;
int k = -i;
```

> Il y a une différence entre un opérateur postfixé et un opérateur préfixé lorsqu'ils sont utilisés conjointement à une affectation. Pour les opérateurs préfixés, l'incrément ou le décrément se fait **avant** l'affectation. Pour les opérateurs postfixés, l'incrément ou le décrément se fait **après** l'affectation.

```
int i = 10;
j = i++; // j vaudra 10 et i vaudra 11

int k = 10;
l = ++k; // l vaudra 11 et k vaudra 11
```

L’opérateur de concaténation de chaînes

Les chaînes de caractères peuvent être concaténées avec l’opérateur **+**. En Java, les chaînes de caractères sont des objets de type String. Il est possible de concaténer un objet de type String avec un autre type. Pour cela, le compilateur insérera un appel à la méthode *toString* de l’objet ou de la classe enveloppe pour un type primitif.

```
String s1 = "Hello ";
String s2 = s1 + " world";
String s3 = " !";
String s4 = s2 + s3;
```

> L’opérateur de concaténation correspond plus à du sucre syntaxique qu’à un véritable opérateur. En effet, il existe la classe StringBuilder dont la tâche consiste justement à nous aider à construire des chaînes de caractères. Le compilateur remplacera en fait notre code précédent par quelque chose dans ce genre :

```
String s1 = "Hello ";

StringBuilder sb1 = new StringBuilder();
sb1.append(s1)
sb1.append(s2);

String s2 = sb1.toString();
String s3 = " !";

StringBuilder sb2 = new StringBuilder();
sb2.append(s2)
sb2.append(s3);

String s4 = sb2.toString();
```

> Concaténer une chaîne de caractères avec une variable nulle ajoute la chaîne « null » :

```
String s1 = "test ";
String s2 = null;
String s3 = s1 + s2; // "test null"
```

Les opérateurs relationnels

Les opérateurs relationnels produisent un résultat booléen (**true** ou **false**) et permettent de comparer deux valeurs :

Opérateurs relationnels

<	Inférieur
>	Supérieur
<=	Inférieur ou égal
>=	Supérieur ou égal
==	Égal
!=	Différent

La liste ci-dessus est donnée par ordre de précédence. Les opérateurs **<**, **>**, **<=**, **>=** ne peuvent s'employer que pour des nombres ou des caractères (**char**).

Les opérateurs **==** et **!=** servent à comparer les valeurs contenues dans les deux variables. Pour des variables de type objet, ces opérateurs **ne comparent pas** les objets entre-eux mais simplement les références contenues dans ces variables.

```
Voiture v1 = new Voiture();
Voiture v2 = v1;

// true car v1 et v2 contiennent la même référence
boolean resultat = (v1 == v2);
```

> Les chaînes de caractères en Java sont des **objets** de type String. Cela signifie qu'il ne faut **JAMAIS** utiliser les opérateurs **==** et **!=** pour comparer des chaînes de caractères.

```
String s1 = "une chaîne";
String s2 = "une chaîne";

// sûrement un bug car le résultat est indéterminé
```

```
boolean resultat = (s1 == s2);
```

La bonne façon de faire est d'utiliser la méthode equals pour comparer des objets :

```
String s1 = "une chaîne";
String s2 = "une chaîne";

boolean resultat = s1.equals(s2); // OK
```

Les opérateurs logiques

Les opérateurs logiques prennent des booléens comme opérandes et produisent un résultat booléen (**true** ou **false**) :

Opérateurs relationnels

!	Négation
&&	Et logique
||	Ou logique

```
boolean b = true;
boolean c = !b // c vaut false

boolean d = b && c; // d vaut false
boolean e = b || c; // e vaut true
```

Les opérateurs **&&** et **||** sont des opérateurs qui n'évaluent l'expression à droite que si cela est nécessaire.

```
ltest() && rtest()
```

Dans l'exemple ci-dessus, la méthode **ltest** est appelée et si elle retourne **true** alors la méthode rtest() sera appelée pour évaluer l'expression. Si la méthode **ltest** retourne **false** alors le résultat de l'expression sera **false** et la méthode **rtest** ne sera pas appelée.

```
ltest() || rtest()
```

Dans l'exemple ci-dessus, la méthode **ltest** est appelée et si elle retourne **false** alors la méthode rtest() sera appelée pour évaluer l'expression. Si la méthode **ltest** retourne **true** alors le résultat de l'expression sera **true** et la méthode **rtest** ne sera pas appelée.

Si les méthodes des exemples ci-dessus produisent des effets de bord, il est parfois difficile de comprendre le comportement du programme.

> Il existe en Java les opérateurs **&** et **|** qui forcent l'évaluation de tous les termes de l'expression quel que soit le résultat de chacun d'entre eux.

```
ltest() | ctest() & rtest()
```

Dans l'expression ci-dessus, peu importe la valeur booléenne retournée par l'appel à ces méthodes. Elles seront toutes appelées puis ensuite le résultat de l'expression sera évalué.

L'opérateur ternaire

L'opérateur ternaire permet d'affecter une valeur suivant le résultat d'une condition.

```
exp booléenne ? valeur si vrai : valeur si faux
```

Par exemple :

```
String s = age >= 18 ? "majeur" : "mineur";
int code = s.equals("majeur") ? 10 : 20;
```

Les opérateurs *bitwise*

Les opérateurs *bitwise* permettent de manipuler la valeur des bits d'un entier.

Opérateurs bitwise

~	Négation binaire
&	Et binaire
^	Ou exclusif (XOR)
|	Ou binaire

```
int i = 0b1;

i = 0b10 | i; // i vaut 0b11
```

```
i = 0b10 & i; // i vaut 0b10

i = 0b10 ^ i; // i vaut 0b00

i = ~i; // i vaut -1
```

Les opérateurs de décalage

Les opérateurs de décalage s'utilisent sur des entiers et permettent de déplacer les bits vers la gauche ou vers la droite. Par convention, Java place le bit de poids fort à gauche quelle que soit la représentation physique de l'information. Il est possible de conserver ou non la valeur du bit de poids fort qui représente le signe pour un décalage à droite.

Opérateurs de décalage

`<<`	Décalage vers la gauche
`>>`	Décalage vers la droite avec préservation du signe
`>>>`	Décalage vers la droite sans préservation du signe

Puisque la machine stocke les nombres en base 2, un décalage vers la gauche équivaut à multiplier par 2 et un décalage vers la droite équivaut à diviser par 2 :

```
int i = 1;
i = i << 1 // i vaut 2
i = i << 3 // i vaut 16
i = i >> 2 // i vaut 4
```

Le transtypage (cast)

Il est parfois nécessaire de signifier que l'on désire passer d'un type vers un autre au moment de l'affectation. Java étant un langage fortement typé, il autorise par défaut uniquement les opérations de transtypage qui sont sûres. Par exemple : passer d'un entier à un entier long puisqu'il n'y aura de perte de données.

Si on le désire, il est possible de forcer un transtypage en indiquant explicitement le type attendu entre parenthèses :

```
int i = 1;
long l = i; // Ok
```

```
short s = (short) l; // cast obligatoire
```

L'opération doit avoir un sens. Par exemple, pour passer d'un type d'objet à un autre, il faut que les classes aient un lien d'héritage entre elles.

> Si Java impose de spécifier explicitement le transtypage dans certaines situations alors c'est qu'il s'agit de situations qui peuvent être problématiques (perte de données possible ou mauvais type d'objet). Il ne faut pas interpréter cela comme une limite du langage : il s'agit peut-être du symptôme d'une erreur de programmation ou d'une mauvaise conception.

> Le transtypage peut se faire également par un appel à la méthode Class.cast. Il s'agit d'une utilisation avancée du langage puisqu'elle fait intervenir la notion de réflexivité.

Opérateur et assignation

Il existe une forme compacte qui permet d'appliquer certains opérateurs et d'assigner le résultat directement à l'opérande de gauche.

Opérateurs avec assignation

Opérateur	Équivalent
+=	a = a + b
-=	a = a - b
***=**	a = a * b
/=	a = a / b
%=	a = a % b
&=	a = a & b
^=	a = a ^ b
|=	a = a | b
<<=	a = a << b
>>=	a = a >> b
>>>=	a = a >>> b

L'opérateur .

L’opérateur **.** permet d’accéder aux attributs et aux méthodes d’une classe ou d’un objet à partir d’une référence.

```
String s = "Hello the world";
int length = s.length();
System.out.println("La chaîne de caractères contient " + length  + " caractères");
```

> On a l’habitude d’utiliser l’opérateur **.** en plaçant à gauche une variable ou un appel de fonction. Cependant comme une chaîne de caractères est une instance de String, on peut aussi écrire :

```
int length = "Hello the world".length();
```

Lorsqu’on utilise la réflexivité en Java, on peut même utiliser le nom des types primitifs à gauche de l’opérateur **.** pour accéder à la classe associée :

```
String name = int.class.getName();
```

L’opérateur ,

L’opérateur virgule est utilisé comme séparateur des paramètres dans la définition et l’appel des méthodes. Il peut également être utilisé en tant qu’opérateur pour évaluer séquentiellement une instruction.

```
int x = 0, y = 1, z= 2;
```

Cependant, la plupart des développeurs Java préfèrent déclarer une variable par ligne et l’utilisation de l’opérateur virgule dans ce contexte est donc très rare.

Created 2025-02-10 11:53:25 UTC by Nicolas
Updated 2025-02-10 12:13:14 UTC by Nicolas