Les types primitifs

Java n'est pas complètement un langage orienté objet dans la mesure où il supporte ce que l'on nomme les *types primitifs*. Chaque type primitif est représenté par un mot-clé :

Types primitifs

Français	Anglais	Mot-clé
Booléen	Boolean	**boolean**
Caractère	Character	**char**
Entier	Integer	**int**
Octet	Byte	**byte**
Entier court	Short integer	**short**
Entier long	Long integer	**long**
Nombre à virgule flottante	Float number	**float**
Nombre à virgule flottante en double précision	Double precision float number	**double**

Une variable de type primitif représente juste une valeur stockée dans un espace mémoire dont la taille dépend du type. À la différence des langages comme C ou C++, l'espace mémoire occupé par un primitif est fixé par la spécification du langage et non par la machine cible.

Taille mémoire

Type	Espace mémoire	Signé
boolean	indéterminé	non
char	2 octets (16 bits)	non
int	4 octets (32 bits)	oui
byte	1 octet (8 bits)	oui
short	2 octets (16 bits)	oui
long	8 octets (64 bits)	oui

Type	Espace mémoire	Signé
float	4 octets (32 bits IEEE 754 floating point)	oui
double	8 octets (64 bits IEEE 754 floating point)	oui

Le type booléen : boolean

Les variables de type booléen ne peuvent prendre que deux valeurs : **true** ou **false**. Par défaut, un attribut de type **boolean** vaut **false**.

On ne peut utiliser que des opérateurs booléens comme **==**, **!=** et **!** sur des variables de type booléen (pas d'opération arithmétique autorisée).

Le type caractère : char

Les variables de type **char** sont codées sur 2 octets non signés car la représentation interne des caractères est l'UTF-16. Cela signifie que la valeur va de 0 à 2^16 - 1. Par défaut, un attribut de type **char** vaut **0** (c'est-à-dire le caractère de terminaison).

Pour représenter un littéral, on utilise l'apostrophe (**simple quote**) :

```
char c = 'a';
```

Même si les caractères ne sont pas des nombres, Java autorise les opérations arithmétiques sur les caractères en se basant sur le code caractère. Cela peut être pratique si l'on veut parcourir l'alphabet par exemple :

```
for (char i = 'a'; i <= 'z'; ++i) {
  // ...
}
```

On peut également affecter un nombre à une variable caractère. Ce nombre représente alors le code caractère :

```
char a = 97; // 97 est le code caractère de la lettre a en UTF-16
```

Affecter une variable de type entier à un variable de type **char** conduit à une erreur de compilation. En effet, le type **char** est un nombre signé sur 2 octets. Pour passer la compilation, il

faut transtyper (**cast**) la variable :

```
int i = 97;
char a = (char) i; // cast vers char obligatoire pour la compilation
```

Les types entiers : byte, short, int, long

Les types entiers différent entre-eux uniquement par l'espace de stockage mémoire qui leur est alloué. Ils sont tous des types signés. Par défaut, un attribut de type **byte**, **short**, **int** ou **long** vaut 0.

La règle de conversion implicite est simple : on peut affecter une variable d'un type à une variable d'un autre type que si la taille mémoire est au moins assez grande.

```
byte b  = 1;
short s = 2;
int i   = 3;
long l  = 4;

// conversion implicite ok
// car la variable à droite de l'expression
// est d'une taille mémoire inférieure
s = b;
i = s;
i = b;
l = b;
l = s;
l = i;
```

Dans tous les autres cas, il faut réaliser un transtypage avec un risque de perte de valeur :

```
b = (byte) s;
s = (short) i;
i = (int) l;
```

Lorsque vous affectez une valeur littérale à une variable, le compilateur contrôlera que la valeur est acceptable pour ce type :

```
byte b = 0;
b = 127; // ok
```

```
b = 128; // ko car le type byte accepte des valeurs entre -128 et 127
```

Les valeurs littérales peuvent s'écrire suivant plusieurs bases :

Écriture des valeurs entières littérales

Base	Exemple
2 (binaire)	0b0010 ou 0B0010
8 (octal)	0174
10 (décimal)	129
16 (hexadécimal)	0x12af ou 0X12AF

On peut forcer une valeur littérale à être interprétée comme un entier long en suffixant la valeur par **L** ou **l** :

```
long l = 100L;
```

Pour plus de lisibilité, il est également possible de séparer les milliers par _ :

```
long l = 1_000_000;
```

> Les opérations arithmétiques entre des valeurs littérales sont effectuées à la compilation. Il est souvent plus lisible de faire apparaître l'opération plutôt que le résultat :

```
int hourInMilliseconds = 60 * 60 * 1000 // plutôt que 3_600_000
```

> La représentation interne des nombres entiers fait qu'il est possible d'aboutir à un dépassement des valeurs maximales ou minimales (*buffer overflow* ou *buffer underflow*) . Il n'est donc pas judicieux d'utiliser ces types pour représenter des valeurs qui peuvent croître ou décroître sur une très grande échelle. Pour ces cas-là, on peut utiliser la classe BigInteger qui utilise une représentation interne plus complexe.

Les types à virgule flottante : float, double

Les types **float** et **double** permettent de représenter les nombres à virgule selon le format IEEE 754. Ce format stocke le signe sur un bit puis le nombre sous une forme entière (la mantisse) et l'exposant en base 2 pour positionner la virgule. Par défaut, un attribut de type **float** ou **double**

vaut 0.

float est dit en simple précision et est codé sur 4 octets (32 bits) tandis que **double** est dit en double précision et est codé sur 8 octets (64 bits).

Il est possible d'ajouter une valeur entière à un type à virgule flottante mais l'inverse nécessite une transtypage (**cast**) avec une perte éventuelle de valeur.

```
int i = 2;
double d = 5.0;
d = d + i;
i = (int) (d + i);
```

Les valeurs littérales peuvent s'écrire avec un **.** pour signifier la virgule et/ou avec une notation scientifique en donnant l'exposant en base 10 :

```
double d1 = .0; // le 0 peut être omis à gauche de la virgule
double d2 = -1.5;
double d3 = 1.5E1; // 1.5 * 10, c'est-à-dire 15.0
double d4 = 0.1234E-15;
```

Une valeur littérale est toujours considérée en double précision. Pour l'affecter à une variable de type **float**, il faut suffixer la valeur par **F** ou **f** :

```
float f = 0.5f;
```

> La représentation interne des nombres à virgule flottante fait qu'il est possible d'aboutir à des imprécisions de calcul. Il n'est donc pas judicieux d'utiliser ces types pour représenter des valeurs pour lesquelles les approximations de calcul ne sont pas acceptables. Par exemple, les applications qui réalisent des calculs sur des montants financiers ne devraient **jamais** utiliser des nombres à virgule flottante. Soit il faut représenter l'information en interne toujours en entier (par exemple en centimes d'euro) soit il faut utiliser la classe BigDecimal qui utilise une représentation interne plus complexe mais sans approximation.

Les classes enveloppes

Comme les types primitifs ne sont pas des classes, l'API standard de Java fournit également des classes qui permettent d'envelopper la valeur d'un type primitif : on parle de **wrapper classes**.

Wrapper classes

Type	Classe associée

boolean	java.lang.Boolean
char	java.lang.Character
int	java.lang.Integer
byte	java.lang.Byte
short	java.lang.Short
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

> Le tableau ci-dessus donne le nom complet des classes, c'est-à-dire en incluant le nom du package (*java.lang*).

Il est possible de créer une instance d'une classe enveloppe soit en utilisant son constructeur soit en utilisant la méthode de classe **valueOf** (il s'agit de la méthode recommandée).

```
Integer i = Integer.valueOf(2);
```

Pour obtenir la valeur enveloppée, on fait appel à la méthode *xxxValue()*, xxx étant le type sous-jacent :

```
Integer i = Integer.valueOf(2);
int x = 1 + i.intValue();
```

Pourquoi avoir créé ces classes ? Cela permet d'offrir un emplacement facile à mémoriser à des méthodes utilitaires. Par exemple, toutes les classes enveloppes définissent une méthode de classe de la forme *parseXXX* qui permet de convertir une chaîne de caractères en un type primitif :

```
boolean b = Boolean.parseBoolean("true");
byte by = Byte.parseByte("1");
short s = Short.parseShort("1");
int i = Integer.parseInt("1");
long l = Long.parseLong("1");
float f = Float.parseFloat("1");
double d = Double.parseDouble("1");
// enfin presque toutes car Character n'a pas cette méthode
```

Une variable de type d'une des classes enveloppes référence un objet donc elle peut avoir la valeur spéciale **null**. Ce cas permet de signifier l'absence de valeur.

Les classes enveloppes contiennent des constantes pour donner des informations utiles. Par exemple, la classe java.lang.Integer déclare les constantes MIN_VALUE et MAX_VALUE qui donnent

respectivement la plus petite valeur et la plus grande valeur représentables par la primitive associée.

Enfin les classes enveloppes sont conçues pour être non modifiables. Cela signifie que l'on ne peut pas modifier la valeur qu'elles enveloppent après leur création.

L'autoboxing

Il n'est pas rare dans une application Java de devoir convertir des types primitifs vers des instances de leur classe enveloppe et réciproquement. Afin d'alléger la syntaxe, on peut se contenter d'affecter une variable à une autre et le compilateur se chargera d'ajouter le code manquant. L'opération qui permet de passer d'un type primitif à une instance de sa classe enveloppe s'appelle le **boxing** et l'opération inverse s'appelle **l'unboxing**.

Le code suivant

```
Integer i = 1;
```

est accepté par le compilateur et ce dernier lira à la place

```
Integer i = Integer.valueOf(1); // boxing
```

De même, le code suivant

```
Integer i = 1;
int j = i;
```

est également accepté par le compilateur et ce dernier lira à la place

```
Integer i = Integer.valueOf(1); // boxing
int j = i.intValue(); // unboxing
```

On peut ainsi réaliser des opérations arithmétiques sur des instances de classes enveloppes

```
Integer i = 1;
Integer j = 2;
Integer k = i + j;
```

Il faut bien comprendre que le code ci-dessus manipule en fait des objets et qu'il implique plusieurs opérations de boxing et de unboxing. Si cela n'est pas strictement nécessaire, alors il vaut mieux utiliser des types primitifs.

L’autoboxing fonctionne à chaque fois qu’une affectation a lieu. Il s’applique donc à la déclaration de variable, à l’affection de variable et au passage de paramètre.

L’autoboxing est parfois difficile à utiliser car il conduit à des expressions qui peuvent être ambiguës. Par exemple, alors que le code suivant utilisant des primitives compile :

```
int i = 1;
float j = i;
```

Ce code faisant appelle à l’autoboxing ne compile pas en l’état :

```
Integer i = 1;
Float j = i; // ERREUR : i est de type Integer
```

Pire, l’autoboxing peut être source de bug. Le plus évident est l’unboxing d’une variable nulle :

```
Integer i = null;
int j = i; // ERREUR : unboxing de null !
```

Une variable de type **Integer** peut être **null**. Dans ce cas, l’unboxing n’est pas possible et aboutira à une erreur (NullPointerException). Si cet exemple est trivial, il peut être beaucoup plus subtil et difficile à comprendre pour un projet de plusieurs centaines (milliers) de lignes de code.

Revision #1
Created 10 February 2025 11:50:10 by Nicolas
Updated 10 February 2025 12:13:14 by Nicolas