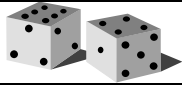


Cours n°: 2 Les types de base.



Objectifs : Connaître les différents types de base de Java.
Difficultés : Aucune, hormis beaucoup d'éléments fondamentaux.

La déclaration de variables

Une variable possède un nom, un type et une valeur. La déclaration d'une variable doit donc contenir deux choses : un nom et le type de données qu'elle peut contenir. Une variable est utilisable dans le bloc où elle est définie. La déclaration d'une variable permet de réserver la mémoire pour en stocker la valeur. Le type d'une variable peut être un type élémentaire ou un objet :

type_élémentaire variable;
classe variable ;

Exemple :

long nombre;

Rappel : les noms de variables en Java peuvent commencer par une lettre, par le caractère de soulignement ou par le signe dollars. Le reste du nom peut comporter des lettres ou des nombres mais jamais d'espaces. Il est possible de définir plusieurs variables de même type en séparant chacune d'elles par une virgule.

Exemple :

int jour, mois, annee ;

Java est un langage à typage rigoureux qui ne possède pas de transtypage automatique lorsque ce transtypage risque de conduire à une perte d'information. Pour les objets, il est nécessaire en plus de la déclaration de la variable de créer un objet avant de pouvoir l'utiliser. Il faut réserver de la mémoire pour la création d'un objet (remarque : un tableau est un objet en java) avec l'instruction new. La libération de la mémoire se fait automatiquement grâce au garbage collector.

Exemple :

MaClasse instance; // déclaration de l'objet
instance = new MaClasse(); // création de l'objet
OU MaClasse instance = new MaClasse(); // déclaration et création de l'objet

Saisissez et testez le programme suivant

Fichier "tp21.java"

```
public class tp21 {

    public static void main(String params[]){
        // Définition de deux variables entières i et j
        // i sera de plus initialisée avec la valeur 15
        int i=15 , j;

        // Définition d'une chaîne de caractères avec
        // comme valeur initiale "Cours Java"
        String chaine = "Cours Java";

        // On donne une valeur à j
        j = 10;

        // On utilise ces variables en les affichant
        System.out.println("i = " + i + " et j = " + j);
        System.out.println("chaine = " + chaine);
    }
}
```

Les types numériques

Le langage propose un grand nombre de types numériques. On peut classer ces types en deux catégories : les types entiers et les types décimaux. Ce qui différencie les types d'une même catégorie, c'est la taille de mémoire utilisée pour contenir une valeur. Plus cette taille est grande, plus les valeurs utilisables sont nombreuses. Vous choisissez donc un type en fonction de l'étendu de valeur souhaitée pour vos variables.

Les types numériques entiers

Quatre types d'entiers vous sont proposés. La différence réside dans la taille nécessaire pour stocker la valeur. Le tableau qui suit précise un petit peu mieux les choses. Le bit y est utilisé pour décrire les tailles. Notez qu'un bit ne peut prendre que deux valeurs (0 ou 1). En conséquence, n bits ne peuvent définir que 2^n valeurs. On regroupe traditionnellement (c'est lié à des aspects électroniques) les bits par groupe de huit, ce qui forme un octet (byte en anglais). Un octet peut donc contenir 2^8 (soit 256) valeurs distinctes.

	byte	short	int	long
Taille (bits)	8	16	32	64
Etendue	0 .. 255	-32768 .. 32767	$-2^{31} .. 2^{31}-1$	$-2^{63} .. 2^{63}-1$

Les types numériques décimaux

De même que pour les types numériques, vous avez deux types pour les nombres décimaux, la seule différence résidant dans la taille utilisée pour stocker une valeur de ce type.

	float	double
Size (bits)	32	64
Exemple de valeurs	3.25f	3.25

Le type booléen

On introduit une variable de ce type pas le mot clé boolean. Ce type accepte seulement deux états : l'un est nommé true et symbolise un état d'acceptation, l'autre, nommé false, symbolise un état de réfutation. Attention, ce n'est plus comme en C : le type booléen n'est pas en Java, un sous-type numérique (je rappelle qu'en C, la valeur 0 est considérée comme fausse et les autres valeurs entières comme vraies).

Le type caractère

Ce type, introduit par le mot clé char, permet la gestion des caractères (comme son nom le laisse présager). Jusqu'à présent, la plupart des langages utilisaient à cet effet le codage ASCII (ou des dérivés). Le codage ASCII permet de représenter seulement 128 caractères : en effet à l'origine, seul les sept bits de poids faibles d'un octet servaient au codage du caractère, le huitième servait alors de bit de parité assurant le bon transport du caractère à travers le réseau. Ensuite, des couches de protocole de transport de données plus sûres ont rendus ce huitième bit utilisable. Différents systèmes dérivés d'ASCII ont fait leur apparition (la table ISO-Latin-1, ...). Malheureusement, aucun de ces standards n'a su s'imposer de manière universelle, et l'échange de données entre différents systèmes reste très problématique.

Pour palier les problèmes engendrés, un nouveau standard de codage de caractères, universel, fut proposé : ce codage est nommé Unicode. Unicode utilise 16 bits pour représenter un caractère et permet donc, potentiellement, de coder 65536 caractères et donc nos caractères accentués, tout comme d'autres caractères provenant d'autres alphabets (Cyrillique, Hébreux, Arabe, Chinois, Grec, ...). Les concepteurs de Java, dans un souci de portabilité du code produit, ont donc trouvés judicieux d'utiliser ce standard. Les exemples qui suivent vous donne la syntaxe à utiliser pour décrire un caractère en Java.

'a'	'\t' pour une tabulation	'\u0521' un caractère quelconque
-----	--------------------------	----------------------------------

Le type chaîne de caractères

Première remarque, par rapport au langage C, le type chaîne de caractères est totalement différent du type tableau de caractères (la description des tableaux Java se trouve quelques paragraphes plus bas). Pour ce qui est de la syntaxe d'une chaîne de caractères, rien n'a changé par rapport à C ou C++ : une chaîne de caractères commence par un caractère double guillemets et se termine aussi par ce même signe. Ainsi, "Dominique" est une chaîne de caractères. Là où ça se complique un petit peu, c'est que le type d'une telle valeur est String, qui est une classe fournie par la machine virtuelle.

Les expressions numériques

Les opérateurs que je nomme numériques, calculent tous un résultat numérique. Cependant, il faut bien noter qu'il y a plusieurs opérateurs qui ont le même nom (par exemple les opérateurs nommés +) mais qui s'appliquent sur des types différents : l'exemple qui suit vous fixera mieux les idées.

<code>int i = 4 + 3;</code>	Dans ce cas, l'opérateur + prend deux entiers (chacun sur 32 bits) et rend un entier, lui aussi représenté sur 32 bits.
<code>double d = 3.2 + 1.84;</code>	Ici, l'opérateur d'addition, prend deux opérandes de types décimaux (représenté chacun sur 64 bits) et retourne un flottant du même type.

En règle générale, il y a autant d'opérateurs + que de types numériques. Mais la question est alors de savoir ce qui doit se passer si l'on tente d'ajouter un entier avec un décimal. Il est clair que le résultat doit aussi être décimal. En conséquence, c'est le deuxième cas donné en exemple qui s'applique, avec une phase implicite de conversion de la valeur entière en valeur décimale. Cette phase de conversion est plus connue sous le nom de **casting**. Toujours en règle générale, un **cast est implicitement effectué** s'il est nécessaire, et **s'il n'y a pas de perte de précision sur la valeur numérique**. Il est donc clair qu'un **cast** implicite d'un décimal en un entier ne sera jamais possible. Il faut alors explicitement demander le **cast**. L'exemple qui suit vous donne la syntaxe à utiliser : il suffit de préfixer la valeur par le type, mit entre parenthèses.

<code>int i = 4 + (int)3.8;</code>	Dans ce cas, l'opérateur + prend deux entiers (chacun sur 32 bits) et rend un entier, lui aussi représenté sur 32 bits. Le résultat est 7 car 3.8 donne 3 après le cast.
------------------------------------	--

Une fois cela assimilé il faut comprendre encore deux ou trois petites choses. Un opérateur, par définition, prend un certain nombre d'expressions (ou opérandes) pour calculer une valeur qui pourra à son tour être utilisée par un autre opérateur. En d'autres termes, une expression est constituée d'opérateurs et de sous expressions, comme le montre l'exemple suivant.

<code>int i = 3 * 6 + 2 * 9;</code>

Finalement, la variable i de type entier sur 32 bits, contiendra la valeur 36. En effet, des règles de priorité sur les opérateurs, impliquent qu'un opérateur va primer sur un autre selon les cas de figure. Donc en reprenant l'exemple suivant, la valeur est calculée en faisant la somme du produit de 3 par 6 et de 2 par 9 (soit 18 + 18). Se reporter à la table de précedence des opérateurs pour plus d'informations.

Nous trouvons notamment, comme opérateurs numériques, la somme (+), la soustraction (-), la multiplication (*), la division (/). Ils s'appliquent tous sur des opérandes numériques quelconques (entières ou flottantes). On en trouve aussi qui s'applique uniquement à des opérandes entières : le reste de la division entière (%), le et bit à bit (&), le ou inclusif bit à bit (|) ou encore le ou exclusif bit à bit (^), le décalage à gauche, bit à bit, (<< équivalent à une multiplication par 2) et le décalage à droite (>> équivalent à une division entière par 2).

Nous trouvons aussi deux opérateurs permettant d'incrémenter (++) ou de décrémenter (--) d'une unité une variable numérique (entière ou flottante). Il faut noter que ces deux opérateurs sont unaires et surtout que l'on peut les utiliser en préfixe ou en postfixe d'une variable. Dans ce cas l'action sémantique n'est pas la même. Pour comprendre cela, regardons l'exemple qui suit. N'oubliez pas de le saisir et de le compiler.

Fichier "tp22.java"
<pre>public class tp22 { public static void main(String args[]){ int a = 10; System.out.println("a = " + a + ";\t(++a*2) = " + (++a*2) + ";\ta = " + a + ";\n"); a = 10; System.out.println("a = " + a + ";\t(a++*2) = " + (a++*2) + ";\ta = " + a + ";\n"); } }</pre>
Résultats
<pre>a = 10; (++a*2) = 22; a = 11; a = 10; (a++*2) = 20; a = 11;</pre>

Bon si vous avez toujours pas fuit devant ces quelques lignes indéchiffrables, je vais pouvoir tenter de vous expliquer ce qu'elles font, et pourquoi elles le font. Donc, comme dans le cas de votre première application Java, le programme commence par la définition d'une classe (ici nommée `essai`). Ensuite, on y spécifie notre seule méthode, qui doit servir de point de départ à l'application : on la nomme donc `main` (nous reviendrons sur la nécessité du **static** ultérieurement, sachez seulement qu'il est obligatoire pour la méthode `main`, idem pour le **public**).

La ligne qui suit sert à créer une variable nommée `a` à laquelle on affecte la valeur 10. Ensuite, tout se complique, mais ça reste du domaine du compréhensible. **System** est un objet prédéfini en Java qui permet d'avoir accès au système (comme son nom l'indique). Cet objet contient dans ses attributs un autre objet nommé **out** qui lui donne accès au flot de données de sortie de l'application, et on applique la méthode **println** qui y place un message (passé en paramètre) ainsi qu'un retour chariot.

Toute la difficulté réside dans le fait de comprendre le message. Ce dernier commence par une chaîne de caractères suivit d'un opérateur `+`. Il est bien clair, qu'il ne s'agit donc pas de l'opérateur dont on a déjà parlé : celui-ci réalise la concaténation de deux chaînes de caractères (nous en reparlerons). Je vous sens alors choqué : ce qui suit n'est pas une chaîne de caractères. Mais par contre, en Java, tous peuvent s'exprimer en ce type. Comme il n'y a pas, à priori, de perte de données, le casting d'un nombre en une chaîne de caractères est donc accepté, et le tour est joué. A force de pratiquer, ces petites astuces vous deviendront familières.

Ce n'est pas fini, on trouve ensuite une autre chaîne de caractères contenant les deux caractères `"\t"` : ils servent à placer une tabulation dans la chaîne. Ensuite on trouve notre fameux opérateur `++` (et oui, vous l'aviez déjà oublié, mais il est toujours le sujet principal de notre débat) : lors de sa première apparition, il est utilisé en préfixe, tandis qu'il l'est en postfixe dans sa seconde apparition. Si vous regardez les résultats (ceux affichés par les **System.out.println**, vous vous apercevrez que la valeur calculée n'est pas la même, bien que l'incréméntation de la variable `a` soit faite. La raison en est très simple : dans le premier cas, on incrémente la variable, et on finit le calcul. Dans le second cas, on effectue le calcul, puis ensuite, on n'oublie surtout pas d'incrémenter la variable. Faites les calculs à la main et vous verrez que vous trouverez alors les mêmes valeurs. Est-ce maintenant plus clair ? Si oui, sachez que l'opérateur `--` fonctionne de même (bien qu'il effectue une décrémentation). Si au contraire ce n'est pas le cas, faites une pause, prenez un café, et reprenez ce paragraphe.

Les expressions booléennes

De même qu'il existe des expressions qui calculent des résultats numériques, il en existe qui calculent des résultats booléens (c'est à dire un résultat pouvant être interprété soit comme vrai, soit comme faux). Il existe donc des opérateurs booléens nécessaires à la construction des ces expressions. Nous y trouvons notamment des prédicats (autre façon de nommer ces opérateurs) de comparaison numérique (`==`, il en existe autant que de types numériques et des conversions peuvent exister), de différenciation (`!=`), d'infériorité (strictes `<` ou non `<=`), de supériorité (`>` et `>=`). Nous trouvons aussi le ou logique (`||`) et le et logique (`&&`) qui prennent tous deux des opérandes booléennes. Il existe encore l'opérateur unaire (qui prend uniquement une seule opérande) de négation (`!`).

<pre>boolean egal=(3==4); int a=3,b=4; boolean diff=((a<b) a>b));</pre>	La variable <code>egal</code> vaut donc à cet instant false. la variable <code>diff</code> sera elle vraie soit si <code>a</code> est plus petite que <code>b</code>, soit si elle est plus grande (donc différente). Sinon <code>diff</code> sera faux.
<pre>boolean bool1=(3==4); boolean bool2=(3.0==4.0);</pre>	Il est bien clair que dans ces deux cas, les opérateurs de comparaisons ne sont les mêmes. Un prend deux opérandes entières, l'autres deux opérandes décimales.

Les expressions retournant des chaînes de caractères

Comme nous l'avons dit précédemment, un opérateur agissant sur des chaînes de caractères, permet de constituer des expressions de type chaîne de caractères. L'opérateur + permet de concaténer deux chaînes pour en former une nouvelle (nous avons déjà vu un exemple d'utilisation il y a quelques paragraphes).

Fichier "tp23.java"
<pre>public class tp23 { public static void main(String args[]){ String a = "Le début et ..."; String b = " la fin !!!"; String c = a + b; System.out.println(c); } }</pre>
Résultat :
Le début et ... la fin !!!

Une chose utile à se rappeler est que toute valeur en Java (pas forcément numérique) peut s'afficher sous forme de chaîne de caractères. L'expression ("a "+1) aura donc pour valeur "a 1".

Les affectations et les opérateurs associés

Une affectation est aussi une expression qui calcule une valeur et l'affecte à une variable. La valeur d'une telle expression est celle calculée dans la partie droite de l'affectation (les opérateurs d'affectation étant binaires infixes). L'opérateur permettant de définir une affectation sera noté =.

Fichier "tp24.java"
<pre>public class tp24 { public static void main(String args[]){ int a; System.out.println("Premier = " + (a=10)*2 + ";\t" + "Second = " + a + ";"); } }</pre>
Résultat :
Premier = 20; Second = 10;

A partir de là, et dans un souci de convivialité, d'autres opérateurs d'affectation sont définis. Le principe reste quasiment le même, à la différence que la valeur qui va être affectée à la variable, est calculé en fonction de l'état de cette variable. Pour mieux se fixer les idées regardons le tableau suivant qui fixe les équivalences entre ces nouveaux opérateurs et l'opérateur d'affectation classique.

a += b;	a = a + b;
a -= b;	a = a - b;
a *= b;	a = a * b;
a /= b;	a = a / b;
a %= b;	a = a % b;
a &= b;	a = a & b;
a = b;	a = a b;
a ^= b;	a = a ^ b;
a >>= b;	a = a >> b;
a <<= b;	a = a << b;

Les autres opérateurs du langage

Il existe encore d'autres opérateurs dont une petite description sera donnée dans le tableau suivant. Nous serons, par la suite amenés à en reparler.

new	Opérateur d'allocation mémoire <code>int tableau[]=new int[10];</code>
[]	Accès aux éléments d'un tableau <code>b = tableau[indice];</code> Attention, les indices d'un tableau commencent toujours à partir de zéro. Si un tableau contient dix éléments, les indices varieront donc entre 0 et 9.
.	Opérateur de traversé, utilisé pour accéder aux champs (attributs et méthodes) des objets. <code>int value = objet.champQuelconque;</code>
?:	L'opérateur conditionnel : c'est le seul opérateur ternaire (qui prend trois opérandes) du langage. Il permet de calculer une expression plutôt qu'une autre selon la valeur booléenne d'une autre expression. L'expression qui sert à choisir quelle autre expression calculer se place devant le point d'interrogation. Celle qui sera calculée si le test est vrai doit être mise entre le point d'interrogation et les deux points, la dernière, après les deux points. <code>boolean b = true;</code> <code>// plein de code Java</code> <code>int value = b ? 3*a+z : 3*a-k;</code>

Précédence des opérateurs

Comme vous l'avez certainement déjà remarqué, il est possible de se servir de parenthèses pour fixer l'ordre d'évaluation des sous-expressions formant une expression Java. Ainsi, les expressions `3*2+4` et `3*(2+4)` n'auront pas les mêmes valeurs (10 et 18). Mais attention tout de même : l'utilisation abusive des parenthèses rend très rapidement un code illisible. Dans le but de ne pas arriver dans de telles situations, sachez que des règles de priorités ont été définies sur les opérateurs. Le tableau suivant fixe cet ordre : du plus prioritaire à l'opérateur de séquençement (,) qui est le moins prioritaire. Dorénavant, utilisez donc bien les parenthèses.

[] .
! ++ --
* / %
+ -
<<>>
== != <<= >=
&
^
&&
?:
= += -= *= /= %= <<= >= &= ^= =
,

Nous venons donc de survoler l'ensemble des opérateurs, ainsi que quelques expressions du langage. Ils nous seront fort utiles pour calculer des valeurs (numériques ou autres) dans nos programmes.

Voici quelques classes qui vont vous permettre de manipuler les types de variables.

Fichier tp25.java	Fichier tp26
<pre> public class tp25 { public static void main (String[] args) { int n ; double x ; n = 5 ; x = 2*n + 1.5 ; System.out.println ("n = " + n) ; System.out.println ("x = " + x) ; double y ; y = n * x + 12 ; System.out.println ("y = " + y) ; } } </pre>	<pre> public class tp26 { public static void main (String args[]) { char code = 'Z'; int index = code - 'A'; System.out.println("index = " + index); } } </pre>
Fichier tp27.java	Fichier tp28.java Attention piegeux
<pre> public class tp27 { public static void main (String args[]) { int n1=0; int n2=0; System.out.println("n1 = " + n1 + " n2 = " + n2); n1=n2++; System.out.println("n1 = " + n1 + " n2 = " + n2); n1=++n2; System.out.println("n1 = " + n1 + " n2 = " + n2); n1=n1++; //attention System.out.println("n1 = " + n1 + " n2 = " + n2); } } </pre>	<pre> public class tp28 { public static void main(String[] args) { Integer n1 = new Integer(47); Integer n2 = new Integer(47); System.out.println(n1 == n2); System.out.println(n1 != n2); } } </pre>



Remarque tp28 : Les opérateurs relationnels == et != fonctionnent avec tous les objets, mais leur utilisation déroute souvent le programmeur Java novice. L'expression **System.out.println(n1 == n2)** imprimera le résultat de la comparaison de type **boolean**. Il semble à priori évident que la sortie sera **true** puis **false**, puisque les deux objets de type **Integer** sont identiques. Mais, bien que le *contenu* des objets soit le même, les *références* sont différentes, et il se trouve que les opérateurs == and != comparent des références d'objet. En réalité la sortie sera **false** puis **true**. Naturellement, cela en surprendra plus d'un. Que faire si on veut comparer le contenu réel d'un objet ? Il faut utiliser la méthode spéciale **equals()** qui existe pour tous les objets (mais non pour les types primitifs, qui s'accommodent mieux de == et !=). Voici comment l'utiliser : Tp29.java

Fichier tp29.java
<pre> public class tp29 { public static void main(String[] args) { Integer n1 = new Integer(47); Integer n2 = new Integer(47); System.out.println(n1.equals(n2)); } } </pre>

Comme vous avez bien travaillé, je ne vous donne pas d'exercices, je préfère que vous relisiez cette leçon. A bientôt pour la troisième partie.