CÉGEP
MARIE-VICTORIN

420-210-MV
Programmation orientée objet

Samuel Fostiné

Les constantes

Dans une application, il arrive fréquemment que l'on utilise des valeurs numériques ou chaînes de caractères qui ne seront pas modifiées pendant le fonctionnement de l'application. Il est conseillé, pour faciliter la lecture du code, de définir ces valeurs sous forme de constantes.

La définition d'une constante se fait en ajoutant le mot-clé `final` devant la déclaration d'une variable. Il est obligatoire d'initialiser la constante au moment de sa déclaration (c'est le seul endroit où il est possible de faire une affectation à la constante).

```
final double TAUXTVA=1.20;
```

La constante peut être alors utilisée dans le code à la place de la valeur littérale qu'elle représente.

```
prixTtc=prixHt*TAUXTVA;
```

Les règles concernant la durée de vie et la portée des constantes sont identiques à celles concernant les variables.

La valeur d'une constante peut également être calculée à partir d'une autre constante.

```
final int TOTAL=100;
final int DEMI=TOTAL/2;
```

De nombreuses constantes sont déjà définies au niveau du langage Java. Elles sont définies comme membres `static` des nombreuses classes du langage. Par convention, les noms des constantes sont orthographiés entièrement en majuscules.

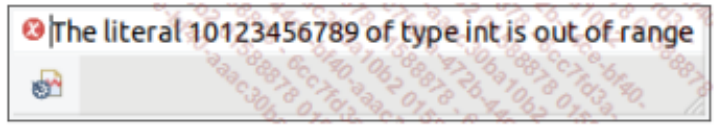
Les différents types d'entiers

- Java dispose de quatre types entiers, correspondant chacun à des emplacements mémoires de taille différente, donc à des limitations différentes.
- Le tableau suivant en récapitule leurs caractéristiques. Notez l'existence de constantes prédéfinies de la forme « *Integer.MAX_VALUE* » qui fournissent les différentes limites.

Type	Taille (octets)	Valeur minimale	Valeur maximale
byte	1	-128 (Byte.MIN_VALUE)	127 (Byte.MAX_VALUE)
short	2	-32 768 (Short.MIN_VALUE)	32 767 (Short.MAX_VALUE)
int	4	-2 147 483 648 (Integer.MIN_VALUE)	2 147 483 647 (Integer.MAX_VALUE)
long	8	-9 223 372 036 854 775 808 (Long.MIN_VALUE)	9 223 372 036 854 775 807 (Long.MAX_VALUE)

```
long l = 10123456789;
```

Je vous rappelle que le compilateur interprète les valeurs littérales numériques entières comme des valeurs de type `int`. L'erreur rencontrée avec l'exemple précédent est la suivante :



Pour corriger ce problème, il faut indiquer au compilateur d'interpréter la valeur littérale comme une valeur de type `long`. Pour réaliser cette opération, il suffit de suffixer le nombre par la lettre `L` (pour `long`) en majuscule ou minuscule :

```
long l = 10123456789L;
```

Les valeurs numériques décimales peuvent être exprimées avec la notation décimale ou la notation scientifique.

```
float surface=2356.8f;
float surface=2.3568e3f;
```

Vous pouvez insérer des caractères _ dans les valeurs numériques littérales pour faciliter la lecture. Les deux syntaxes suivantes sont équivalentes.

```
long quantite=1234876567;
long quantite=1_234_876_567;
```

L’utilisation de valeurs littérales

Il est très fréquent d’initialiser les variables d’un type numérique avec une valeur littérale :

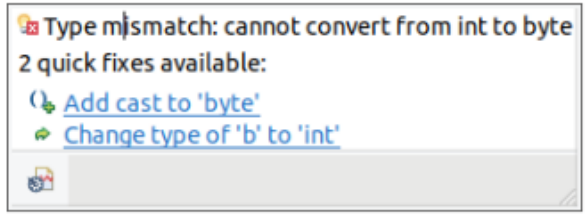
```
int populationFrancaise = 66990000;
double pourcentageFemme = 51.649;
```

Le compilateur (`javac`) interprète les valeurs numériques entières comme des valeurs de type int et les valeurs numériques décimales comme des valeurs de type `double`. En conséquence, les deux déclarations ci-dessus compilent.

Dans les déclarations suivantes, une erreur de compilation intervient :

```
byte b = 153;
```

Le compilateur indique l’erreur suivante :



La valeur `153` est en dehors de la plage de valeurs acceptable par une variable de type `byte`. Le compilateur indique qu’il est impossible pour lui de convertir cette valeur de type `int` et en `byte`. Si la valeur définie est comprise dans la plage du type, le problème disparaît.

Le même comportement est observable pour le type `short` lorsque l’on tente d’initialiser la variable avec une valeur en dehors de la plage acceptable par ce type.

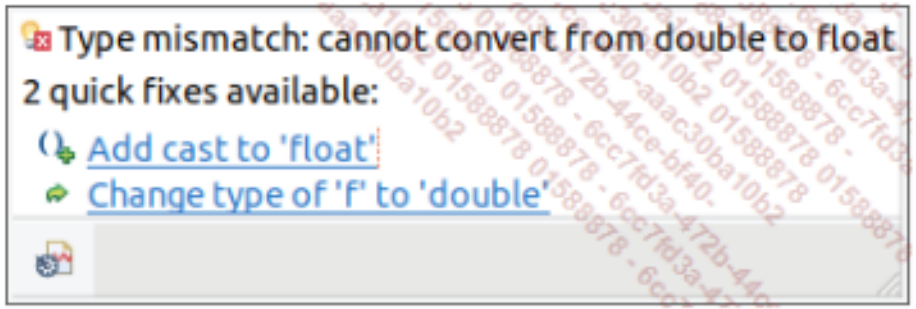
Les types flottants

- Java possède deux types de flottant correspondant chacun à des emplacements mémoires de tailles différentes : float et double.
- Le tableau suivant en récapitule leurs caractéristiques. Notez l'existence de constantes prédéfinies de la forme « *Float.MAX_VALUE* » et « *Double.MAX_VALUE* » qui fournissent les différentes limites.

Type	Taille (octets)	Valeur minimale absolue	Valeur maximale absolue
float	4	-1.40239846E-45 (Float.MIN_VALUE)	3.40282347E38 (Float.MAX_VALUE)
double	8	4.9406564584124654E-324 (Double.MIN_VALUE)	1.79769313486231 6E308 (Double.MAX_VALUE)

```
float f = 10.1;
```

Le compilateur indique l'erreur suivante :

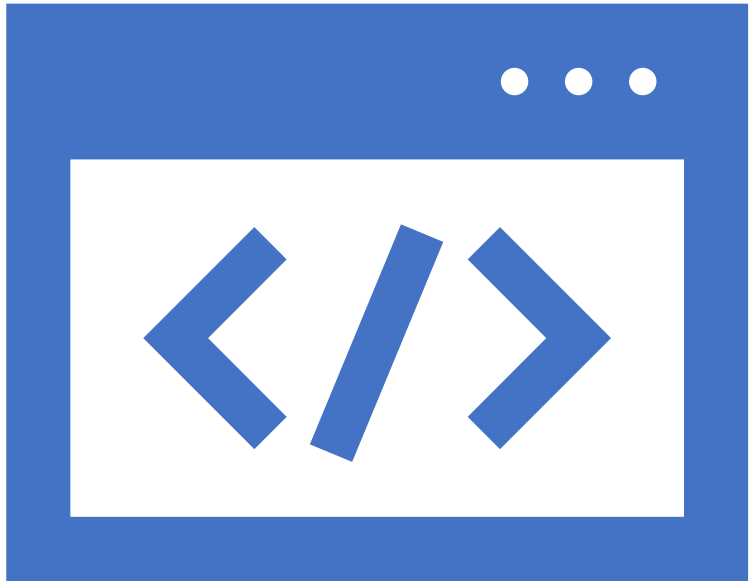


Pour corriger ce problème, il faut indiquer au compilateur d'interpréter la valeur littérale comme une valeur de type `float`. Pour réaliser cette opération, il suffit de suffixer le nombre par la lettre `F` (pour `float`) en majuscule ou minuscule :

```
float f = 10.1F;
```

Les types caractères

- Comme la plupart des langages, Java permet de manipuler des caractères. Mais il offre l'originalité de les représenter en mémoire sur deux octets.



Les types booléens

Les types de données booléens sont utilisés pour représenter quelque chose qui n'a que deux réponses possibles, par exemple une question Oui/Non.

Booléen est utilisé pour représenter les valeurs vraies et fausses. Ainsi, le type de données booléen n'a que deux littéraux qui sont vrais et faux. Les types booléens sont par défaut à false s'ils ne sont pas initialisés.

Data Types in Java

1 byte = 8 bit

Data Type	Default Value	Default Size	Value Range	Example
boolean	false	1 bit (which is a special type for representing true/false values)	true/false	boolean b=true;
char	'\u0000'	2 byte (16 bit unsigned unicode character)	0 to 65,535	char c='a';
byte	0	1 byte (8 bit Integer data type)	-128 to 127	byte b=10;
short	0	2 byte (16 bit Integer data type)	-32768 to 32767	short s=11;
int	0	4 byte (32 bit Integer data type)	-2147483648 to 2147483647.	int i=10;
long	0L	8 byte (64 bit Integer data type)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long l=100012;
float	0.0f	4 byte (32 bit float data type)	1.40129846432481707e-45 to 3.40282346638528860e+38 (positive or negative).	float f=10.3f;
double	0.0d	8 byte (64 bit float data type)	4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative)	double d=11.123;

Boxing et Unboxing des types primitifs

- En Java, chacun des 8 types primitifs possède son objet. Cet objet se nomme « enveloppe » « Wrapper ».
- Le processus d'encapsulation de la valeur primitive dans le type Wrapper est appelé boxing. D'autre part, obtenir une valeur primitive à partir du type Wrapper s'appelle unboxing
- Le compilateur Java effectue l'autoboxing et l'autounboxing chaque fois que cela est nécessaire automatiquement. Où l'autoboxing est la conversion du type primitif en type référence (objet de la classe Wrapper correspondante), par exemple : int en Integer. Le déballage automatique est la conversion du type de référence (objet de la classe Wrapper) en primitif, par exemple : conversion de Integer en int.

Les Types Wrappers (types de référence) fournis par java pour les types primitifs correspondants sont :

boolean	Boolean
byte	Byte
character	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

L'API Java

Le langage Java possède un nombre impressionnant de classes (objets) dans sa librairie que l'on peut utiliser comme on veut. Il n'est pas nécessaire de les inclure, l'IDE (Eclipse) s'en chargera à notre place. Cela se nomme une API (Application Programming Interface).

Le rôle de toutes ces classes est de nous permettre de minimiser le code que l'on doit inventer.

Ces classes sont regroupées en « Package ».

Voici des liens vers quelques classes de l'API 19 Java

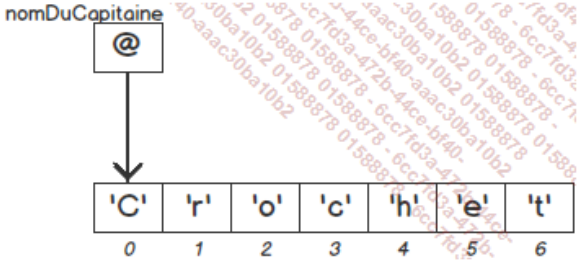
Classe	Lien vers l'API
Object	https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Object.html
System	https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/System.html
String	https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/String.html
Math	https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Math.html
Integer	https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Integer.html
Double	https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Double.html
Character	https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Character.html
Vector	https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Vector.html

Les chaînes de caractères: String

Pour pouvoir stocker des chaînes de caractères, il faut utiliser le type `String` qui représente une suite de zéro à n caractères. Ce type n'est pas un type de base, mais une classe. Cependant, pour faciliter son emploi, il peut être utilisé comme un type de base du langage. Les chaînes de caractères sont invariables, car lors de l'affectation d'une valeur à une variable de type chaîne de caractères, de l'espace est réservé en mémoire pour le stockage de la chaîne. Si par la suite, cette variable reçoit une nouvelle valeur, un nouvel emplacement lui est assigné en mémoire. Heureusement, ce mécanisme est transparent pour nous et la variable fera toujours automatiquement référence à la valeur qui lui a été assignée. Avec ce mécanisme, les chaînes de caractères peuvent avoir une taille variable. L'espace occupé en mémoire est automatiquement ajusté en fonction de la longueur de la chaîne de caractères. Pour affecter une chaîne de caractères à une variable, il faut saisir le contenu de la chaîne entre guillemets comme dans l'exemple ci-dessous.

```
String nomDuCapitaine = "Crochet";
```

La représentation mémoire de cette variable peut être matérialisée ainsi :



De nombreuses méthodes de la classe `String` permettent la manipulation des chaînes de caractères et sont détaillées dans la suite ce chapitre.

Pour les exemples suivants, nous allons travailler avec deux chaînes :

```
String chaine1 = "l'hiver sera pluvieux";
String chaine2 = "l'hiver sera froid";
```

Extraire un caractère particulier

Pour obtenir le caractère présent à une position donnée d'une chaîne de caractères, il faut utiliser la fonction `charAt` en fournissant comme argument l'index du caractère que l'on souhaite obtenir. Le premier caractère est à l'index zéro comme pour un tableau. Cette fonction retourne un caractère (`char`).

```
System.out.println("le 3ème caractère de la chaine 1 est " +
                   chaine1.charAt(2));
```

L'exécution de cette instruction donne le résultat suivant :

```
le 3ème caractère de la chaine 1 est h
```

Obtenir la longueur d'une chaîne

Pour déterminer la longueur d'une chaîne, la fonction `length` de la classe `String` est disponible.

```
System.out.println("la chaine 1 contient " +
                    chaine1.length() +
                   " caractères");
```

L'exécution de cette instruction donne le résultat suivant :

```
la chaine 1 contient 21 caractères
```

Découper une chaîne

La fonction `substring` de la classe `String` retourne une portion d'une chaîne de caractères en fonction de la position de départ et de la position de fin qui lui sont passées comme paramètres. La chaîne obtenue commence par le caractère situé à la position de départ et se termine au caractère précédant la position de fin.

```
System.out.println("un morceau de la chaine 1 : " +
                   chaine1.substring(2,7));
```

L'exécution de cette instruction donne le résultat suivant :

```
un morceau de la chaine 1 : hiver
```

Tester l'égalité de deux chaînes

Lorsqu'on fait une comparaison de deux chaînes, on est tenté d'utiliser le double égal (`==`), comme pour les types valeurs (cf. la section Les opérateurs un peu plus loin dans le chapitre). Cet opérateur fonctionne correctement sur les types de base, mais il ne faut pas perdre de vue que les chaînes de caractères sont des types objet. L'opérateur `==` vérifie l'égalité des variables (donc les valeurs pour les types valeurs, mais les adresses pour les types références).

Il faut donc utiliser les méthodes de la classe `String` pour effectuer des comparaisons de chaînes de caractères. La méthode `equals` effectue une comparaison de la chaîne avec celle qui est passée en paramètre. Elle retourne un booléen égal à `true` si les deux chaînes sont identiques (en valeur) et bien sûr un booléen égal à `false` dans le cas contraire. Cette fonction fait une distinction entre minuscules et majuscules lors de la comparaison. La fonction `equalsIgnoreCase` effectue un traitement identique, mais sans tenir compte de cette distinction.

```
if(chaine1.equals(chaine2))
{
    System.out.println("les deux chaines sont identiques");
}
else
{
    System.out.println("les deux chaines sont différentes");
}
```

- Dans certains cas, l'opérateur == est bien capable de réaliser une comparaison correcte de chaînes de caractères.

```java
String s1="toto";
String s2="toto";
if(s1==s2)
{
  System.out.println("chaines identiques");
}
else  {
  System.out.println("chaines différentes");
}
```

En fait, pour économiser de l'espace en mémoire, Java n'utilise dans ce cas qu'une seule instance de la classe `String` pour les variables `s1` et `s2` car le contenu des deux chaînes est identique.

Les deux variables `s1` et `s2` référencent donc la même zone mémoire et l'opérateur `==` constate donc l'égalité (des adresses).

Si en revanche nous utilisons le code suivant qui demande explicitement la création d'une instance de la classe `String` pour chacune des variables `s1` et `s2`, l'opérateur `==` ne constate bien sûr plus l'égalité des chaînes.

```java
String s1=new String("toto");
String s2=new String("toto");
if(s1==s2)
{
  System.out.println("chaines identiques");
}
else
{
  System.out.println("chaines différentes");
}
```

Comparer deux chaînes

Pour réaliser une comparaison, vous pouvez utiliser la méthode `compareTo` de la classe `String` ou la méthode `compareToIgnoreCase` si vous ne souhaitez pas prendre en compte la casse. Ces deux méthodes attendent en paramètres la chaîne à comparer avec l'instance courante sur laquelle on appelle la méthode. Le résultat de la comparaison est retourné sous forme d'un entier inférieur à zéro si la chaîne est inférieure à celle reçue comme paramètre, égal à zéro si les deux chaînes sont identiques, et supérieur à zéro si la chaîne est supérieure à celle reçue comme paramètre.

```
if(chaine1.compareTo(chaine2)>0)
{
  System.out.println("chaine1 est supérieure a chaine2");
}
else if(chaine1.compareTo(chaine2)<0)
{
  System.out.println("chaine1 est inférieure a chaine2");
}
else
{
  System.out.println("les deux chaines sont identiques");
}
```

Les fonctions `startsWith` et `endsWith` permettent de tester si la chaîne débute par la chaîne reçue en paramètre ou si la chaîne se termine par la chaîne reçue en paramètre. La fonction `endsWith` peut par exemple être utilisée pour tester l'extension d'un nom de fichier.

```
String nom="Code.java";

if(nom.endsWith(".java"))
{
    System.out.println("c\'est un fichier source java");
}
```

Supprimer les espaces

La fonction `trim` permet de supprimer les espaces situés avant le premier caractère significatif et après le dernier caractère significatif d'une chaîne.

```
String eni="          eni         ";
System.out.println("longueur de la chaine : " +
                   eni.length());
System.out.println("longueur de la chaine nettoyée : " +
                   eni.trim().length());
```

Changer la casse

Tout en majuscules :

```
System.out.println(chaine1.toUpperCase());
```

Tout en minuscules :

```
System.out.println(chaine1.toLowerCase());
```

Rechercher dans une chaîne

La méthode `indexOf` de la classe `String` permet la recherche d'une chaîne à l'intérieur d'une autre. Le paramètre correspond à la chaîne recherchée. La fonction retourne un entier indiquant la position à laquelle la chaîne a été trouvée ou `-1` si la chaîne n'a pas été trouvée. Par défaut, la recherche commence au début de la chaîne, sauf si vous utilisez une autre version de la fonction `indexOf` qui, elle, attend deux paramètres, le premier paramètre étant la chaîne recherchée et le deuxième la position de départ de la recherche.

```
String recherche;
int position;
recherche = "e";
position = chaine1.indexOf(recherche);
while (position >= 0)
{
  System.out.println("chaine trouvée à la position " +
                     position);
  position = chaine1.indexOf(recherche,position+1);
}
System.out.println("fin de la recherche");
```

Nous obtenons à l'affichage :

```
chaîne trouvée à la position 5
chaîne trouvée à la position 9
chaîne trouvée à la position 18
fin de la recherche
```

Remplacer une partie d'une chaîne

Il est parfois souhaitable de pouvoir rechercher la présence d'une chaîne à l'intérieur d'une autre, comme dans l'exemple précédent, mais également remplacer les portions de chaîne trouvées. La fonction `replace` permet de spécifier une chaîne de substitution pour la chaîne recherchée. Elle attend deux paramètres :

- la chaîne recherchée,
- la chaîne de remplacement.

```
String chaine3;
chaine3= chaine1.replace("hiver", "été");
System.out.println(chaine3);
```

Nous obtenons à l'affichage :

```
l'été sera pluvieux
```

Formater une chaîne

La méthode `format` de la classe `String` permet d'éviter de longues et fastidieuses opérations de conversion et de concaténation. C'est une méthode de classe et non une méthode d'instance. Le premier paramètre attendu par cette fonction est une chaîne de caractères spécifiant sous quelle forme on souhaite obtenir le résultat. Cette chaîne contient un ou plusieurs motifs de formatage représentés par le caractère `%`, suivis d'un caractère spécifique indiquant sous quelle forme doit être présentée l'information. Il doit ensuite y avoir autant de paramètres qu'il y a de motifs de formatage. La chaîne renvoyée est construite par le remplacement de chacun des motifs de formatage par la valeur du paramètre correspondant, le remplacement se faisant dans l'ordre d'apparition des motifs. Le tableau suivant présente les principaux motifs de formatage disponibles.

Motif	Description
`%b`	Insertion d'un booléen
`%s`	Insertion d'une chaîne de caractères
`%d`	Insertion d'un nombre entier
`%o`	Insertion d'un entier affiché en octal
`%x`	Insertion d'un entier affiché en hexadécimal
`%f`	Insertion d'un nombre décimal
`%e`	Insertion d'un nombre décimal affiché au format scientifique
`%n`	Insertion d'un saut de ligne. Équivalent à la séquence `\n` mais ne fonctionne que dans les méthodes de formatage comme `format`.

Voici un exemple de code utilisant cette fonction :

```
boolean b=true;
String s="chaine";
int i=56;
double d=5.5;
System.out.println(String.format("boolean : %b %n" +
                "chaine de caractères : %s %n" +
                "entier : %d %n" +
                "entier en hexadécimal : %x %n" +
                "entier en octal : %o %n" +
                "décimal : %f %n" +
                "décimal précis au dixième : %.1f %n" +
                "décimal au format scientifique : %e %n",
                b,s,i,i,i,d,d,d));
```

Remarquez la séquence `%.1f` pour l'affichage en décimal. Cette instruction permet d'indiquer le nombre de chiffres significatifs souhaités après la virgule.

L'exécution de ce code donne le résultat suivant :

```
boolean : true
chaine de caractères : chaine
entier : 56
entier en hexadécimal : 38
entier en octal : 70
décimal : 5,500000
décimal précis au dixième : 5,5
décimal au format scientifique : 5.500000e+00
```

Les opérateurs

Les opérateurs

- Les opérateurs sont des mots-clés du langage permettant l'exécution d'opérations sur le contenu de certains éléments, en général des variables, des constantes, des valeurs littérales, ou des retours de fonctions. La combinaison d'un ou de plusieurs opérateurs et d'éléments sur lesquels les opérateurs vont s'appuyer se nomme une expression. Ces expressions sont évaluées au moment de l'exécution en fonction des opérateurs et des valeurs qui sont associées.
- Les opérateurs peuvent être répartis en sept catégories

1. Les opérateurs unaires

Opérateur	Action
`-`	Valeur négative
`~`	Complément à 1 (inversion des bits)
`++`	Incrémentation
`--`	Décrémentation
`!`	Négation

2. L’opérateur d’affectation

Le seul opérateur disponible dans cette catégorie est l’opérateur =. Il permet d’affecter une valeur à une variable. Le même opérateur est utilisé, quel que soit le type de la variable (numérique, chaîne de caractères...).

Cet opérateur peut être combiné avec un opérateur arithmétique, logique ou binaire.

La syntaxe suivante :

```
x+=2;
```

est équivalente à :

```
x=x+2;
```

3. Les opérateurs arithmétiques

Les opérateurs arithmétiques permettent d'effectuer des calculs sur le contenu des variables. Le type du résultat de l'opération correspond au type le plus grand des deux opérandes.

Opérateur	Opération réalisée	Exemple	Résultat
`+`	Addition pour des valeurs numériques ou concaténation pour des chaînes	`6+4`	`10`
`-`	Soustraction	`12-06`	`6`
`*`	Multiplication	`3*4`	`12`
`/`	Division	`25/03`	`8` ou `8.333...`
`%`	Modulo (reste de la division entière)	`25%3`	`1`

- Il est à noter que pour la division, il y a une subtilité :
  - Si les deux opérandes sont des numériques entiers, la division opérée est une division entière. Ainsi, le résultat de l’opération 25/3 sera 8.
  - En revanche, si au moins un des deux opérandes est un numérique décimal, alors le résultat de la même opération sera 8.333...

4. Les opérateurs bit à bit

On ne présentera pas cet opérateur dans le cours.

5. Les opérateurs de comparaison

Les opérateurs de comparaison sont utilisés dans les structures de contrôle d'une application. Ils renvoient une valeur de type `boolean` en fonction du résultat de la comparaison effectuée. Cette valeur sera ensuite utilisée par la structure de contrôle.

Opérateur	Test réalisé	Exemple	Résultat
`==`	Egalité	`2 == 5`	`false`
`!=`	Inégalité	`2 != 5`	`true`
`<`	Infériorité	`2 < 5`	`true`
`>`	Supériorité	`2 > 5`	`false`
`<=`	Infériorité ou égalité	`2 <= 5`	`true`
`>=`	Supériorité ou égalité	`2 >= 5`	`false`
`instanceof`	Comparaison du type de la variable avec le type indiqué	`O1 instanceof Client`	`true` si la variable `O1` référence un objet créé à partir de la classe client ou d'une sous-classe

6. L'opérateur de concaténation

L'opérateur +, déjà utilisé pour l'addition, est également utilisé pour la concaténation de chaînes de caractères. Le fonctionnement de l'opérateur est déterminé par le type des opérandes. Si un des opérandes est du type `String`, alors l'opérateur + effectue une concaténation avec éventuellement une conversion implicite de l'autre opérande en chaîne de caractères.

Le petit inconvénient de l'opérateur `+` est qu'il ne brille pas par sa rapidité pour les concaténations. En fait, ce n'est pas réellement l'opérateur qui est en cause, mais la technique utilisée par Java pour gérer les chaînes de caractères (elles ne peuvent pas être modifiées après création). Si vous avez de nombreuses concaténations à exécuter sur une chaîne, il est préférable d'utiliser la classe `StringBuilder` (ou `StringBuffer` uniquement si vous travaillez en environnement multi-thread).

Exemple

```
long duree;
String lievre;
String tortue="";
long debut, fin;
debut = System.currentTimeMillis();
for (int i = 0; i <= 10000; i++)
{
    tortue = tortue + " " + i;
}
fin = System.currentTimeMillis();
duree = fin-debut;
System.out.println("durée pour la tortue : " + duree + "ms");
debut = System.currentTimeMillis();
StringBuilder sb = new StringBuilder();
for (int i = 0; i <= 10000; i++)
{
    sb.append(" ");
    sb.append(i);
}
lievre = sb.toString();
fin = System.currentTimeMillis();
duree = fin-debut;
System.out.println("durée pour le lièvre : " + duree + "ms");
if (lievre.equals(tortue))
{
    System.out.println("les deux chaînes sont identiques");
}
```

Résultat de la course :

```
durée pour la tortue : 249ms
durée pour le lièvre : 10ms
les deux chaînes sont identiques
```

7. Les opérateurs logiques

Les opérateurs logiques permettent de combiner les expressions dans des structures conditionnelles ou des structures de boucle.

Opérateur	Opération	Exemple	Résultat
`&`	Et logique	`test1 & test2`	`true` si test1 et test2 valent `true`
`|`	Ou logique	`test1 | test2`	`true` si test1 ou test2 vaut `true`
`^`	Ou exclusif	`test1 ^ test2`	`true` si test1 ou test2 vaut `true` mais `false` si les deux valent `true` simultanément
`!`	Négation	`!test1`	Inverse le résultat du test
`&&`	Et logique	`test1 && test2`	Comme le `&` mais test2 ne sera évalué que si test1 vaut `true`
`||`	Ou logique	`test1 || test2`	Comme le `|` mais test2 ne sera évalué que si test1 vaut `false`

Il faudra être prudent avec les opérateurs `&&` et `||` car l'expression que vous testerez en second (test2 dans notre cas) pourra parfois ne pas être exécutée. Si cette deuxième expression modifie une variable, celle-ci ne sera modifiée que dans les cas suivants :

- Si test1 vaut `true` dans le cas du `&&`.
- Si test1 vaut `false` dans le cas du `||`.

Ordre d’évaluation des opérateurs

Lorsque plusieurs opérateurs sont combinés dans une expression, ils sont évalués dans un ordre bien précis. Les incrémentations et décrémentations préfixées sont exécutées en premier, puis les opérations arithmétiques, les opérations de comparaison, les opérateurs logiques et enfin les affectations.

Les opérateurs arithmétiques ont entre eux également un ordre d’évaluation dans une expression. L’ordre d’évaluation est le suivant :

- Négation (-)
- Multiplication et division (`*`, `/`)
- Modulo (`%`)
- Addition et soustraction (`+`, `–`), concaténation de chaînes (`+`)

Si un ordre d’évaluation différent est nécessaire dans votre expression, il faut placer les portions à évaluer en priorité entre parenthèses comme dans l’expression suivante :

```
x= (z * 4) ^ (y * (a + 2));
```

Vous pouvez utiliser autant de niveaux de parenthèses que vous le souhaitez dans une expression. Il importe cependant que l’expression contienne autant de parenthèses fermantes que de parenthèses ouvrantes sinon le compilateur générera une erreur.

Les structures de contrôle

Structure de décision

```
if (condition)instruction;
```

```
if (condition)
{
     instruction 1;
     ...
     instruction n;
}
```

```
if (condition)
{
     instruction 1;
     ...
     instruction n;
}
else
{
     instruction 1;
     ...
     instruction n;
}
```

```
if (condition1)
{
     instruction 1
     ...
     instruction n
}
else if (condition 2)
{
     instruction 1
     ...
     instruction n
}
else
{
     instruction 1
     ...
     instruction n
}
```

Structure ternaire

Cette structure particulière correspond à une instruction `if...else`. Elle n'est utilisable que pour l'affectation d'une valeur. C'est une structure concise remplaçant avantageusement l'instruction `if...else`.

```
String message = condition ? "si true" : "si false";
```

Le `?` permet de déclencher l'évaluation de la condition. Si la condition vaut `true` alors c'est la valeur fournie après le `?` qui est retournée, sinon c'est la valeur fournie après les `:` qui est retournée.

L'instruction équivalente à un `if...else` est la suivante :

```
if(condition)
   message="si true";
else
   message="si false";
```

Instructions switch en Java – version historique

- Au lieu d'écrire plusieurs instructions if...else, vous pouvez utiliser l'instruction switch.
- L'instruction switch sélectionne l'un des nombreux blocs de code à exécuter :

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

```
int day = 4;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  case 5:
    System.out.println("Friday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
  case 7:
    System.out.println("Sunday");
    break;
}
// Outputs "Thursday" (day 4)
```

Structure switch – nouvelle génération

La structure `switch` historique souffrait de quelques limitations :

- Obligation de répéter le mot-clé `case` pour chaque valeur à évaluer.
- Obligation d'utiliser le mot clé `break` pour sortir du `switch`.
- Impossibilité d'initialiser simplement une variable à partir de cette structure.

La nouvelle version du `switch` répond à ces problématiques. Voici la nouvelle syntaxe de la structure (l'ancienne syntaxe est bien sûr toujours compatible) :

```
[TypeRetour variableRetour=]
switch (expression)
{
    case valeur1, valeur2 -> instruction
    case valeur3 -> {
              TypeRetour valeurRetour;
              instruction 1
              ...
              instruction n
              [yield valeurRetour]
            }
    default -> instruction
}
```

Exemple: Switch nouvelle génération

```java
public class SwitchExpressionWithYield {
    public static void main(String[] args) {
        String result = switchExample(2);
        System.out.println(result);
    }

    static String switchExample(int value) {
        return switch (value) {
            case 1 -> "One";
            case 2 -> {
                System.out.println("Processing value 2");
                yield "Two";
            }
            case 3 -> "Three";
            default -> "Other";
        };
    }
}
```

Les structures de boucle

Structure de boucle - While

```
while (condition)
{
    instruction 1
    ...
    instruction n
}
```

```
int i=0;
while (i<=10)//Pas de ; ici. Sinon le bloc sous-jacent n'est pas
             //lié à la boucle
{
    System.out.println(i);
    i++;
}
```

Structure de boucle – do…while

```
do
{
    System.out.println(i);
    i++;
}
while(i<=10);
```

```
do
{
    instruction 1
    ...
    instruction n
}
while (condition);
```

Structure de boucle - for

```
for(initialisation;condition;instruction d'itération)
{
    instruction 1
    ...
    instruction n
}
```

```
int multiplicateur;
for(multiplicateur=1;multiplicateur<=10;multiplicateur++)
{
   for (int table = 1; table <= 10; table++)
   {
      System.out.print(table*multiplicateur + "\t");
   }
   System.out.println();
}
```

Nous obtenons le résultat suivant :

```
1    2     3     4     5     6     7     8     9     10
2    4     6     8     10    12    14    16    18    20
3    6     9     12    15    18    21    24    27    30
4    8     12    16    20    24    28    32    36    40
5    10    15    20    25    30    35    40    45    50
6    12    18    24    30    36    42    48    54    60
7    14    21    28    35    42    49    56    63    70
8    16    24    32    40    48    56    64    72    80
9    18    27    36    45    54    63    72    81    90
10   20    30    40    50    60    70    80    90    100
```

Interruption d’une boucle

break

- Vous avez déjà vu l'instruction break utilisée dans un chapitre précédent de ce didacticiel. Il était utilisé pour « sortir » d'une instruction switch.
- L'instruction break peut également être utilisée pour sortir d'une boucle.
- Cet exemple arrête la boucle lorsque i est égal à 4 :

```
public class Main {
  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      if (i == 4) {
        break;
      }
      System.out.println(i);
    }
  }
}

affiche: 0 1 2 3
```

continue

- L'instruction continue interrompt une itération (dans la boucle), si une condition spécifiée se produit, et continue avec l'itération suivante dans la boucle.
- Cet exemple ignore la valeur 4 :

```
public class Main {
  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      if (i == 4) {
        continue;
      }
      System.out.println(i);
    }
  }
}

//Affiche: 0 1 2 3 5 6 7 8 9 sans le 4
```

Saisie d’utilisateur
Scanner

Entrée d'utilisateur

Les types de données : Le Scanner pour la gestion des flux standards

- La classe `java.util.Scanner` permet de lire les flux standards. Elle dispose de méthode pour lire ces flux de données formatées.

- Un objet `Scanner` se construit à partir d'un flux d'octets ou de caractères (comme par exemple, l'entrée standard : `System.in`) :

```
Scanner scanner = new Scanner(System.in);
```

Les types de données : Le Scanner pour la gestion des flux standards (suite)

- Exemple :

```
Scanner scanner = new Scanner(System.in);
System.out.print("Nom et prénom ? : ");
String saisie = scanner.nextLine();
System.out.println("Bonjour " + saisie);
```

Les types de données : Le Scanner pour la gestion des flux standards (suite)

- Quelques méthodes de lecture :
  - `byte nextByte(), short nextShort(), int nextInt(), long nextLong(), float nextFloat(), double nextDouble(), BigInteger nextBigInteger(), BigDecimal nextBigDecimal()`
  - Ces méthodes renvoient la prochaine valeur du type indiqué
  - `String nextLine()`
  - Cette méthode renvoie tous les caractères du flux jusqu'à une marque de fin de ligne

Les types de données : Le Scanner pour la gestion des flux standards (suite)

- Gestion des exceptions
  - Les méthodes de lecture du flux sont susceptibles de renvoyer des exceptions dans les cas suivants :
    - La saisie ne correspond pas au type de données attendu (Une chaîne saisie sur un `nextInt()` par exemple)
      - `java.util.InputMismatchException`
    - Il n'y a pas d'éléments dans le flux
      - `java.util.NoSuchElementException`
    - Le scanner est fermé
      - `java.util.IllegalStateException`

Les types de données : Le Scanner pour la gestion des flux standards (suite)

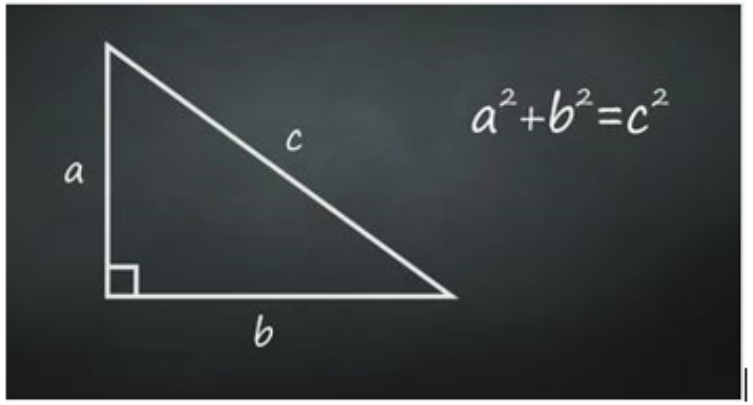
- Tester la présence des données
  - La classe `Scanner` possède un ensemble de méthodes permettant de tester la présence et le type de la prochaine donnée (sans la lire !)
  - `boolean hasNext()`
    - Y a-t-il une donnée disponible pour la lecture ?
  - `boolean hasNextLine()`
    - Y a-t-il une ligne disponible pour la lecture ?
  - `boolean hasNextByte(), boolean hasNextShort(), boolean hasNextInt(), boolean hasNextLong(), boolean hasNextFloat(), boolean hasNextDouble(), boolean hasNextBigInteger(), boolean hasNextBigDecimal()`

Exercices

Pratique 1

Écrire un programme Java qui implémente le théorème de Pythagore.

Le théorème de Pythagore est un théorème de géométrie euclidienne qui met en relation les longueurs des côtés dans un triangle rectangle. Il s'énonce fréquemment sous la forme suivante : si un triangle est rectangle, le carré de la longueur de l'hypoténuse (ou côté opposé à l'angle droit) est égal à la somme des carrés des longueurs des deux autres côtés.



Pratique 2

Écrire un programme Java qui simule un avertisseur de contrôle de vitesse des voitures.

- Si un automobiliste respecte la limite de vitesse permise, le programme lui affiche le message suivant :
  - Merci! Vous respectez la limite de vitesse!

- Si un automobiliste ne respecte pas la limite de vitesse permise, le programme lui affiche le message suivant :
  - Ralentissez! Vous ne respectez pas la limite de vitesse!

- Si un automobiliste roule à une vitesse en dessous de 30 de la limite de vitesse permise, le programme lui affiche le message suivant :
  - Bougez-vous! Vous allez être en retard pour le souper!

Pratique 3

Écrire un programme Java qui simule le processus suivant :

Soit un magasin de chocolat qui offre les rabais suivants :

- Si le client achète 5 boites de chocolat et plus, il aura un rabais de 10%
- Si le client achète 50 boites de chocolat et plus, il aura un rabais de 15%

Voici un exemple de test :

```
**************************************************************************|
Veuillez entrer le nombre de boites de chocolat que vous voulez acheter:   |
2
Vous n'avez pas droit au rabais!                                           |
**************************************************************************|
Veuillez entrer le nombre de boites de chocolat que vous voulez acheter:   |
8
Vous avez droit à un rabais de  10%                          |
**************************************************************************|
Veuillez entrer le nombre de boites de chocolat que vous voulez acheter:   |
50
Vous avez droit à un rabais de 15%                            |
**************************************************************************|
```

Générer trois nombres aléatoires compris entre 0 et 1000, puis vérifier si vous avez deux nombres pairs suivis par un nombre impair. Si ce n'est pas le cas, recommencer jusqu'à ce que vous ayez la combinaison pair, pair, impair. Afficher ensuite le nombre d'essais nécessaires pour obtenir cette combinaison.

Indice : la classe `Random` propose un ensemble de méthodes permettant d'obtenir un nombre aléatoire. Concentrez-vous sur la méthode suivante en lisant la javadoc :

```
public int nextInt(int bound)
```

Pour utiliser cette méthode, il est nécessaire d'avoir un objet de type `Random` :

```
Random rd = new Random();
rd.nextInt(…);
```

Générer un nombre aléatoire compris entre 0 et 1000. Demander ensuite à l'utilisateur de deviner le nombre choisi par l'ordinateur. Il doit saisir un nombre compris entre 0 et 1000 lui aussi. Comparer le nombre saisi avec celui choisi par l'ordinateur et afficher sur la console « c'est plus » ou « c'est moins » selon le cas. Recommencer jusqu'à ce que l'utilisateur trouve le bon nombre. Afficher alors le nombre d'essais nécessaires pour trouver la bonne réponse.

Références

Java Tutorial (w3schools.com)

Java - Les fondamentaux du langage (avec exercices pratiques et corrigés)" de Thierry RICHARD