Types et instructions de base Java

Par Aïcha El Golli

aicha.elgolli@essai.ucar.tn

Commentaires

- Trois formes de commentaires :

```
/**
 *  Somme : Calcule une somme
 *
 *  @author  Marc Duchemin
 *  @version 2.3  08.07.2014
 */
public class Somme {
  /*-------------------------------------------------+
  |  Programme principal                             |
  +-------------------------------------------------*/
  public static void main(String[] params) {

    //--- Initialisation des variables
    int  a     = 1;
    int  b     = 2;
    int  total;       // Totalisateur

    total = a + b;

    //--- Affichage du résultat de l'addition
    System.out.println("Résultat = " + total);
  }
}
```

// …Texte…

- ✓ Commence dès // et se termine à la fin de la ligne
- ✓ Sur une seule ligne
- ✓ A utiliser de préférence pour les commentaires généraux

/* …Texte… */

- ✓ Le texte entre /* et */ est ignoré par le compilateur
- ✓ Peuvent s'étendre sur plusieurs lignes
- ✓ Ne peuvent pas être imbriqués
- ✓ Peuvent être utiles pour inactiver (temporairement) une zone de code

/** …Texte… */

- ✓ Commentaire de documentation (comme /* … */ mais avec fonction spéciale)
- ✓ Interprétés par l'utilitaire *javadoc* pour créer une documentation au format HTML
- ✓ Peuvent contenir des balises de documentation (@author, @param, …)
- ✓ Peuvent contenir des balises HTML *(Tags)*

Commentaires

- javadoc : outil qui analyse le code Java, et produit **automatiquement** une documentation HTML avec la liste des classes, leurs attributs, méthodes… avec leurs éventuels "commentaires de documentation" respectifs
- Commenter le plus possible et judicieusement
- Chaque déclaration (variable, mais aussi méthode, classe)
- Commenter clairement (utiliser au mieux les 3 possibilités)

Instructions, blocs et blancs

- Les instructions Java se terminent par un ;
- Les blocs sont délimités par :
- { pour le début de bloc
- } pour la fin du bloc
- Un bloc permet de définir un regroupement d'instructions. La définition d'une classe ou d'une méthode se fait dans un bloc.
- Les espaces, tabulations, sauts de ligne sont autorisés. Cela permet de présenter un code plus lisible.

Point d'entrée d'un programme Java

- Pour pouvoir faire un programme exécutable il faut toujours une classe qui contienne une méthode particulière, la méthode « main »
  - c'est le point d'entrée dans le programme : le microprocesseur sait qu'il va commencer à exécuter les instructions à partir de cet endroit

```
public static void main(String []args)
{
…
…
}
```

IDENTIFICATEURS (1)

- On a besoin de nommer les classes, les variables, les constantes, etc. ; on parle d'identificateur.
- Règles pour les identificateurs :
  - Doivent commencer par une lettre ou un souligné (ou un caractère monétaire $)
  - Suivi (éventuellement) d'un nombre quelconque de lettres, chiffres ou soulignés (ou de caractères monétaires)
  - Distinction entre les majuscules et les minuscules (éviter de créer des identificateurs qui ne se distinguent que par la casse)

IDENTIFICATEURS (1)

```
maVariable1
ma_variable1
_maVariable1
$maVariable1
maVariable1$
_uneVariableCachée
```

possibilité d'utiliser des caractères accentués mais pas recommandé

```
1Variable
ma Variable1
maVariable1.
```

ne doit pas être un mot réservé du langage (noms prédéfinis)

- `abstract, assert, boolean, break, byte, case, catch, char, class, continue, default, do, double, else, extends, final, finally, float, for, foreach, if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, super, switch, synchronized, this, throw, throws, transient, try, var, void, volatile, while`

IDENTIFICATEURS (2)

Conventions sur les identificateurs :

- La première lettre est majuscule pour les **classes** et les interfaces

  exemples : MaClasse, UneJolieFenetre

- La première lettre est minuscule pour les **méthodes**, les **attributs** et les **variables**

  exemples : somme, i, nom

Si plusieurs mots sont accolés, mettre une majuscule à chacun des mots sauf le premier.

exemple : uneVariableEntiere, uneFenetre, setLongueur

- Les **constantes** sont entièrement en majuscules

  exemple : LONGUEUR_MAX

Les variables et types de données en java

- Une **variable** définit une **case mémoire nommée et typée**.
- Avant de pouvoir être utilisée, une variable doit préalablement être **déclarée** (définition de l'identificateur et du type).
- Syntaxe : ***Type Identificateur ;***
- 2 grands groupes de types de données :
- *types primitifs* (entiers, flottants, …)
- *objets ou instances de classe* (c'est-à-dire le nom d'une classe ou d'un tableau)
- Java manipule différemment les valeurs des types primitifs et les objets: les variables contiennent
- *des valeurs de types primitifs*
- *ou des références aux objets (*null : valeur d'une "référence vers rien" (pour tous types de références) et indique qu'une variable de type non primitif ne référence rien)

Les types de données prédéfinis
types primitifs

Type	Contient	Taille	Valeurs
boolean	Booléen	[1 bit [1)]]	`true, false`
char	Caractère (entier non-signé)	16 bits	\u0000..\uFFFF
byte	Entier signé	8 bits	-128..127
short	Entier signé	16 bits	$-2^{15}..\ 2^{15}-1$
int	Entier signé	32 bits	$-2^{31}..\ 2^{31}-1$
long	Entier signé	64 bits	$-2^{63}..\ 2^{63}-1$
float	Nombre en virgule flottante	32 bits	$\pm 3.4*10^{38}$
double	Nombre en virgule flottante	64 bits	$\pm 1.7*10^{308}$

- La taille nécessaire au stockage de ces types est indépendante de la machine.
  - Avantage : portabilité
  - Inconvénient : "conversions" coûteuses

Les types de données prédéfinis
enveloppeurs (wrappers)

Chacun des types primitifs peut être "enveloppé" dans un objet provenant d'une classe prévue à cet effet et appelée *Wrapper* (mot anglais signifiant *enveloppeur*). Les enveloppeurs sont donc des objets représentant un type primitif.

Enveloppeur	Type primitif
Character	char
Byte	byte
Short	short
Integer	int
Long	long
Float	float
Double	double
Boolean	boolean

Avantages :
Les Wrappers peuvent être utilisés comme n'importe quel objet, ils ont donc leurs propres méthodes.

Inconvénients :
•L'objet enveloppant utilise plus d'espace mémoire que le type primitif. Par exemple, un int prends 4 octets en mémoire mais un Integer utilisera 32 octets sur une machine virtuelle en 64 bits (20 octets en 32 bits).
•L'objet enveloppant est immuable, c'est à dire qu'il ne peut pas être modifié, toute modification de sa valeur nécessite de créer un nouvel objet et de détruire l'ancien, ce qui augmente le temps de calcul.

Types primitifs

- **Booléen (`boolean`)**
  - Ne peuvent prendre que deux valeurs : *Vrai* ou *Faux*
  - Ne peuvent pas être interprétés comme des valeurs numériques [0, 1]
  - Valeurs littérales (valeurs qui figurent directement dans le code)
    - ✓ `true` *Vrai*
    - ✓ `false` *Faux*

- **Caractère (`char`)**
  - Caractères *Unicode* (codage normalisé sur 16 bits)
  - Site de référence de la norme : www.unicode.org
  - Peuvent être traités comme des entiers (non-signés !)
  - Valeurs littérales
    - ✓ Entre apostrophes : `'A'` (attention : `"A"` n'est pas de type `char`)
    - ✓ Séquence d'échappement pour certains caractères spéciaux (voir tabelle page suivante)

Types primitifs

- Séquences d'échappement (`char`)

Code	Signification
`\b`	Retour en arrière *(Backspace)*
`\t`	Tabulateur horizontal
`\n`	Saut de ligne *(Line-feed)*
`\f`	Saut de page *(Form-feed)*
`\r`	Retour de chariot *(Carriage-Return)*
`\"`	Guillemet
`\'`	Apostrophe
`\\`	Barre oblique arrière *(Backslash)*
`\`*xxx*	Caractère *Latin-1* (xxx : valeur octale `000`..`377`)
`\u`*xxxx*	Caractère *Unicode* (xxxx : valeur hexadécimale `0000`..`FFFF`)

Types primitifs : les entiers

- **Entiers signés (`byte`, `short`, `int`, `long`)**
  - Valeurs littérales (`int` par défaut) : notation habituelle
  - Suffixe `l` ou `L` pour type `long` (`L` est préférable)
  - Préfixe `0` (zéro) pour valeur octale (base 8)
  - Préfixe `0b` ou `0B` pour valeur binaire (base 2) *Java 7*
  - Préfixe `0x` ou `0X` pour valeur hexadécimale (base 16)
  - Exemples :

```
0         // valeur de type int
123       // valeur de type int
-56       // valeur de type int
0377      // 377 [octal] = 255 [décimal]
433L      // valeur de type long
0b1011    // valeur binaire de type int
0xff      // valeur hexadécimale de type int
0xA0B3L   // valeur hexadécimale de type long
```

Types primitifs : les entiers

- Opérations sur les entiers
  - opérateurs arithmétiques +, -, *
  - / :division entière si les 2 arguments sont des entiers
  - % : reste de la division entière
    - exemples :
      - 15 / 4 donne 3
      - 15 % 2 donne 1
  - les opérateurs d'incrémentation **++** et de décrémentation et **--**
    - ajoute ou retranche 1 à une variable
      - int n = 12;
      - n ++; //Maintenant n vaut 13
    - 8++; est une instruction illégale
    - peut s'utiliser de manière suffixée : ++n. La différence avec la version préfixée se voit quand on les utilise dans les expressions. En version suffixée la (dé/inc)rémentation s'effectue en premier

```
int m=7; int n=7;
int a=2 * ++m; //a vaut 16, m vaut 8
int b=2 * n++; //b vaut 14, n vaut 8
```

- conversion automatique seulement vers les types entiers plus grands(int --> long) et vers les types flottants

Passage d'un type entier à un autre

- transfert de la valeur d'une variable entière vers une autre variable entière dépend du type des variables
  - 1er cas : la taille de la variable affectée est supérieure ou égale à la taille de la variable transférée
    - conversion de type implicite autorisée (il n'y a pas de perte de valeur)

```
byte c = 127;
short s = c;
int i = s;
long l = i;
```

toutes les variables ont la valeur 127

  - 2ème cas : la taille de la variable affectée est inférieure à la taille de la variable transférée

```
long l = 127;
int i = l;
short s = i;
byte b = s;
```

```
Error: incompatible types: possible lossy conversion from long to int
Error: incompatible types: possible lossy conversion from int to short
Error: incompatible types: possible lossy conversion from short to byte
```

    - le changement de type doit être explicité à l'aide d'un opérateur de transtypage (cast)

```
long l = 127;
int i = (int) l;
short s = (short) i;
byte b = (byte) s;
```

le transtypage peut occasionner une perte de données

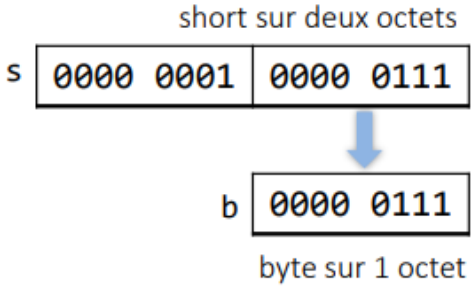
```
short s = 263;
byte b = (byte) s;
System.out.println(b);  --->  7
```

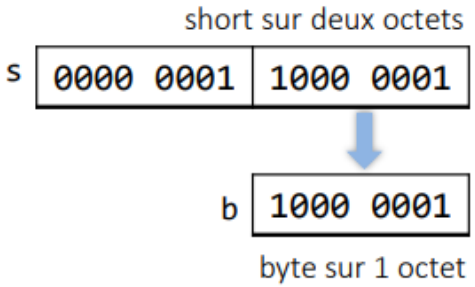
Passage d'un type entier à un autre

- transtypage avec perte de données

```
short s = 263;
byte b = (byte) s;
System.out.println(b);  --->  7
```

```
short s = 385;
byte b = (byte) s;
System.out.println(b);  --->  -127
```





bit de signe								
0	1	1	1	1	1	1	1	= 127
0				...				= ...
0	0	0	0	0	0	1	0	= 2
0	0	0	0	0	0	0	1	= 1
0	0	0	0	0	0	0	0	= 0
1	1	1	1	1	1	1	1	= −1
1	1	1	1	1	1	1	0	= −2
1				...				= ...
1	0	0	0	0	0	0	1	= −127
1	0	0	0	0	0	0	0	= −128

Représentation en complément à deux sur 8 bits.

https://fr.wikipedia.org/wiki/Compl%C3%A9ment_%C3%A0_deux

Types primitifs : les réels

- **Nombres en virgule flottante (`float`, `double`)**
  - Norme IEEE 754-1985
  - Précision
    - ✓ `float` : env. 6 chiffres significatifs
    - ✓ `double` : env. 15 chiffres significatifs
  - Valeurs littérales (`double` par défaut) : notation habituelle (*n.m*)
  - Suffixe **`f`** ou **`F`** pour type `float`
  - Suffixe **`d`** ou **`D`** pour type `double` (rarement nécessaire)
  - Notation exponentielle avec **`e`** ou **`E`** suivi de l'exposant
  - Valeurs spéciales : *infini, -infini, zéro négatif, NaN (Not a Number)*

Attention : Les nombres en virgule flottante sont des **approximations** de nombres réels (tous les nombres réels ne peuvent pas être représentés de manière exacte).

Il faut en tenir compte dans les comparaisons et notamment dans les tests d'égalité (prendre en compte un *"epsilon"*).

Types primitifs : les réels

- Nombres en virgule flottante (**`float`**, **`double`**)

```
123.45      // valeur de type double (par défaut)
17d         // valeur de type double
-4.032F     // valeur de type float
6.02e23     //  6.02  x 10^23 de type double
-5.076E-2f  // -5.076 x 10^-2 de type float
```

- Le caractère '_' peut être inséré dans les littéraux numériques *Java 7*

```
long  creditCardNr  = 1234_5678_9012_3456L;
int   tenMillions   = 10_000_000;
int   bytePattern   = 0b11000011_00001111_11110000_00110011;
float pi            = 3.14_15_93f;
```

- Cas particuliers (problèmes numériques) :

```
1.2/0.0    // Infini                Double.POSITIVE_INFINITY
-5.1/0.0   // Moins l'infini        Double.NEGATIVE_INFINITY
0.0/0.0    // Not a Number (NaN)    Double.NaN
```

Types primitifs : les réels

- Les opérateurs
  - opérateurs classiques +, -, *, /
  - attention pour la division :
    - 15 / 4 donne 3 ***division entière***
    - 15 % 2 donne 1
    - 11.0 / 4 donne 2.75
      (si l'un des termes de la division est un réel, la division retournera un réel).
  - puissance : utilisation de la méthode pow de la classe Math.
    - double y = Math.pow(x, a) équivalent à x^a, x et a étant de type double
- conversion automatique : seulement float --> double
- la conversion "manuelle" en entier tronque la partie décimale :

```
float x=-2.8f;
int i = (int)x; // => i==-2
```

Types de primitifs : les booléens

- ne peut PAS être converti en entier
- Les opérateurs logiques de comparaisons
  - Egalité : opérateur **==**
  - Différence : opérateur **!=**
  - supérieur et inférieur strictement à : opérateurs **>** et **<**
  - supérieur et inférieur ou égal : opérateurs **>=** et **<=**

Types primitifs : les booléens

- Les autres opérateurs logiques
  - et logique : **&&**
  - ou logique : ||
  - non logique : **!**
  - Exemples : si a et b sont 2 variables booléennes

```
boolean a,b, c;
a= true;
b= false;
c= (a && b); // c vaut false
c= (a || b); // c vaut true
c= !(a && b); // c vaut true
c=!a; // c vaut false
```

Types primitifs : les caractères

- Notation
  - **char** a,b,c; // a,b et c sont des variables du type char
  - a='a'; // a contient la lettre 'a'
  - b= '\u0022' //utilisation de la notation unicode b contient le //caractère *guillemet* : "
  - c=97; // x contient le caractère de rang 97 : ‘a‘ utilisation du code //numérique du caractère

- test du type : Character.isLetter(c), Character.isDigit(c), … (où c est une variable de type char)
- convertible automatiquement en int ou long (et manuellement en byte ou short)
- inversement, (char)val est le caractère dont le code Unicode est l'entier val

Types primitifs exemple et remarque

```
int x = 0, y = 0;
float z = 3.1415F;
double  w = 3.1415;
long t = 99L;
boolean test = true;
char c = 'a';
```

- Remarque importante :
  - Java exige que toutes les variables soient définies et initialisées. Le compilateur sait déterminer si une variable est susceptible d'être utilisée avant initialisation et produit une erreur de compilation.

Forcer un type en Java

Java langage fortement typé

- le type de donnée est associé au nom de la variable, plutôt qu'à sa valeur. (Avant de pouvoir être utilisée une variable doit être déclarée en associant un type à son identificateur).
- la compilation ou l'exécution peuvent détecter des erreurs de typage

- Dans certains cas, nécessaire de forcer le compilateur à considérer une expression comme étant d'un type qui n'est pas son type réel ou déclaré
  - *On utilise le* cast ou transtypage*:*

(type-forcé) expression

- **Exemple**

int i = 64;
char c = (char) i;

Casts entre types primitifs

- Un *cast* entre types primitifs peut occasionner une perte de données
  - *Par exemple, la conversion d'un* ***int*** *vers un* ***short*** *peut donner un nombre complètement différent du nombre de départ.*

```
int i = 32768;
short s = (short) i;
System.out.println(s);          // -32768;
```

- Un *cast* peut provoquer une simple perte de précision
  - *Par exemple, la conversion d'un* ***long*** *vers un* ***float*** *peut faire perdre des chiffres significatifs mais pas l'ordre de grandeur*

```
long l1 = 928999999L;
float f = (float) l1;//on peut s'en passer du cast explicite
System.out.println(f);  //9.29E8
long l2 = (long) f;
System.out.println(l2);  //929000000
```

Casts entre types primitifs

- Les affectations entre types primitifs peuvent utiliser un *cast* implicite si elles ne peuvent provoquer qu'une perte de précision (ou, encore mieux, aucune perte)

```
int i = 130;
double x = 20 * i;
```

- Sinon, elles doivent comporter un *cast* explicite

```
short s = 65; // cas particulier affectation int "petit"
s = 1000000; // provoque une erreur de compilation
int i = 64;
byte b = (byte)(i + 2); // b = 66
char c = i; // caractère dont le code est 64  '@'
b = (byte)128; // b = -128 !
```

Casts entre entiers et caractères

- La correspondance **char** → **int**, **long** s'obtient par *cast* implicite
- Les correspondances:
  - **char** → **short**, **byte** , et
  - **long**, **int**, **short, byte** → **char** nécessitent un *cast* explicite (entiers sont signés et pas les **char**)

```
int i = 80;
char c = 68; // caractère dont le code est 68
c = (char)i;
i = c;
short s = (short)i;
char c2 = s; // provoque une erreur
```

Conversions de types (casts)

- Conversions des types primitifs

Conversion vers

Conversion de	boolean	byte	short	char	int	long	float	double
boolean	–	N	N	N	N	N	N	N
byte	N	–	E	R	E	E	E	E
short	N	R	–	R	E	E	E	E
char	N	R	R	–	E	E	E	E
int	N	R	R	R	–	E	E*	E
long	N	R	R	R	R	–	E*	E*
float	N	R!	R!	R!	R!	R!	–	E
double	N	R!	R!	R!	R!	R!	R*	–

N : pas de conversion possible
E : conversion automatique

R : conversion forcée (restrictive)
***** : perte de précision (promotion)
! : valeur tronquée

Opérateurs: ordre de priorité

Level	Operator	Description	Associativity
16	() [] .	parentheses array access member access	left-to-right
15	++ --	unary post-increment unary post-decrement	left-to-right
14	+ - ! ~ ++ --	unary plus unary minus unary logical NOT unary bitwise NOT unary pre-increment unary pre-decrement	right-to-left
13	() new	cast object creation	right-to-left
12	* / %	multiplicative	left-to-right
11	+ - +	additive string concatenation	left-to-right
10	<< >>>>>	shift	left-to-right
	< <=		
	>>>	shift	
9	< <= > >= instanceof	relational	left-to-right
8	== !=	equality	left-to-right
7	&	bitwise AND	left-to-right
6	^	bitwise XOR	left-to-right
5	|	bitwise OR	left-to-right
4	&&	logical AND	left-to-right
3	||	logical OR	left-to-right
2	?:	ternary	right-to-left
1	= += -= *= /= %= &= ^= |= <<= >>= >>>=	assignment	right-to-left
0	->	lambda expression arrow	right-to-left

Attention : Lors d'une opération sur des opérandes de types différents, le compilateur détermine le type du résultat en prenant le type le plus précis des opérandes. Par exemple, une multiplication d'une variable de type float avec une variable de type double donne un résultat de type double. Lors d'une opération entre un opérande entier et un flottant, le résultat est du type de l'opérande flottant.

L'arithmétique entière

Pour les types numériques entiers, Java met en œuvre une sorte de mécanisme de conversion implicite vers le type int appelée promotion entière. Ce mécanisme fait partie des règles mises en place pour renforcer la sécurité du code.

dans un expression arithmétique les variables de type **byte** ou **short** sont converties vers le type **int**

```
short s1 = 5 + 2;
```
s1 vaut 7

```
short s2 = s1 + 3;
```
Error: incompatible types: possible lossy conversion from int to short

```
short s2 = (short) (s1 + 3);
```
s2 vaut 10

```
short s3 = s1 + s2;
```
Error: incompatible types: possible lossy conversion from int to short

```
short s3 = (short) (s1 + s2);
```
s3 vaut 17

Les opérandes et le résultat de l'opération sont convertis en type int. Le résultat est affecté dans un type short : il y a donc risque de perte d'informations et donc une erreur est émise à la compilation. Cette promotion évite un débordement de capacité sans que le programmeur soit pleinement conscient du risque : il est nécessaire, pour régler le problème, d'utiliser une conversion explicite ou cast.

La division par zéro pour les types entiers lève l'exception ArithmeticException.

L'arithmétique en virgule flottante

Avec des valeurs float ou double, la division par zéro ne produit pas d'exception mais le résultat est indiqué par une valeur spéciale qui peut prendre trois états :
•indéfini : Float.NaN ou Double.NaN (not a number)
•indéfini positif : Float.POSITIVE_INFINITY ou Double.POSITIVE_INFINITY, + ∞
•indéfini négatif : Float.NEGATIVE_INFINITY ou Double.NEGATIVE_INFINITY, - ∞
Conformément à la norme IEEE754, ces valeurs spéciales représentent le résultat d'une expression invalide NaN, une valeur supérieure au plafond du type pour infini positif ou négatif.

X	Y	X / Y	X % Y
valeur finie	0	+ ∞	NaN
valeur finie	+/- ∞	0	x
0	0	NaN	NaN
+/- ∞	valeur finie	+/- ∞	NaN
+/- ∞	+/- ∞	NaN	NaN

Exemple :

```
01./* test sur la division par zero de
nombres flottants */
02.
03.class test2 {
04.public static void main (String args[]) {
05.float valeur=10f;
06.double resultat = valeur / 0;
07.System.out.println("index = " + resultat);
08.}
09.}
```

L'incrémentation et la décrémentation

```
public static void main(String[ ] args){

int n1=0;
int n2=0;
System.out.println(n1++);//est équivalent à System.out.println(x); x = x + 1;
System.out.println(++n1);//est équivalent à x = x + 1; System.out.println(x);
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);
}
```

```
0
2
n1 = 2 n2 = 0
n1 = 0 n2 = 1
n1 = 2 n2 = 2
n1 = 2 n2 = 2
```

Constantes

- variable dont la valeur ne peut plus être changée une fois fixée
  - se déclare avec le mot-clé final :

```
final double PI = 3.14159;
```

ce qui interdit d'écrire ensuite…

```
PI = 3.14; //ERREUR…
```

  - possibilité de calculer la valeur de la constante plus tard à l'exécution, et ailleurs qu'au niveau de la déclaration :

```
final int MAX_VAL;
//OK : constante "blanche"
//…
MAX_VAL = lireValeur();
```

Affichage sur la console

- Pour afficher une valeur littérale ou le contenu d'une variable sur la console de sortie, on utilise les lignes de code suivantes :
  - **`System.out.print(…);`** // Affichage (reste sur la même ligne)
  - **`System.out.println(…);`** // Affichage et retour à la ligne
- Exemples :

```
System.out.println("Résultats");
System.out.println("---------");
```

```
Résultats
---------
```

```
int  size = 123;
char unit = 'm';
System.out.print("Longueur : ");
System.out.print(size);
System.out.print(" ");
System.out.println(unit);
```

```
Longueur : 123 m
```

Même affichage mais en utilisant l'opérateur de concaténation (+)

```
int  size = 123;
char unit = 'm';
System.out.println("Longueur : " + size + " " + unit);
```

Instructions conditionnelles

Syntaxe

```
if ( expression_booléenne ) instruction
```

ou bien

```
if ( expression_booléenne ) instruction_A else instruction_B
```

exemple

```
if (i==j) {   j = j -1;
              i = 2 * j; }
          else
          i = 1;
```

• L'opérateur ternaire : (condition) ? valeur-vrai : valeur-faux

Exp:

```
if (niveau == 5)
        total = 10;
else
        total = 5 ;
// equivalent à total = (niveau ==5) ? 10 : 5;
System.out.println((sexe == " H ") ? " Mr " : " Mme ");
```

Emboîtement d'instructions if … else

- L'instruction `if` / `else` peut contenir dans chacune de ses deux branches une autre instruction `if` / `else` qui peut elle-même contenir une autre instruction …
- On peut ainsi avoir plusieurs niveaux d'emboîtement formant une logique plus ou moins complexe.
- En l'absence de blocs d'instructions (`{…}`), **la clause `else` se rapporte toujours au `if` précédent** (sans `else`) **le plus proche** (l'indentation ne change pas la logique du code !).

```
if (x >= 2)
  if (x <= 20)
    status = 1;
else
  if (x <= 10)
    status = 2;
  else
    status = 3;
```

```
if (x >= 2)
  if (x <= 20)
    status = 1;
  else
    if (x <= 10)
      status = 2;
    else
      status = 3;
```

```
if (x >= 2) {
  if (x <= 20)
    status = 1;
}
else {
  if (x < 0)
    status = 0;
  else
    status = 2;
}
```

L'indentation est importante pour la lisibilité du code !

Cas multiples

- l'utilisation de if / else peut s'avérer lourde quand on doit traiter plusieurs sélections et de multiples alternatives
- pour cela existe en Java le **switch** / **case** assez identique à celui de C/C++

```
switch (expr) {
case cst1:
// instructions si expr==cst1
break;
case cst2:
// instructions si expr==cst2
case cst3:
// instructions si
// expr==cst3 || expr==cst2
break;
default:
// instructions si aucune
// des valeurs prévues
break;
}
```

- Si une instruction case ne contient pas de break alors les traitements associés au case suivant sont exécutés.
- Il est possible d'imbriquer des switch

- expr : des types primitifs d'une taille maximum de 32 bits (byte, short, int, char) ou bien type **énuméré** (défini avec enum) ou String depuis java 7
- cst1, ... : littéral ou constante (final)

Instruction switch

Java 7

```
String typeOfDay;
switch (dayOfWeek) {
  case "Monday":
    typeOfDay = "Start of work week";       break;
  case "Tuesday":
  case "Wednesday":
  case "Thursday":
    typeOfDay = "Midweek";                  break;
  case "Friday":
    typeOfDay = "End of work week";         break;
  case "Saturday":
  case "Sunday":
    typeOfDay = "Weekend";                  break;
  default:
    typeOfDay = "Invalid day of the week";  break;
}
```

Attention : La comparaison prend en compte les minuscules/majuscules

Boucles de répétitions

boucle tantque … faire - instruction **while ()**

- Syntaxe **while (** *expression booléenne* **)**
  instruction
- L'instruction est exécutée 0, 1 ou *n* fois

Exemple

```
int i = 0;
int  somme = 0;
while (i <= 10) {
somme += i;
i++;
}
System.out.println("Somme des 10 premiers entiers" + somme);
```

Boucles de répétitions

boucle répeter … jusqu'à – instruction **do while ()**

- Syntaxe **do{**
  instruction
  }while (*expression booléenne* **) ;**
- L'instruction est exécutée 1 ou *n* fois

Exemple

```
int i = 0;
int somme = 0;
do
{
        somme += i;
        i++;
} while (i <= 10);
System.out.println("Somme des 10 premiers entiers" +
somme);
```

Boucles d'itération

for (initialisations **;**boolExpr **;** incréments**)**

{

// Corps de la boucle

}

initialisations : déclaration et/ou affectations, séparées par des virgules

incréments : expressions séparées par des virgules

équivalent à :

```
initialisations;
while (boolExpr) {
//Corps de la boucle
incréments;
}
```

Exemple:

```
int  i;
int somme = 0;
for (i = 0; i <= 10; i++) {
   somme += i;
}
System.out.println("Somme des 10 premiers entiers " + somme);
```

```
for  (i=0,j=1;i<10&&(i+j)<10; i++,j*=2) { //… }
for ( ; ; ) ;
```

La condition peut ne pas porter sur l'index de la boucle :0

Exemple :

```
1.boolean trouve = false;
2.for (int i = 0 ; !trouve ; i++ ) {
3.if ( tableau[i] == 1 ) {
4.trouve = true;
5.... //gestion de la fin du parcours du tableau
6.}
7.}
```

Il est possible de nommer une boucle pour permettre de l'interrompre même si cela est peu recommandé :

Exemple :

```
01.int compteur = 0;
02.boucle:
03.while (compteur < 100) {
04.
05.for(int compte = 0 ; compte < 10 ; compte ++)
{
06.compteur += compte;
07.System.out.println("compteur = "+compteur);
08.if (compteur> 40) break boucle;
09.}
10.}
```

Entrées/sorties sur console

- Affichage sur la console
  - `System.out.println(`*chaîne de caractères à afficher*`)` avec retour à ligne
  - `System.out.print(`*chaîne de caractères à afficher*`)` sans retour à la ligne

  - chaîne de caractères peut être :
    - une constante chaîne de caractères (`String`)

```
System.out.println("coucou");
```

    - une expression de type `String`

```
int age = 30;
System.out.println(age);
```

Ici **age** est une variable de type **int**
Elle est **automatiquement** convertie en **String**

    - une combinaison (concaténation) de constantes et d'expressions de type String. La concaténation est exprimée à l'aide de l'opérateur **+**

```
double poids = 64.5;
System.out.println("L'age de la personne est " + age + " son poids " + poids);
```

age (int) et **poids** (double) sont **automatiquement** converties en **String**

Entrées/sorties sur console

- Lecture de valeurs au clavier
  - dans les versions initiales de Java il n'y avait pas de moyen "simple" de faire ces opérations. Ce n'est plus le cas, depuis la version 5 (1.5) de Java
  - utiliser la classe **Scanner** du package standard java.util (version JDK 1.5 et supérieur).

Utilisation de la classe Scanner


```java
/**
 *   programme de demonstration de la classe Scanner qui fournit
 *   des méthodes de lecture au clavier simples pour les
 *   types de données de base les plus courants.
 *
 */

import java.util.Scanner;

public class Scan {

   public static void main(String[] args){

           Scanner sc = new Scanner(System.in);

       System.out.print("entrez une chaine de caractères : ");
           String s = sc.next();
       System.out.println("chaine lue : " + s);

       System.out.print("entrez un entier : ");
           int i = sc.nextInt();
       System.out.println("entier lu : " + i);

       System.out.print("entrez une réel (float) : ");
           float f = sc.nextFloat();
       System.out.println("réel (float) lu : " + f);

       System.out.print("entrez une réel (double) : ");
           double d = sc.nextDouble();
       System.out.println("réel (double) lu : " + d);

       System.out.print("entrez un booléen : ");
           boolean b = sc.nextBoolean();
       System.out.println("booleen lu : " + b);

           System.out.print("entrez deux coordonnées x et y séparées par un espace : ");
           float x = sc.nextFloat();
           float y = sc.nextFloat();
       System.out.println("x : " + x  + " y : " + y);
```

Utilisation de la classe Scanner

- ***Attention*** *: il y a un type de variable primitive qui n'est pas pris en compte par la classe Scanner ; il s'agit des variables de type* ***char****.*
- *Voici comment on pourrait récupérer un caractère :*

```
System.out.println("Saisissez une lettre :");
Scanner sc = new Scanner(System.in);
String str = sc.nextLine();
char carac = str.charAt(0);
System.out.println("Vous avez saisi le caractère : " + carac);
```

Qu'est-ce que nous avons fait ici ?
Nous avons récupéré une chaîne de caractères, puis nous avons utilisé une méthode de l'objet String (ici, ***charAt(0)*** *) afin de récupérer le premier caractère saisi !*
Même si vous tapez une longue chaîne de caractères, l'instruction ***charAt(0)*** *ne renverra que le premier caractère...*