

Éléments de base en Java

P.O.O.

LMI 2 – Semestre 4 – Option Info

Année 2008-09

Retour vers les objets

- Les classes

- Portée des variables

- Tester l'égalité

- Passage de paramètres

Les tableaux

Les types primitifs

- Les différents types primitifs

- Passage type primitif - instance d'une classe enveloppe

Les opérateurs

- Les différents opérateurs

- Exercices !

Les structures de contrôle

Retour vers les objets

Les classes

Portée des variables

Tester l'égalité

Passage de paramètres

Les tableaux

Les types primitifs

Les différents types primitifs

Passage type primitif - instance d'une classe enveloppe

Les opérateurs

Les différents opérateurs

Exercices !

Les structures de contrôle

Une classe est définie par :

- ▶ ses attributs, ou variables
- ▶ ses méthodes

Les attributs sont

- ▶ définis par leur **type** ;
- ▶ qualifiés par un **quantificateur d'accès** (**private**, ...)

Les méthodes sont définies par leur signature.

On appelle **signature d'une méthode**

- ▶ son nom,
- ▶ le type de ce qu'elle retourne,
- ▶ et la nature de ses arguments.

C'est ce qu'on appelle parfois aussi son entête.

- ▶ une méthode peut définir des variables locales ;
- ▶ une définition de variable locale intervient n'importe où dans la méthode
- ▶ mais une variable ne peut être utilisée qu'après sa déclaration
- ▶ et uniquement dans le même bloc d'instruction que sa déclaration
- ▶ une variable locale ne peut jamais être qualifiée par un quantificateur d'accès !

- ▶ mécanisme classique de portée pour les variables locales à une fonction. Les variables locales ont une portée limitée **entre l'endroit où elles sont déclarées et la fin du bloc dans lequel elles sont déclarées** ;
- ▶ lorsqu'on veut différencier un paramètre formel (prioritaire dans sa fonction) et une variable d'instance de l'objet, de même nom que le paramètre : **this**

```
class Livre{
    private String titre, auteur;
    Livre(String titre, String auteur){
        this.titre = titre;
        this.auteur = auteur;
    }
}
```

- ▶ une variable d'un type primitif contient sa valeur
- ▶ une variable d'un type objet contient une référence vers l'objet qu'il désigne

Conséquences sur le test d'égalité :

- ▶ sur des variables d'un type primitif

```
int a, b;  
...  
if (a==b) {...}
```

comportement attendu : teste si la valeur de **a** et la valeur de **b** sont égales.

- ▶ sur des variables d'un type objet

```
// class Livre
boolean aMemeAuteurQue(Livre l){
    return auteur == l.auteur;
}
```

Teste si `auteur` et `l.auteur` contiennent la référence du même objet chaîne de caractères.

Ce n'est pas toujours ce qui est souhaité...

Test d'égalité sur des objets

```
String mahfouz = "N. Mahfouz";  
Livre l1 = new Livre("Les mille et une nuits",  
                    mahfouz,"Flon");  
Livre l2 = new Livre("La belle du Caire",  
                    mahfouz,"Payard");  
if (l1.aMemeAuteurQue(l2)) { // le test va réussir...  
    ...  
}
```

Mais

```
Livre l1 = new Livre("Les mille et une nuits",  
                    "N. Mahfouz","Flon");  
Livre l2 = new Livre("La belle du Caire",  
                    "N. Mahfouz","Payard");  
if (l1.aMemeAuteurQue(l2)) {  
    // pas sûr que le test réussisse ...  
}
```



Test d'égalité sur des objets

La méthode `equals` fait un test sur le contenu de l'objet.

```
// class Livre
boolean aMemeAuteurQue(Livre l){
    return auteur.equals(l.auteur);
}
// ailleurs
Livre l1 = new Livre("Les mille et une nuits",
                    "N. Mahfouz","Flon");
Livre l2 = new Livre("La belle du Caire",
                    "N. Mahfouz","Payard");
if (l1.aMemeAuteurQue(l2)) {
    // le test va réussir!!
    ...
}
```

`equals` fait un test d'équivalence.

On peut aussi préciser, pour une classe que l'on écrit, ce qu'on souhaite comme équivalence en redéfinissant la méthode `equals`

```
public boolean equals(Livre l){  
    return titre.equals(l.titre) && auteur.equals(l.auteur)  
        && editeur.equals(l.editeur);  
}
```

Par défaut, la méthode `equals` teste l'égalité (au sens de `equals`) sur tous les attributs de l'objet. Ici, on décide de limiter les tests sur trois attributs.

Dans les méthodes, le passage de paramètres se fait uniquement par valeur :

Une **copie** du contenu du paramètre effectif est transmis à la méthode.

```
void neDoublePas(int i){  
    i = 2*i;  
    System.out.println("valeur de i: "+i);  
}  
...  
int i=3;  
System.out.println("i = "+i);  
neDoublePas(i);  
System.out.println("i = "+i);
```

Passage de paramètres de type objet

Le paramètre formel contient une copie de la référence vers l'objet : il pointe donc le même objet que le paramètre effectif. Il peut donc modifier l'objet en utilisant les méthodes (les accesseurs) de l'objet lui-même.

Passage de paramètres de type objet

Le paramètre formel contient une copie de la référence vers l'objet : il pointe donc le même objet que le paramètre effectif. Il peut donc modifier l'objet en utilisant les méthodes (les accesseurs) de l'objet lui-même.

```
// class ServiceAchat
public void achatExemplairesLivre(Livre l, int nbEx){
    l.nouvelExemplaire(nbEx);
}
```

```
//class Livre
public void nouvelExemplaire(int nb){
    if (nb > 0)
        nbreExemplaires += nb;
}
```

Retour vers les objets

- Les classes

- Portée des variables

- Tester l'égalité

- Passage de paramètres

Les tableaux

Les types primitifs

- Les différents types primitifs

- Passage type primitif - instance d'une classe enveloppe

Les opérateurs

- Les différents opérateurs

- Exercices !

Les structures de contrôle

On déclare un tableau en ajoutant des crochets derrière le type des éléments du tableau :

```
Livre[] biblio;
```

biblio est déclaré mais n'est pas encore créé.

Instanciation d'un tableau :

```
biblio = new Livre[3];
```

Le nombre d'éléments du tableau est donné à l'instanciation du tableau.

On remplit le tableau :

```
biblio[0] = new Livre("L'avenir de l'eau", "E. Orsenna", "Flourens")  
biblio[1] = new Livre("La moitié d'une vie", "V.S. Naipaul", "Gallimard")  
biblio[2] = new Livre("Les mille et une nuits", "N. Mahfouz", "Gallimard")
```

```
for (int i = 0; i < biblio.length; i++){  
    System.out.println(biblio[i]);  
}
```

- ▶ `length` contient le nombre de cellules du tableau
- ▶ les cellules sont indicées de 0 à `length - 1`
- ▶ **Attention!** `length` n'est pas le nombre d'éléments réellement présents dans le tableau, mais la nombre de cellules.

Retour vers les objets

Les classes

Portée des variables

Tester l'égalité

Passage de paramètres

Les tableaux

Les types primitifs

Les différents types primitifs

Passage type primitif - instance d'une classe enveloppe

Les opérateurs

Les différents opérateurs

Exercices !

Les structures de contrôle

entiers	byte	entier signé sur 8 bits	-2^7 à $2^7 - 1$ (-128 à 127)
	short	entier signé sur 16 bits	-2^{15} à $2^{15} - 1$ (-32768 à 32767)
	int	entier signé sur 32 bits	-2^{31} à $2^{31} - 1$
	long	entier signé sur 64 bits	-2^{63} à $2^{63} - 1$
réels	float	signé sur 32 bits	simple précision
	double	signé sur 64 bits	double précision

- ▶ Constantes entières : par défaut `int`.
Deviennent `long` si valeur suivie d'un `l` ou d'un `L`.
`static long échelleCarte[] = {100000L,10000L}`

- ▶ Constantes entières : par défaut `int`.
Deviennent `long` si valeur suivie d'un `l` ou d'un `L`.
`static long échelleCarte[] = {100000L,10000L}`
- ▶ Constantes réelles : par défaut `double`.
Deviennent `float` si valeur suivie d'un `f` ou d'un `F`.
`static float tableTVA[] = {0.055,0.0206,0.25}`
`// erreur à la compilation car restriction de`
`domaine`
`// valeurs double déclarées comme float.`

- ▶ Constantes entières : par défaut `int`.

Deviennent `long` si valeur suivie d'un `l` ou d'un `L`.

```
static long échelleCarte[] = {100000L,10000L}
```

- ▶ Constantes réelles : par défaut `double`.

Deviennent `float` si valeur suivie d'un `f` ou d'un `F`.

```
static float tableTVA[] = {0.055,0.0206,0.25}
```

// erreur à la compilation car restriction de
domaine

// valeurs `double` déclarées comme `float`.

La bonne déclaration est :

```
static float tableTVA[] = {0.055f,0.0206f,0.25f}
```

C'est le type `char`.

- ▶ Les constantes caractères doivent être entre ' et ' ('A','B',...)
- ▶ on peut comparer des valeurs de type `char` avec les opérateurs usuels de comparaison ;
- ▶ le type `char` Java correspond à des caractères codés en Unicode sur 16 bits.
- ▶ un `char` est donc un entier de l'intervalle $[0, 65535 (= 2^{16} - 1)]$.
Un `char` peut intervenir dans un calcul, partout où un `int` peut intervenir.

Attention : Java effectue une conversion de type uniquement s'il y a **élargissement du domaine**.

```
v_short = v_char ;
```

Attention : Java effectue une conversion de type uniquement s'il y a **élargissement du domaine**.

```
v_short = v_char ;//refusé
```

Attention : Java effectue une conversion de type uniquement s'il y a **élargissement du domaine**.

```
v_short = v_char ; //refusé  
v_char = v_short ;
```

Attention : Java effectue une conversion de type uniquement s'il y a **élargissement du domaine**.

```
v_short = v_char ;//refusé  
v_char = v_short ;//refusé
```

Attention : Java effectue une conversion de type uniquement s'il y a **élargissement du domaine**.

```
v_short = v_char ; //refusé  
v_char = v_short ; //refusé  
v_char = v_int ;
```

Attention : Java effectue une conversion de type uniquement s'il y a **élargissement du domaine**.

```
v_short = v_char ; //refusé  
v_char = v_short ; //refusé  
v_char = v_int ; //refusé
```

Attention : Java effectue une conversion de type uniquement s'il y a **élargissement du domaine**.

```
v_short = v_char ; //refusé  
v_char = v_short ; //refusé  
v_char = v_int ; //refusé  
v_int = v_char ;
```

Attention : Java effectue une conversion de type uniquement s'il y a **élargissement du domaine**.

```
v_short = v_char ; //refusé  
v_char = v_short ; //refusé  
v_char = v_int ; //refusé  
v_int = v_char ; //OK
```

On peut néanmoins forcer une conversion de type de manière explicite :

```
char car = 'A' ;  
short sh = (short) car ;  
System.out.println(sh) ;
```

On peut néanmoins forcer une conversion de type de manière explicite :

```
char car = 'A' ;  
short sh = (short) car ;  
System.out.println(sh) ;
```

65

Le type `boolean` a les deux valeurs `true` et `false`.

- ▶ c'est un type à part entière, indépendant des types numériques ou `char` ;
- ▶ pas de conversion possible ;
- ▶ en particulier, pas comme en C ou en C++, où
 - ▶ une expr. évaluée à zéro n'est pas équivalente à `false`
 - ▶ une expr. évaluée à une valeur différente de zéro n'est pas équivalente à `true`

(différence avec le C et le C++)

Il existe un type objet correspondant à chaque type primitif :

- ▶ `Character`
- ▶ `Boolean`
- ▶ `Integer`
- ▶ `Float`
- ▶ `Double`
- ▶ `Long`
- ▶ `Short`
- ▶ `Byte`

Elles contiennent toutes :

- ▶ un constructeur prenant en argument une valeur du type primitif correspondant
- ▶ un constructeur avec une `String` en argument
- ▶ et une fonction qui retourne la valeur de l'objet dans le type primitif correspondant

Les instances de ces classes sont **non-modifiables**.

- ▶ `Character(char c)` et `char charValue()`
- ▶ `Boolean(boolean b)` et `boolean booleanValue()`
- ▶ `Integer(int i)` et `int intValue()`, `short shortValue()`, `float floatValue()`...

Jusqu'au JDK 1.4

```
Integer[] tabInt = new Integer[3];  
tabInt[0] = new Integer(5);  
int i = tabInt[0].intValue();
```

À partir du JDK 1.5, c'est le compilateur qui fait le travail !
L'empaquetage/dépaquetage est fait automatiquement.

```
Integer[] tabInt = new Integer[3];  
tabInt[0] = 5;  
int i = tabInt[0];
```

Jusqu'au JDK 1.4

```
Integer i = new Integer(3);  
// pour incrémenter i  
i = new Integer(i.intValue()+1);
```

Avec le JDK 1.5, la syntaxe est plus légère (mais l'implémentation est la même!)

```
Integer i = new Integer[3];  
// pour incrémenter i  
i = i+1;
```

C'est le compilateur qui effectue la conversion vers `int` et la création d'un nouvel `Integer`.

Les instances de Integer sont non-modifiables

Lorsqu'on a des calculs à faire, il faut les faire sur les types primitifs !

```
public Integer beaucoupDeTrucsACalculer(Integer i){  
    // conversion dans un type primitif  
    int petitI = i;  
    // tous les calculs se font maintenant sur petitI  
    ...  
    // et la conversion dans l'autre sens sera faite automatiquement  
    return petitI;  
}
```

Retour vers les objets

- Les classes

- Portée des variables

- Tester l'égalité

- Passage de paramètres

Les tableaux

Les types primitifs

- Les différents types primitifs

- Passage type primitif - instance d'une classe enveloppe

Les opérateurs

- Les différents opérateurs

- Exercices !

Les structures de contrôle

Opérateurs usuels : +, -, *, / et %

Ils conservent le type des arguments.

Opérateurs usuels : +, -, *, / et %

Ils conservent le type des arguments.

```
int a = 3; int b = 2;  
int c = 3/2; // c == 1  
float cf = (float)3/2; // cf == 1.5
```

Exemple

```
public class Operateurs{
    public static void main(String[] args){
        int a, b;
        if (args.length==2){
            a = Integer.parseInt(args[0]);
            b = new Integer(args[1]);
        }
        else {
            a = 5;
            b = 4;
        }
        System.out.println(" La résultat de "+
            a+" / "+b+" est "+(a/b) );
        System.out.println(" La résultat de "+
            a+" % "+b+" est "+(a%b) );
    }
}
```

Exemple

```
float af = (float) a;
float bf = (float) b;
System.out.println(" La résultat de "+
    af+" / "+bf+ " est "+(af/bf) );
System.out.println(" La résultat de "+
    af+" % "+bf+" est "+(af%bf) );
char c = 'A' + 1;
System.out.println("(A+1) donne "+c);
} // main
} // class Operateurs
```

L'opérateur +

L'opérateur `+` est également l'opérateur de concaténation des chaînes de caractères ! Utilisation de la méthode statique

`valueOf()` de la classe `String` pour effectuer des conversions explicites :

L'opérateur + est également l'opérateur de concaténation des chaînes de caractères ! Utilisation de la méthode statique

`valueOf()` de la classe `String` pour effectuer des conversions explicites :

```
public String concatene(int i, int j){  
    return String.valueOf(i)+String.valueOf(j);  
}
```

```
String s = concatene(5,6); // s="56"
```

- ▶ `+op` : promeut `op` au rang de `int` s'il est un `byte`, un `short`, ou un `char` ;
- ▶ `-op` : opposé de `op` ;

et les raccourcis suivants (à la C) :

- ▶ `op++` : l'expression complète est évaluée, puis `op` est incrémenté de 1 ;
- ▶ `++op` : `op` est incrémenté de 1 puis l'expression complète est évaluée ;
- ▶ les opérateurs `op--` et `--op`.

```
int i = 0 ;  
int[] tab = new int[5] ;  
tab[i++] = 0 ;
```

```
int i = 0 ;  
int[] tab = new int[5] ;  
tab[i++] = 0 ;  
// tab[0] vaut 0 et i vaut 1  
tab[++i] = 10 ;
```

```
int i = 0 ;  
int[] tab = new int[5] ;  
tab[i++] = 0 ;  
// tab[0] vaut 0 et i vaut 1  
tab[++i] = 10 ;  
//tab[2] vaut 10 et i vaut 2
```

Opérateurs classiques : `<`, `<=`, `>`, `>=`, `==`, `!=`.

Attention :

- ▶ l'opérateur `==` se manipule différemment selon qu'on compare deux variables de types primitifs ou deux références d'objets ;
- ▶ comme dans tout langage, il est fortement déconseillé d'utiliser l'opérateur `==` avec des `float` ou des `double`.

Opérateurs `&&`, `||`, `!`, `&`, `|`, `^`

Significations respectives : et, ou, non, et, ou, ou exclusif

Opérateurs `&&` et `||` : comportement coupe-circuit(à la Scheme).

Ils n'évaluent la deuxième opérande que si nécessaire.

```
Integer i ;
```

```
...
```

```
return (i != null) & i.compareTo(10)<0 ;
```

Opérateurs `&&`, `||`, `!`, `&`, `|`, `^`

Significations respectives : et, ou, non, et, ou, ou exclusif

Opérateurs `&&` et `||` : comportement coupe-circuit(à la Scheme).

Ils n'évaluent la deuxième opérande que si nécessaire.

```
Integer i ;
```

```
...
```

```
return (i != null) & i.compareTo(10)<0 ;
```

Risque d'erreur ! Il faut écrire :

```
return (i != null) && i.compareTo(10)<0 ;
```

ou

```
return (i != null) && i<10 ;
```

Opérateurs de décalage et opérateurs logiques

Opérateurs classiques de décalage de bits vers la gauche ou vers la droite (<< et >>).

```
int d = 18;  
System.out.println("Decalage a gauche de un "  
                    +(d<<1)+" de deux "+(d<<2));
```

Opérateurs classiques de décalage de bits vers la gauche ou vers la droite (<< et >>).

```
int d = 18;
System.out.println("Decalage a gauche de un "
                  +(d<<1)+" de deux "+(d<<2));

// provoque l'affichage de 36 puis 72
System.out.println("Decalage a droite de un "
                  +(d>>1)+" de deux "+(d>>2));
```

Opérateurs classiques de décalage de bits vers la gauche ou vers la droite (<< et >>).

```
int d = 18;
System.out.println("Decalage a gauche de un "
                  +(d<<1)+" de deux "+(d<<2));

// provoque l'affichage de 36 puis 72
System.out.println("Decalage a droite de un "
                  +(d>>1)+" de deux "+(d>>2));

//provoque l'affichage de 9 puis de 4.
```

Opérateurs logiques (opérations bit à bit) : `&`, `|`, `^` (ici, l'opérateur `^` est le ou exclusif).

```
System.out.println("36 & 12 = "+(36&12));  
System.out.println("36 | 12 = "+(36|12));  
System.out.println("36 ^ 12 = "+(36^12));
```

Opérateurs d'affectation

C'est l'opérateur =, mais il y a aussi les raccourcis

+=, -=, /=, %=, *=, &=, |=, <<=, >>=

Opérateurs d'affectation

C'est l'opérateur =, mais il y a aussi les raccourcis

`+=, -=, /=, %=, *=, &=, |=, <<=, >>=`

```
int i = 5;  
i += 2;
```

est équivalent à

```
int i = 5;  
i = i+2;
```

Attention aux opérateurs non commutatifs :

`a /= b;` est équivalent à `a = a/b;`

`instanceof` permet de tester si une variable est une instance d'une classe.

? : C'est l'opérateur ternaire si-sinon.

```
int i=5;
System.out.println("Il y a "+i+" étudiant"
                    +(i>1?"s":""));
```

`new` C'est l'opérateur de création d'un objet (opérateur d'instanciation)

[] opérateur de déclaration et d'accès à un tableau.

Quelques exercices du tutorial.

1. Quels sont les opérateurs de l'expression Java suivante :

```
tableauDEntiers[j] > tableauDEntiers[j+1]
```

2. Soit le bout de code suivant :

```
int i = 10 ;  
int n = i++%5 ;
```

2.1 quelles sont les valeurs de `i` et de `n` après l'exécution de ces instructions ?

2.2 même question si on remplace `i++` par `++i`.

3. Quelle est la valeur de `i` après l'exécution des instructions

suivantes :

```
int i = 8 ; | int i = 17 ;  
i >>= 2 ; | i >>= 1 ;
```

Retour vers les objets

- Les classes

- Portée des variables

- Tester l'égalité

- Passage de paramètres

Les tableaux

Les types primitifs

- Les différents types primitifs

- Passage type primitif - instance d'une classe enveloppe

Les opérateurs

- Les différents opérateurs

- Exercices !

Les structures de contrôle

Structures conditionnelles :

- ▶ structure `if (expr-bool)...else`
- ▶ Pour énumérer différents cas. Uniquement avec une variable de type `int` ou `enum`.

```
int jour = 8;
switch (jour) {
    case 1:  System.out.print("Lundi"); break;
    case 2:  System.out.print("Mardi"); break;
    case 3:  System.out.print("Mercredi"); break;
    case 4:  System.out.print("Jeudi"); break;
    case 5:  System.out.print("Vendredi"); break;
    case 6:  System.out.print("Samedi"); break;
    case 7:  System.out.print("Dimanche"); break;
    default: System.out.print("pffff"); break;
}
```

```
enum JOUR {LUN,MAR,MER,JEU,VEN,SAM,DIM};  
JOUR jour;  
switch (jour) {  
    case LUN:  
    case MAR:  
    case MER:  
    case JEU:  
    case VEN: System.out.println("boulot"); break;  
    case SAM:  
    case DIM: System.out.println("we"); break;  
}
```

Structures itératives :

- ▶ `while (expr-bool){...}`
- ▶ `do {...} while (expr-bool)`
- ▶ `for (init ; expr-bool ; increment){...}`

Structures itératives :

- ▶ `while (expr-bool){...}`
- ▶ `do {...} while (expr-bool)`
- ▶ `for (init ; expr-bool ; increment){...}`

```
for ( ; ; ) { // boucle infinie
    ...
}
```

L'instruction `break` permet de quitter une boucle.