

# Programmation Orientée Objet en Java

---

Licence Fondamentale

SMI – S5

---

El Mostafa DAOUDI

*Département de Mathématiques et d'Informatique,*

*Faculté des Sciences*

*Université Mohammed Premier*

*Oujda*

*[m.daoudi@fso.ump.ma](mailto:m.daoudi@fso.ump.ma)*

*Septembre 2012*

---

El Mostafa DAOUDI- p. 1

## Quelques Références:

- Cours JAVA SMI S5, Daoudi 2011/2012

## Livre:

- Titre: Programmer en JAVA ,  
Auteur Claude Delnoy,  
Editeur: Eyrolles
- Thinking in Java, Bruce Eckel

## Ressources Internet:

- [http ://www.java.sun.com](http://www.java.sun.com)
- Richard Grin: <http://deptinfo.unice.fr/~grin>
- Cours Mickaël BARON - 2007
- Cours Interface graphique en Java API swing, Juliette Dibie-Barthélemy mai 2005
- .....

---

El Mostafa DAOUDI- p. 2

## Ch I. Introduction générale au langage Java

### I. Introduction

- Java est un langage orienté objet: l'entité de base de tout code Java est la classe
- Créé en 1995 par *Sun Microsystems*
- Sa syntaxe est proche du langage C
- Il est fourni avec le JDK (Java Development Kit)
  - Outils de développement
  - Ensemble de paquetages très riches et très variés
- Multi-tâches (*threads*)
- Portable grâce à l'exécution par une machine virtuelle

---

El Mostafa DAOUDI- p. 3

- En Java, **tout se trouve dans une classe**. Il ne peut y avoir de déclarations ou de code en dehors du corps d'une classe.
- La classe elle-même ne contient pas directement du code.
  - Elle contient des attributs.
  - et des méthodes (équivalents à des fonctions).
- Le code se trouve **exclusivement** dans le corps des méthodes, mais ces dernières peuvent aussi contenir des déclarations de variables locales (visibles uniquement dans le corps de la méthode).

---

El Mostafa DAOUDI- p. 4

## II. Environnement de Programmation

### 1. Compilation

- La compilation d'un programme Java ne traduit pas directement le code source en fichier exécutable. Elle traduit d'abord le code source en un code intermédiaire appelé «*bytecode*». C'est le *bytecode* qui sera ensuite exécuté par une machine virtuelle (JVM ; *Java Virtual Machine*). Ceci permet de rendre le code indépendant de la machine qui va exécuter le programme.

- *Sun* fournit le compilateur *javac* avec le JDK. Par exemple,  
`javac MonPremProg.java`

compile la classe **MonPremProg** dont le code source est situé dans le fichier **MonPremProg.java**

---

El Mostafa DAOUDI- p. 5

- Si le fichier **MonPremProg.java** fait référence, par exemple, à des classes situées dans les répertoires `/prog/exemple` et `/cours`, alors la compilation se fait de la façon suivante:

sous windows: `javac -classpath /prog/exemple ; /cours; MonPremProg.java`

sous Linux: `javac -classpath /prog/exemple : /cours; MonPremProg.java`

- On peut désigner le fichier à compiler par un chemin absolu ou relatif :  
`javac home/user2/MonPremProg.java`
- Cette compilation crée un fichier nommé «**MonPremProg.class**» qui contient le *bytecode*
- Si un système possède une JVM, il peut exécuter tous les *bytecodes* (fichiers *.class*) compilés sur n'importe quel autre système.

---

El Mostafa DAOUDI- p. 6

## 2. Exécution du *bytecode*

- Le *bytecode* doit être exécuté par une JVM. Cette JVM n'existe pas; elle est simulée par un programme qui
  - lit les instructions (en *bytecode*) du programme **.class**,
  - les traduit dans le langage machine relatif à la machine sur laquelle il sera exécuté.
  - Lance leur exécution
- Pour exécuter, Sun fournit le programme *java* qui simule une JVM. Il suffira d'utiliser la commande:  
java MonPremProg
- Si des classes d'autres répertoires sont nécessaires, alors faut alors utiliser l'option `-classpath` de la même façon que la compilation:

sous windows: java `-classpath` /prog/exemple ; /cours MonPremProg  
sous Linux: java `-classpath` /prog/exemple : /cours MonPremProg

---

El Mostafa DAOUDI- p. 7

## 3. Mon premier programme en Java

Considérons le code source suivant:

```
public class MonPremProg {  
    public static void main(String args[]) {  
        System.out.println(" Bonjour: mon premier programme Java " );  
    }  
}
```

### Important:

1. Ce code doit être sauvegarder obligatoirement dans le Fichier source nommé « MonPremProg.java »
2. Une classe exécutable doit posséder une méthode ayant la signature **public static void main(String[] args).**

---

El Mostafa DAOUDI- p. 8

Dans le cas de l'environnement JDK de SUN.

- Pour compiler, il suffit d'utiliser la commande *javac*:

```
javac MonPremProg.java
```

- Pour exécuter, il suffira d'utiliser la commande:

```
java MonPremProg
```

qui interprète le *bytecode* de la méthode *main()* de la classe *MonPremProg*

---

El Mostafa DAOUDI- p. 9

L'exécution du programme *MonPremProg* affiche à l'écran, comme résultat, la chaîne de caractères:

Bonjour: mon premier programme Java

Ceci grâce à l'instruction:

```
System.out.println(" Bonjour: mon premier programme Java ");
```

---

El Mostafa DAOUDI- p. 10

- De manière générale, dans tout programme destiné à être exécuté doit contenir une méthode particulière nommée `main()` définie de la manière suivante:

```
public static void main(String args[]) {  
    /* corps de la méthode */  
}
```

- Le paramètre `args` de la méthode `main()` est un tableau d'objets de type `String`. Il est exigé par le compilateur Java.
- La classe contenant la méthode `main()` doit obligatoirement être `public` afin que la machine virtuelle y accède.
- Dans l'exemple précédent, le contenu de la classe `MonPremProg` est réduit à la définition d'une méthode `main()`.

---

El Mostafa DAOUDI- p. 11

- Un fichier source peut contenir plusieurs classes mais une seule doit être public (dans l'exemple c'est la classe: `MonPremProg` ).
- Le nom du fichier source est identique au nom de la classe publique qu'il contient, suivi du suffixe `.java`. Dans l'exemple précédent, le fichier source doit obligatoirement avoir le nom: `MonPremProg.java`

---

El Mostafa DAOUDI- p. 12

## Ch. II Les classes et les Objets

### I. Généralité sur la Programmation Orientée Objet

- La Programmation Orientée Objet (POO) propose une méthodologie de programmation centrée sur les objets, où **un objet peut** être vu comme une entité regroupant un ensemble de données et de **méthodes** de traitement.
- Le programmeur
  - Doit d'abord identifier les **objets** qui doivent être utilisés (ou manipulés) par le programme: on commence par décider quels objets doivent être inclus dans le programme.
  - Il va ensuite écrire les traitements, **en associant chaque traitement à un objet donné.**

Il s'agit donc :

- de déterminer les objets présents dans le programme.
- d'identifier leurs données .
- de définir les traitements à faire sur ses objets .

---

El Mostafa DAOUDI- p. 13

### Intérêts de la Programmation Orientée Objet (POO)

- Programmation modulaire: Faciliter de la réutilisation de code.
- Encapsulation (Principe de la POO) et abstraction (proche du monde réel):
  - Regrouper les caractéristiques dans une classe.
  - Cacher les membres d'une classe: choix dans les niveaux de confidentialité.
- Programmation par « composants » (chaque portion de code est isolée) : Faciliter de l'évolution du code.

### Les grands principes de la POO

- l'encapsulation
- L'héritage
- Le Polymorphisme

---

El Mostafa DAOUDI- p. 14

## II. Les classes

Une classe (ou type d'objets) représente une famille d'objets qui partagent des propriétés communes.

⇒ Une classe regroupe les objets qui ont :

- La même structure (même ensemble d'attributs).
- Le même comportement (même méthodes).
  
- Les classes servent pour la création des objets
  - Un objet est une **instance d'une classe**
- Un programme orienté objet est constitué de classes qui permettent de créer des objets qui s'envoient des messages.
- L'ensemble des interactions entre les objets définit un algorithme.
- Les relations entre les classes reflètent la décomposition du programme.

---

El Mostafa DAOUDI- p. 15

## Les membres d'une classe:

- Champs (appelés aussi attributs ou données membres): l'ensemble des membres définissent l'état d'un objet (chaque objet a ses *données propres*).
- Méthodes (appelés aussi fonctions membres ou comportement): définissent un ensemble d'*opérations applicables à l'objet* (on manipule les objets par des appels de ses méthodes).

L'ensemble des méthodes est appelé l'**interface de l'objet**.

Une interface définit toutes les opérations qu'on peut appliquer à l'objet (définit tout ce qu'il est possible de "faire" avec un objet).

---

El Mostafa DAOUDI- p. 16



**Exemple:** *Rectangle* est une classe utilisée pour créer des objets représentant des rectangles particuliers.

- Elle regroupe 4 données de type réel qui caractérisent le rectangle: longueur , largeur et origine (x,y) (la position en abscisse et en ordonnée de son origine).
- On suppose qu'on peut effectuer les opérations de déplacement et de calcul de la surface du rectangle.
- Les symboles + et – sont les spécificateurs d'accès (voir plus loin)

**Exemple (notation UML)**

Rectangle	Nom de la classe
- longueur - largeur - x - y	Description des attributs
+ deplacer(int,int) + calculSurface()	Description des méthodes

---

El Mostafa DAOUDI- p. 17

**Création d'une classe**

Pour créer une classe (un nouveau type d'objets) on utilise le mot clé **class**.

La syntaxe pour créer une classe de nom **ClasseTest** est la suivante:

```
class ClasseTest{ /* ClasseTest est le nom de la classe à créer */  
    /* Corps de la classe  
    - Description des attributs (données membres)  
    - Description des méthodes  
    */  
}
```

Pour les commentaires on utilise:

```
// pour un commentaire sur une seule ligne  
/* pour un commentaire étalé sur  
plusieurs lignes. Il doit terminer avec */
```

---

El Mostafa DAOUDI- p. 18

### Exemple:

Soit « *Rectangle* » une classe utilisée pour créer des objets représentant des rectangles particuliers. Un objet de type « *Rectangle* » est caractérisé par:

- La longueur de ses cotés.
- Sa position dans le plan: cette position peut être définie par la position de son centre dans le plan. On suppose aussi que les cotés du rectangle sont parallèles à l'axe des abscisses et l'axe des ordonnées.

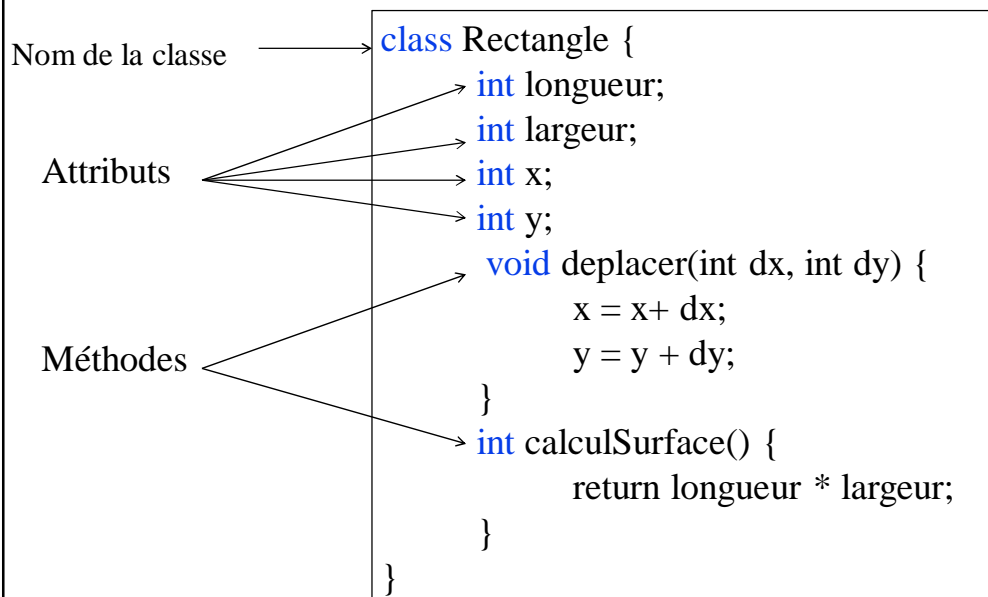
On suppose qu'on peut effectuer sur un objet de type rectangle, les opérations suivantes:

- calcul de la surface
- déplacement dans le plan

---

El Mostafa DAOUDI- p. 19

### Code java de la classe Rectangle :



---

El Mostafa DAOUDI- p. 20

### III. Création et manipulation d'Objets

#### 1. Introduction

- Une fois la classe est définie, on peut créer des objets (variables).  
⇒ Donc chaque objet est une variable d'une classe. Il a son espace mémoire, il admet une valeur propre à chaque attribut. Les valeurs des attribut caractérisent l'état de l'objet.
- L'opération de création de l'objet est appelée une **instanciation**. Un objet est aussi appelé une **instance** d'une classe. Il est référencé par une variable ayant un état (ou valeur).
- On parle indifféremment d'**instance**, de **référence** ou d'**objet**

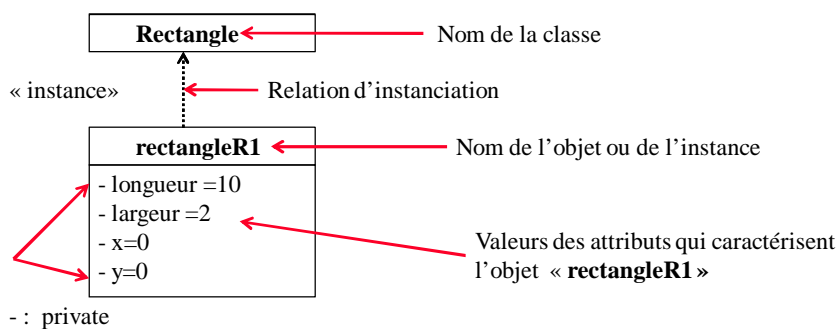
---

El Mostafa DAOUDI- p. 21

Une classe permet d'instancier plusieurs objets.

- les attributs et les méthodes d'un objet (une instance d'une classe) sont les mêmes pour toutes les instances de la classe.
- Les valeurs des attributs sont propres à chaque objet.

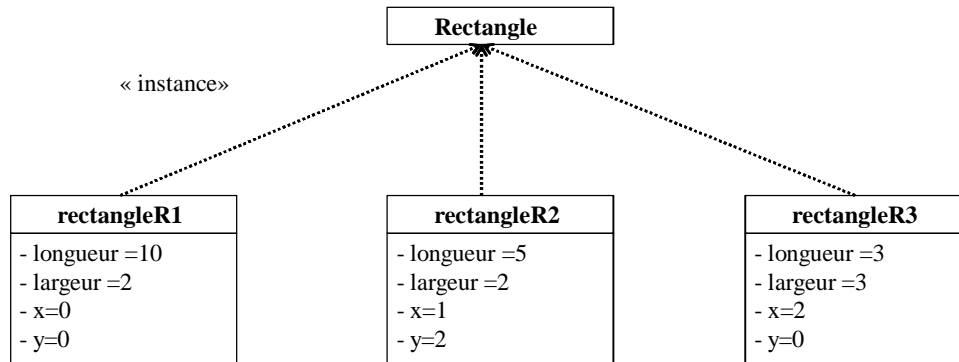
Par exemple « rectangleR1 » est une instance de la classe « Rectangle »



---

El Mostafa DAOUDI- p. 22

- Les valeurs des attributs peuvent être différents.
- ⇒ Chaque instance d'une classe possède ses propres valeurs pour chaque attribut (chaque objet a son propre état).



El Mostafa DAOUDI- p. 23

## **2. Etapes de création des Objets**

Contrairement aux types primitifs, la création d'objets se passe en deux étapes:

- Déclaration de l'objet
- Création de l'objet

### **2.1. Déclaration d'un objet:**

Chaque objet appartient à une classe. C'est une variable comme les autres. Il faut notamment qu'il soit déclaré avec son type.

#### **Syntaxe:**

```

NomDeClasse1  objetId;
NomDeClasse2  objetId1, objetId2, ...;
  
```

« NomDeClasse1 » et « NomDeClasse2 » désignent les noms de deux classes. Elles déclarent

- « objetId » comme variable de type « NomClasse1 »
- « objetId1 », « objetId2 », ... comme variables de type « NomClasse2 »

El Mostafa DAOUDI- p. 24

**Attention:**

- La déclaration d'une variable de type primitif réserve un emplacement mémoire pour stocker la variable.
- Par contre, la déclaration d'un objet, ne réserve pas une place mémoire pour l'objet, mais seulement un emplacement pour **une référence** à cet objet.

⇒ les objets sont manipulés avec des références

---

El Mostafa DAOUDI- p. 25

**Exemple:**

Considérons la classe **ClasseTest**

```
class ClasseTest {  
    // corps de la classe ClasseTest  
}
```

l'instruction:

**ClasseTest objA;**

- Déclare **objA** comme objet (variable) de type **ClasseTest**
- Définit le nom et le type de l'objet.
- Déclare que la variable **objA** est une référence à un objet de la classe **ClasseTest**. Cela veut dire qu'on va utiliser une variable **objA** qui référencera un objet de la classe **ClasseTest**.
- Aucun objet n'est créé: un objet seulement déclaré, vaut « null ».

objA

null
------

---

El Mostafa DAOUDI- p. 26

## 2.2. Création d'un objet

Après la déclaration d'une variable, on doit faire la création (et allocation) de la mémoire de l'objet qui sera référencé par cette variable.

- La création doit être demandée explicitement dans le programme en faisant appel à l'opérateur *new*.
- La création réserve de la mémoire pour stocker l'objet et initialise les attributs.

L'expression *new* **NomDeClasse()** crée un emplacement pour stocker un objet de type **NomDeClasse**.

### Important:

- avant d'utiliser un objet on doit le créer.
- la déclaration seule d'un objet, ne nous permet pas de l'utiliser.

---

El Mostafa DAOUDI- p. 27

### Exemple

```
class ClasseTest {  
    /* Corps de la classe ClasseTest */  
}
```

```
ClassTest objA ; /* déclare une référence sur l'objet objA */  
objA= new ClasseTest(); /* crée un emplacement pour stocker l'objet objA */
```

Les deux expressions précédentes peuvent être remplacées par :

```
ClassTest objA = new ClasseTest();
```

### **En générale:**

1. Chaque objet met ses données membres dans sa propre zone mémoire.
2. En général, les données membres ne sont partagées entre les objets de la même classe.

---

El Mostafa DAOUDI- p. 28

**Exemple :** Considérons la classe rectangle

```
public class Rectangle{
    int longueur;
    int largeur;
    int x;
    int y;
    public static void main (String args[]) {
        Rectangle rectangleR1; /* déclare une référence sur l'objet rectangleR1 */
        rectangleR1 = new Rectangle(); /* création de l'objet rectangleR1 */
        // rectangle R1 est une instance de la classe Rectangle
        // Les deux expressions peuvent être remplacées par :
        // Rectangle rectangleR1=new Rectangle();
    }
}
```

---

El Mostafa DAOUDI- p. 29

### **2.3. Initialisation par défaut**

La création d'un objet entraîne toujours une initialisation par défaut de tous les attributs de l'objet même si on ne les initialise pas:

Type	Valeur par défaut
boolean	false
char	'\u0000' (null)
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0.0f
double	0.0
Class	null

---

El Mostafa DAOUDI- p. 30

**Attention:** Cette garanti d'initialisation par défaut ne s'applique pas aux variables locales (variables déclarée dans un bloc: par exemple les variables locales d'une méthode) (voir plus loin).

---

El Mostafa DAOUDI- p. 31

## **IV. Les méthodes**

### **1. Définition**

En Java, chaque fonction **est définie dans une classe**. Elle est définie par :

- un type de retour
- un nom
- une liste (éventuellement vide) de paramètres **typés en entrée**
- une suite d'instructions (un bloc d'instructions) qui constitue le corps de la méthode

### **Syntaxe:**

```
typeRetour nomMethode ( « Liste des paramètres » ) {  
    /*  
        corps de la méthode: les instructions décrivant la méthode  
    */  
}
```

---

El Mostafa DAOUDI- p. 32



- nomMethode : c'est le nom de la méthode
- « Liste des paramètres » : liste des arguments de la méthode. Elle définit les types et les noms des informations qu'on souhaite passer à la méthode lors de son appel.
- typeRetour : c'est le type de la valeur qui sera retournée par la méthode après son appel. Si la méthode ne fournit aucun résultat, alors typeRetour est remplacé par le mot clé **void**.

**Remarque:**

Dans le cas où la méthode retourne une valeur, la valeur retournée par la fonction doit être spécifiée dans le corps de la méthode par l'instruction de retour:

return expression;

ou

return; // (possible uniquement pour une fonction de type void)

---

El Mostafa DAOUDI- p. 33

**2. Passage des paramètres**

Le mode de passage des paramètres dans les méthodes dépend de la nature des paramètres :

- **par valeur** pour les types primitifs.
- **par valeur des références** pour les objets: la référence est passée par valeur (i.e. le paramètre est une copie de la référence), **mais le contenu de l'objet référencé peut être modifié par la fonction** (car la copie de référence pointe vers le même objet...) :

---

El Mostafa DAOUDI- p. 34

### Exemple:

```
class ClasseTest {  
    void methode1(int j){  
        j+=12;  
    }  
    void methode2() {  
        int j = 3;  
        methode1(j);  
        System.out.println(j=" + j); // j=3  
    }  
}
```

---

El Mostafa DAOUDI- p. 35

### 3. Surcharge des méthodes

- On parle de surcharge (en anglais overload) lorsqu'un même symbole possède plusieurs significations différentes entre lesquelles on choisit en fonction du contexte. Par exemple la symbole « + » dans l'instruction « a+b » dépend du type des variables a et b.
- En Java , on peut surcharger *une méthode*, c'est-à-dire, définir (dans la même classe) plusieurs méthodes qui ont le même nom mais pas la même signature (différenciées par le type des arguments)

- **Remarques:**

- La signature d'une méthode est: le nom de la méthode et l'ensemble des types de ses paramètres.
- En Java, le type de la valeur de retour de la méthode ne fait pas partie de sa signature.

---

El Mostafa DAOUDI- p. 36

Considérons l'exemple suivant ou la classe **ClasseTest** est dotée de 2 méthodes calcMoyenne:

- La première a deux arguments de type double
- La deuxième a trois arguments de type double

---

El Mostafa DAOUDI- p. 37

```
class ClasseTest{
    public double calcMoyenne(double m1, float m2) {
        return (m1+m2)/2;
    }
    public float calcMoyenne(double m1, double m2, double m3) {
        return (m1+m2+m3)/3;
    }
}

public class TestSurcharge {
    public static void main(String[] args) {
        ClasseTest objA;
        System.out.println("Moy1="+objA.calcMoyenne(10., 12.2));
        System.out.println("Moy2="+objA.calcMoyenne(10., 12.6, 8.));
    }
}
```

---

El Mostafa DAOUDI- p. 38

## V. Accès aux membres d'un objet

### 1. Accès aux attributs(données membres)

Considérons la classe `ClassTest` suivante:

```
class ClasseTest {  
    double x;  
    boolean b;  
}
```

```
ClasseTest objA = new ClasseTest();  
// cette instruction a créé un objet de référence objA  
// on dit aussi que objA est une instance de la classe ClassTest  
// ou tout simplement on a créé un objet : objA
```

```
ClasseTest objB = new ClasseTest();
```

---

El Mostafa DAOUDI- p. 39

Une fois l'objet est créé, on peut accéder à ses données membres de la manière suivante: on indique le nom de l'objet suivi par un point, suivi par le nom de l'attribut comme suit:

*nomObjet.nomAttribut*

où:

*nomObjet* = nom de la référence à l'objet (nom de l'objet)

*nomAttribut* = nom de la donnée membre (nom de l'attribut).

**Exemple:** pour les objets `objA` et `objB` de la classe `ClassTest` qui sont déjà créés:

`objA.b=true;` // affecte true à l'attribut b de l'objet `objA`.

`objA.x=2.3;` // affecte le réel 2.3 à l'attribut x de l'objet `objA`.

`objB.b=false;` // affecte false à l'attribut b de l'objet `objB`.

`objB.x=0.35;` // affecte le réel 0.35 à l'attribut x de l'objet `objB`.

---

El Mostafa DAOUDI- p. 40

## 2. Accès aux méthodes: appels des méthodes

- Les méthodes ne peuvent être définies que comme des composante d'une classe.
- Une méthode ne peut être appelée que pour un objet.
- L'appel d'une méthode pour un objet se réalise de la manière suivante: on indique le nom de l'objet suivi d'un point, suivi du nom de la méthode et de sa liste d'arguments:

*nomObjet.nomMethode(arg1, ....).*

où

*nomObjet*: nom de l'objet

*nomMethode*: nom de la méthode.

---

El Mostafa DAOUDI- p. 41

**Exemple 1:** soit f () une méthode qui prend un paramètre de type double et qui retourne une valeur de type int.

```
class ClasseTest {  
    // attributs  
    int f(double x) {  
        int n;  
        // corps de la fonction f()  
        return n;  
    }  
}
```

```
ClasseTest objA = new ClasseTest(); // créé un objet objA;  
int j = objA.f(5.3); // affecte à j la valeur retournée par f()
```

---

El Mostafa DAOUDI- p. 42

**Exemple 2:** Considérons la classe Rectangle dotée de la méthode initialise qui permet d'affecter des valeurs à l'origine (aux attributs x et y).

```
public class Rectangle{
    int longueur, largeur;
    int x,y;
    void initialise_origine(int x0, int y0) {
        x=x0; y=y0;
    }
    public static void main (String args[]) {
        Rectangle r1;
        r1.longueur=4; r1.largeur=2;
        r1.initialise_origine(0,0); // affecte 0 aux attribut x et y
        // Affiche l'erreur : NullPointerException
        /* En effet l'objet r1 est seulement déclaré. Il n'est pas encore créé.
           Avant de l'utiliser, il faut tout d'abord le créer */
    }
}
```

---

El Mostafa DAOUDI- p. 43

**Exemple :** crée à l'origine un rectangle r1 de longueur 4 et de largeur 2

```
public class Rectangle{
    int longueur, largeur;
    int x,y;
    void initialise_origine(int x0, int y0) {
        x=x0; y=y0;
    }
    public static void main (String args[]) {
        Rectangle r1=new Rectangle ();
        r1.longueur=4; r1.largeur=2;
        r1.initialise_origine(0,0);
    }
}
```

---

El Mostafa DAOUDI- p. 44

## VI. Encapsulation

- Une classe permet d'envelopper les objets : Un objet est vu par le reste du programme comme une entité opaque.
- L'enveloppement [*wrapping*] des attributs et méthodes à l'intérieur des classes plus (+) le contrôle d'accès aux membres de l'objet est appelé *encapsulation*.
- Le contrôle d'accès aux membres de l'objet est appelé *cacher l'implémentation*: Les membres **publics sont vus de l'extérieur** mais les membres **privés sont cachés**.
- L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés.

---

El Mostafa DAOUDI- p. 45

- Possibilité d'accéder aux attributs d'une classe Java mais ceci n'est pas recommandé car contraire au principe d'encapsulation: Les données (attributs) doivent être protégés.
- Accessibles pour l'extérieur par des méthodes particulières (par exemple sélecteurs).
- Plusieurs niveaux de visibilité peuvent être définis en précédant, la déclaration d'un attribut, d'une méthode ou d'un constructeur, par un modificateur (spécificateur d'accès) : **private**, **public** ou **protected**.

---

El Mostafa DAOUDI- p. 46

### **Spécificateurs d'accès:**

L'encapsulation permet de définir 3 niveaux de visibilité des éléments de la classe. Ces 3 niveaux de visibilité (**public**, **private** et **protected**) définissent les droits d'accès aux données suivant le type d'accès:

- **Membres privés** : un membre d'une classe C dont la déclaration est précédée par le modificateur **private** est considéré comme un membre privé. Il n'est accessible que depuis l'intérieur. C'est à dire seulement les méthode de la classe elle-même (classe C) qui ont le droit d'accéder à ce membre.
- **Accès par défaut**: lorsqu'aucun spécificateur d'accès n'est mentionné devant un membre de la classe C, on dit que ce membre a *l'accessibilité par défaut*. Ce membre *n'est accessible que depuis la classe C et depuis les autres classes du paquet auquel C appartient (classes du même paquetage que la classe C)*.

---

El Mostafa DAOUDI- p. 47

- **Membres protégés** : un membre d'une classe C dont la déclaration est précédé par le modificateur **protected** se comporte comme **private** avec moins de restriction: il n'est accessible que depuis:
  - la classe C,
  - les classes du paquet auquel C appartient (les classes du même paquetage que la classe C)
  - et les sous-classes, directes ou indirectes, de C.

**Attention:** une classe dérivée a un accès aux membres **protected** mais pas aux membres **private**.

- **Membres publics** : un membre de la classe C dont la déclaration commence par le modificateur **public** est accessible partout où C est accessible. Il peut être accéder depuis l'extérieure (par une classe quelconque).

---

El Mostafa DAOUDI- p. 48



### **Exemple 1:** accès pour modification depuis l'extérieur d'un champs

```
class ClasseTest {
    public int x;
    private int y;
    public void initialise (int i, int j){
        x=i;
        y=j; // ok: accès depuis l'intérieur (depuis la classe elle-même) à l'attribut private y
    }
}

public class TestA{ // fichier source de nom TestA.java
    public static void main (String args[]) {
        ClasseTest objA;
        objA=new ClasseTest(); // On peut aussi déclarer ClasseTest objA=new ClasseTest();
        objA.initialise(1,3); /* ok: accès depuis l'extérieur (depuis une autre classe) à une
                               méthode public. Après appel de initialise(), x vaut 1 et y vaut 3 */

        objA.x=2; // ok: accès depuis l'extérieur à l'attribut public x. la valeur de x devienne 2

        objA.y=3; // ne compile pas car accès depuis l'extérieur à un attribut privé: private y
    }
}
```

El Mostafa DAOUDI- p. 49

### **Exemple 2:** affichage depuis l'extérieur d'un champs.

```
class ClasseTest {
    public int x;
    private int y;
    public void initialise (int i, int j){
        x=i;
        y=j; // ok: accès à l'attribut private y depuis l'intérieur
    }
}

public class TestA{ // fichier source de nom TestA.java
    public static void main (String args[]) {
        ClasseTest objA = new ClasseTest();
        objA.initialise(1,3); // ok a initialise() depuis l'extérieur car elle est public.
        System.out.println(" x= "+objA.x); /* affiche x = 1 car on peut accéder
                                             à x depuis l'extérieur: x est attribut public */
        System.out.println(" y= "+objA.y);
        // ne compile pas car on ne peut pas accéder à y qui est un attribut privé.
    }
}
```

El Mostafa DAOUDI- p. 50

## VII. Méthodes d'accès aux valeurs des variables depuis l'extérieur

Comment peut on accéder à la valeur d'une variable protégée ??

Considérons la classe etudiant qui a les champs nom et prenom private.

### Exemple:

```
class Etudiant {
    private String nom, prenom;
    public Etudiant(String st1, String st2){
        nom=st1; prenom=st2;
    }
}
public class MethodeStatic{
    public static void main(String[] argv) {
        Etudiant e= new Etudiant("Mohammed","Ali");
        System.out.println("Nom = "+e.nom);
        // ne compile pas car le champs nom est privé
        /* comment faire pour afficher le nom de l'étudiant e; ??
    }
}
```

---

El Mostafa DAOUDI- p. 51

Pour afficher la valeur de l'attribut nom (champs private), on définit Un accesseur (méthode getNom) qui est une méthode permettant de lire, depuis l'extérieur, le contenu d'une donnée membre protégée.

### Exemple:

```
class Etudiant {
    private String nom, prenom;
    public initialise(String nom, String Prenom){
        this.nom=nom; this.prenom=prenom;
    }
    public String getNom (){
        return nom;
    }
    public String getPrenom (){
        return prenom
    }
}
```

---

El Mostafa DAOUDI- p. 52

```

public class MethodeStatic{
    public static void main(String[] argv) {
        Etudiant e= new Etudiant("Mohammed","Ali");
        e.initialise("Mohammed","Ali");
        System.out.println("Nom = "+e.getNom());
        System.out.println("Prenom = "+e.getPrenom());
    }
}

```

---

El Mostafa DAOUDI- p. 53

Comment peut-on modifier (depuis l'extérieur) le contenu d'un attribut privé ?.

**Exemple:**

```

class Etudiant {
    private int cne;
}

```

```

public class MethodeStatic{
    public static void main(String[] argv) {
        Etudiant e= new Etudiant();
        // Modifier le champs cne (attribut private)
        e.cne=23541654;
        // ne compile pas car le champ cne est un champ protégé (private)
    }
}

```

---

El Mostafa DAOUDI- p. 54

Pour modifier la valeur de l'attribut nom (champs private), on définit **Un modificateur (mutateur)** qui est une méthode permettant de modifier le contenu d'une donnée membre protégée.

**Exemple:**

```
class Etudiant {
    private int cne;
    public void setCNE (int cne){
        this.cne=cne;
    }
}
public class MethodeStatic{
    public static void main(String[] argv) {
        Etudiant e= new Etudiant();
        e.setCNT(23541654);
    }
}
```

---

El Mostafa DAOUDI- p. 55

### **VIII. Autoréférences: emploi de this**

- Possibilité au sein d'une méthode de désigner *explicitement l'instance courante (l'objet courant)*: faire référence à l'objet qui a appelé cette méthode.
- Par exemple pour accéder aux attributs "masqués" les paramètres de la méthode.

```
class ClasseA {
    ....
    public void f(...) {
        ..... // ici l'emploi de this désigne la référence à l'objet
               // ayant appelé la méthode f
    }
}
```

---

El Mostafa DAOUDI- p. 56

**Exemple:**

```
class Etudiant {  
    private String nom, prenom;  
    public initialise(String st1, String st2) {  
        nom = st1;  
        prenom = st2;  
    }  
}
```

Comme les identificateurs st1 et st2 sont des arguments muets pour la méthode initialise(), alors on peut les noter nom et prenom qui n'ont aucune relation avec les champs *private* nom et prenom. Dans ce cas la classe Etudiant peut s'écrire comme suit:

---

El Mostafa DAOUDI- p. 57

```
class Etudiant {  
    private String nom, prenom;  
    public initialise(String nom, String prenom){  
        this.nom=nom;  
        this.prenom=prenom;  
    }  
}
```

---

El Mostafa DAOUDI- p. 58

## IX. Champs et méthodes de classe

### 1. Champs de classe (variable de classe)

Considérons la définition simpliste suivante:

```
class ClasseTest {  
    int n;  
    double x;  
}
```

Chaque objet de type **ClasseTest** possède ses propres valeurs (ses propres données) pour les champs n et x.

⇒ Les valeurs des champs ne sont pas partagées entre les objets.

**Exemple :** on crée deux objets différents instances de la classe **ClasseTest**:

```
ClasseTest objA1=new ClasseTest();
```

```
ClasseTest objA2= new ClasseTest();
```

objA1.n et objA2.n désignent deux données différentes.

objA1.x et objA2.x désignent aussi deux données différentes.

---

El Mostafa DAOUDI- p. 59

Certains attributs peuvent être partagés par toutes les instances d'une classe. C'est-à-dire ils peuvent être définis indépendamment des instances (objets):

Par exemple: le nombre d'étudiants = le nombre d'objets étudiants créés.

Les attributs qui peuvent être partagés entre tous les objets sont nommés champs de classe ou variables de classe. Ils sont comparables aux «variables globales ».

⇒ Ces variables n'existent qu'en un seul exemplaire. Elles sont définies comme les attributs mais précédés du mot-clé **static**.

---

El Mostafa DAOUDI- p. 60

**Exemple:** Considérons la classe:

```
class ClasseTest {  
    static int n; // la valeur de n est partagée par toutes les instances  
    double y;  
}
```

**ClasseTest** objA1=new **ClasseTest**(), objA2=new **ClasseTest**();

Ces déclarations créées deux objets (instances) différents: objA1 et objA2.

Puisque n est un attribut précédé du modificateur **static** , alors il est partagé par tous les objets en particulier les objets objA1 et objA2 partagent la même variable n.

⇒ objA1.n et objA2.n désignent la même donnée.

⇒ La valeur de l'attribut n est indépendante de l'instance (l'objet).

Pour accéder au champs statique n, on utilise le nom de la classe

**ClasseTest.n** // champs (statique) de la classe **ClasseTest**

---

El Mostafa DAOUDI- p. 61

**Exemple1:**

```
class ClasseTest {  
    int n;  
    double y;  
}  
public class MethodeStatic{  
    public static void main(String[] argv) {  
        ClasseTest objA1=new ClasseTest();  
  
        objA1.n+=4; // La valeur de l'attribut n de l'objet objA1 vaut 4;  
        ClasseTest objA2=new ClasseTest();  
        // objA2.n = ?  
        /* la valeur de l'attribut n de l'objet objA2 vaut 0.  
        initialisation par défauts des attributs. */  
    }  
}
```

---

El Mostafa DAOUDI- p. 62

### Exemple2:

```
class ClasseTest {  
    static int n; // la valeur de n est partagée par toutes les instances  
    float y;  
}  
public class MethodeStatic{  
    public static void main(String[] argv) {  
        ClasseTest objA1=new ClasseTest();  
        objA1.n+=4; // objA1.n vaut 4;  
        // équivalent à ClasseTest.n=4;  
        ClasseTest objA2=new ClasseTest();  
        // objA2.n = ?  
        // objA2 vaut 4 car champs statique .  
    }  
}
```

---

El Mostafa DAOUDI- p. 63

## 2. Méthodes de classe

- Ce sont des méthodes qui ont un rôle indépendant d'un objet spécifique. Elles exécutent une action indépendante d'une instance particulière de la classe
- La déclaration d'une méthode de classe se fait à l'aide du mot clé **static**.
- L'appelle d'une telle méthode ne nécessite que le nom de la classe correspondante.

**Important:** Une méthode de classe ne pourra pas accéder à des champs usuels (champs non statiques).

---

El Mostafa DAOUDI- p. 64



### Exemple:

```
class ClasseTest{  
    ...  
    private static int n; // champs de classe  
    private float x; // champs usuel  
    public static void f() { // méthode de classe  
        /* ici (dans le corps de la méthode f()), on ne pas accéder  
           au champs x, champs usuel  
           Par contre on peut accéder au champs statique n.  
        */  
    }  
}
```

---

El Mostafa DAOUDI- p. 65

### X. Le mot clé final

L'attribut *final* indique que la valeur de la variable ne peut être modifiée : on pourra lui donner une valeur une seule fois dans le programme.

#### Variable d'instance final

- Une variable d'instance *final* est une constante pour chaque objet.

#### Remarques:

- une variable d'instance *final* peut avoir 2 valeurs différentes pour 2 objets différents.
  - une variable d'instance *final* peut ne pas être initialisée à sa déclaration mais elle doit avoir une valeur à la sortie de tous les constructeurs.
- 

El Mostafa DAOUDI- p. 66

### Variable locale final

Considérons une variable locale précédé par le mot clé *final*.

- Si la variable est d'un type primitif, sa valeur ne pourra pas changer.
- Si la variable référence un objet, elle ne pourra pas référencer un autre objet mais l'état de l'objet peut être modifié

#### Exemple:

```
final Etudiant e = new Etudiant("Mohammed", "Ali");  
...  
e.nom = "Ahmed"; // ok, changement de l'état de l'objet e  
e.setCNE(32211452); // ok, changement de l'état de l'objet e  
e = new Etudiant("Ahmed"); /* Interdit car cette instruction  
indique que e référence à un autre objet */
```

---

El Mostafa DAOUDI- p. 67

### Constantes de classe

- Usage
  - Ce sont des variables de classes déclarées avec le mot-clé final
  - Ce sont des constantes liées à une classe
  - Elles sont écrites en MAJUSCULES
  - Pour y accéder, il faut utiliser non pas un identificateur d'objet mais le nom de la classe

#### Exemple:

```
public class Galerie {  
    public static final int MASSE_MAX = 150;  
}  
if (maVoiture.getWeightLimite() <= Galerie.MASSE_MAX) { ... }
```

---

El Mostafa DAOUDI- p. 68

## Ch. III. Les constructeurs

### I. Un exemple introductif

Considérons la classe point suivante: elle dispose des méthodes:

- *initialise* pour attribuer des valeurs aux coordonnées d'un points
- *déplace* pour modifier les coordonnées d'un point
- *affiche* pour afficher les coordonnées d'un point

---

El Mostafa DAOUDI- p. 69

```
public class Point{
    private int x; // abscisse
    private int y; // ordonnée

    public void initialise (int abs,int ord){
        x=abs;
        y=ord;
    }
    public void deplace (int dx, int dy){
        x+=dx;
        y+=dy;
    }
    public void affiche (){
        System.out.println(" abscisse : "+ x + " ordonnée : " + y);
    }
}
```

---

El Mostafa DAOUDI- p. 70

```

public class TstPoint{
    public static void main (String args[]) {
        Point pA=new Point(); // création de l'objet pA
        pA.initialise(1,1); // initialisation de l'objet pA
        pA.deplace(2,0);
        Point pB=new Point(); // création de l'objet pB
        pB.initialise(2,3); // initialisation de l'objet pB
        pB.affiche();
    }
}

```

---

El Mostafa DAOUDI- p. 71

### **Remarque:**

La création et l'initialisation des attributs sont séparées:

- Tout d'abord on crée l'objet (l'instance).
- Ensuite, dans notre exemple, pour chaque instance on appelle la méthode initialise() pour initialiser l'objet ainsi créer.
  - ⇒ On personnalise l'état de l'objet
- Si on n'appelle pas la méthode initialise, l'objet de type Point ainsi créer ne sera pas initialisé.
  - ⇒ Pas d'erreur à la compilation mais risque de problème sémantique : toute manipulation de l'objet, utilisera les valeurs de l'initialisation par défaut.

⇒ Créer un objet, ne garanti pas son initialisation

---

El Mostafa DAOUDI- p. 72

## **II. Définition de Constructeur**

- Le constructeur est une méthode spécial qui peut contenir des instructions à exécuter à la création des objets, en particulier : l'initialisation des attributs.
  - ⇒ La création et l'initialisation peuvent être unifiées
  - ⇒ Quand une classe possède un constructeur, Java l'appelle automatiquement à toute création d'objet avant qu'il ne puisse être utilisé.
- Chaque classe a un ou plusieurs constructeurs qui servent à
  - créer les instances
  - initialiser l'état de ces instances
- Règles de définition d'un constructeur : un constructeur:
  - porte le même nom que la classe.
  - n'a pas de type retour.
  - peut disposer d'un nombre quelconque d'arguments (éventuellement aucun).

---

El Mostafa DAOUDI- p. 73

**Exemple:** Considérons la classe point et transformons la méthode initialise en un constructeur.

```
public class Point{
    private int x; // abscisse: variable d'instance
    private int y; // ordonnée: variable d'instance

    public Point (int abs, int ord) { // Constructeur, remplace initialise
        x=abs;
        y=ord;
    }
    public void deplace (int dx, int dy){
        x+=dx;
        y+=dy;
    }
    public void affiche (){
        System.out.println(" abscisse : "+ x + " ordonnée : " + y);
    }
}
```

---

El Mostafa DAOUDI- p. 74

```

public class TestPoint{
    public static void main (String args[]) {
        Point pA=new Point(1,1); // création et initialisation de l'objet pA
        pA.deplace(2,0);
        Point pB=new Point(2,3); // création et initialisation de l'objet pB
        pB.affiche();
    }
}

```

**Remarque:**

- Les instructions: Point pA=new Point(); et pA.initialise(1,1) ; sont remplacées par: Point pA = new Point(1,1);

**Attention:** L'instruction:

Point pA = new Point();

ne convient plus car le constructeur a besoin de deux arguments

---

El Mostafa DAOUDI- p. 75

**Un autre exemple**

```

public class Etudiant {
    private String nom, prenom;           // Variables d'instance
    private int codeNatEtudiant;

    public Etudiant(String n, String p) { // définition d'un constructeur
        nom = n;
        prenom = p;
    }
    public void setCNE (int cne) {
        codeNatEtudiant = cne;
    }
    public static void main(String[] args) {
        Etudiant e;
        e = new Etudiant("Mohammed", "Ali"); // création d'une instance
        e.setCNE(23456765);
    }
}

```

---

El Mostafa DAOUDI- p. 76

### III. Quelques règles concernant les constructeurs

1. Aucun type, même void, ne doit figurer devant son nom.
2. Lorsque le code de la classe comporte un constructeur alors il doit être appelé à chaque création.
3. Lorsque le code d'une classe ne comporte pas de constructeur, un constructeur par défaut sera automatiquement ajouté par Java. Dans ce cas on peut créer des objets de la classe ClasseTest par l'instruction `ClasseTest c = new ClasseTest();`  
//ok si ClasseTest n'a pas de constructeur.
4. **Règle générale:** Si pour une classe ClasseA donnée, l'instruction `ClasseA a=new ClasseA();` est acceptée  
Cela signifie que:
  - Soit la classe ClasseA ne possède pas de constructeur
  - Soit la classe ClasseA possède un constructeur sans arguments.

---

El Mostafa DAOUDI- p. 77

5. Un constructeur ne peut pas être appelé de la même manière que les autres méthodes. Par exemple:  
`Point a=new Point(1,1);` // ok car Point possède un constructeur  
`a.Point (2,3);` // Interdit car Point() est un constructeur et  
// non une méthode classique.
6. Si un constructeur est déclaré `private`, dans ce cas il ne pourra plus être appelé de l'extérieur, c'est-à-dire il ne pourra pas être utilisé pour instancier des objets. **Exemple:**  

```
class ClasseA{
    private ClasseA() { ... } // constructeur privée sans arguments
    ...
}
...
ClasseA a=new ClasseA();
// erreur, car constructeur ClasseA() est privé
```

---

El Mostafa DAOUDI- p. 78

#### IV. Initialisation des attributs

On distingue 3 types d'initialisation des attributs:

1. Initialisation par défaut: les champs d'un objet sont toujours initialisés par défaut.

Type de l'attribut ( champ)	Valeur par défaut
boolean	false
char	caractère de code nul
int, byte, short, int long	0
float, double	0.f ou 0.
class	null

El Mostafa DAOUDI- p. 79

#### 2. Initialisation explicite des champs d'un objet:

Un attribut peut être initialisé pendant sa déclaration. Par exemple:

```
class ClasseTest{  
    private int n=10;  
    private int p;  
}
```

#### 3. Initialisation par le constructeur:

On peut aussi initialiser les attributs lors de l'exécution des instructions du corps du constructeur. Exemple du constructeur Point(int,int) voir avant:

```
Point pA=new Point(8,12);
```

El Mostafa DAOUDI- p. 80



D'une manière générale, la création d'un objet entraîne toujours, par ordre chronologique, les opérations suivantes:

- L'initialisation par défaut de tous les champs de l'objet.
- L'initialisation explicite lors de la déclaration du champ.
- L'exécution des instructions du corps du constructeur.

---

El Mostafa DAOUDI- p. 81

**Exemple:** Considérons la classe suivante:

```
class ClasseTest{  
    public ClasseTest (...) { ... } // Constructeur ClasseTest  
    ...  
    private int n=10;  
    private int p;  
}
```

L'instruction suivante:

```
ClasseTest objA=new ClasseTest(...);
```

Entraîne successivement:

1. initialisation implicite (par défaut) des champs n et p de l'objet *objA* à 0
2. initialisation explicite: On affecte au champ n la valeur 10 (la valeur figurant dans sa déclaration).
3. exécution des instruction du constructeur: Le corps du constructeur n'est exécuté qu'après l'initialisation par défaut et l'initialisation explicite.

---

El Mostafa DAOUDI- p. 82

```

class ClasseA {
    public ClasseA() { // ici n =20, p =10 et np = 0
        np=n*p;
        n=5;
    }
    private int n=20, p=10;
    private int np;
}
public class Init {
    public static void main (String args[]) {
        ClasseA objA=new ClasseA();
        // ici objA.n =5, objA.p = 10, mais objA.np = 200
    }
}

```

Après appel du constructeur on a: n=5 , p=10 , np = 200

---

El Mostafa DAOUDI- p. 83

### **V. Surcharge des constructeurs (Plusieurs constructeurs)**

Une classe peut avoir plusieurs constructeurs. Comme le nom du constructeur est identique au nom de la classe:  
 ⇒ Java permet la surcharge des constructeurs.

```

class Etudiant {
    private String nom, prenom; // Variables d'instance
    private int codeNatEtudiant;
    public Etudiant(String n, String p) { // Constructeur
        nom = n;
        prenom = p;
    }
    public Etudiant(String n, String p, int cne) { // Constructeur
        nom = n;
        prenom = p;
        codeNatEtudiant = cne;
    }
}

```

---

El Mostafa DAOUDI- p. 84

```
public class TestEtudiant {  
    public static void main (String[] args) {  
        Etudiant e1, e2;  
        e1 = new Etudiant("Mohammed", "Ali");  
        e2 = new Etudiant("Ouardi", " fatima", 22564321);  
    }  
}
```

---

El Mostafa DAOUDI- p. 85

## **VI. Utilisation du mot clé this dans un constructeurs**

Il est possible, qu'au sein d'un constructeur, on peut appeler un autre constructeur de la même classe. Pour faire on fait appel au mot clé **this** qui est utilisé cette fois comme nom de méthode.

Supposons que la classe Point admet deux constructeurs. Un constructeur initialise à l'origine (au point (0,0)) et un autre initialise à un point quelconque.

---

El Mostafa DAOUDI- p. 86

**Exemple:** La classe Etudiant peut être réécrite de la façon suivante:

```
class Etudiant {  
    private String nom, prenom; // Variables d'instance  
    private int codeNatEtudiant;  
    public Etudiant(String n, String p) { // Constructeur  
        nom = n;  
        prenom = p;  
    }  
    public Etudiant(String n, String p, int cne) { // Constructeur  
        this(n,p); // appel au constructeur Etudiant(String,String);  
        codeNatEtudiant = cne;  
    }  
}
```

---

El Mostafa DAOUDI- p. 87

## Ch. IV. Généralités sur la structure lexicale de Java

### I. Variables

- Les variables d'instances:
  - sont déclarées en dehors de toute méthode.
  - conservent l'état d'un objet, instance de la classe
  - sont accessibles et partagées par toutes les méthodes de la classe
- Les variables locales:
  - sont déclarées à l'intérieur d'une méthode
  - conservent une valeur utilisée pendant l'exécution du bloc dans lequel elles sont définies.
  - ne sont accessibles que dans le bloc dans lequel elles ont été déclarées.
- Déclaration des variables: En Java, toute variable utilisée doit être déclarée avant son utilisation.

---

El Mostafa DAOUDI- p. 88

La syntaxe de la déclaration :

*type identificateur;*

Où

- *identificateur* est une suite de caractères pour désigner les différentes entités manipulés par un programme: variables, méthode, classe, objet, ...
- *type* désigne le type de la variable déclarée

**Remarque:** concernant les identificateurs:

- Tous les caractères sont significatifs.
- On fait la différence entre les majuscules et les minuscules.
- Mots réservés: un identificateur ne peut pas correspondre à un mot réservé du langage (par exemple if, else, while, ..., ).

**Exemple:**

```
int val;  
double delta;  
Rectangle r1; /* Rectangle est une classe */
```

---

El Mostafa DAOUDI- p. 89

### **Mots réservé du langage Java**

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

---

El Mostafa DAOUDI- p. 90

## II. Style de programmation non standard

Conventions de nommage en Java: ne sont pas obligatoire, mais fortement recommandées (c'est essentiel pour la lisibilité du code et sa maintenance).

- **Identificateur**: pas d'emploi de \$ (et de caractères non ASCII)
- **Nom de classe** :
  - Si le nom de la classe est composé de plusieurs mots, ils sont accolés (on ne les sépare pas avec le trait \_ souligné).
  - Commencez chaque mot par une majuscule

**Exemple**: HelloWorld, ClasseTest, MonPremierProgramme.

---

El Mostafa DAOUDI- p. 91

- **Nom de variable ou méthode** :
  - Si le nom est composé d'un seul mot: la première lettre est minuscule.
  - Si le nom est composé par plusieurs mots, ils sont accolés et respecte la règle suivante:
    - La première lettre du premier mot est minuscule.
    - Les premières lettre des autres mots sont majuscules
- **Exemple**: maPremiereNote, initialiseAttribut(), calculTaux(), main(), objEtudiant, ...
- **Les constantes**: majuscules et séparation des mots par \_ :  
VALEUR\_MAX;
- **Usage non recommandé** :
  - mapremierevariable
  - ma\_premiere\_variable

---

El Mostafa DAOUDI- p. 92

### Les commentaires

- Sur une seule ligne : style C++  
`// Ceci un commentaire`  
`int taux = 75; // taux de réussite`
- Sur plusieurs lignes : style C  
`/* Première ligne du commentaire`  
`suite du commentaire */`

---

El Mostafa DAOUDI- p. 93

### Mise en page des programmes

La mise en page d'un programme Java est libre.

- Une instruction peut être étendue sur plusieurs lignes.
- Une ligne peut comporter plusieurs instructions

### Emploi du code Unicode:

Java utilise le codage Unicode, qui est basé sur 2 octets (16 bits), par conséquent on peut coder 65 536 symboles, ce qui permet de couvrir la plus part des symboles utilisés dans le monde. Pour assurer la portabilité, le compilateur commence par traduire le fichier source en Unicode.

- On peut, par exemple, utiliser l'identificateur élément: `int élément;`
- On peut aussi utiliser le code Unicode d'un caractère (mais à éviter pour la lisibilité) de la manière suivante: `\uxxxx` représente le caractère ayant comme code Unicode la valeur `xxxx`.  
Par exemple `\u0041` représente le caractère A.

---

El Mostafa DAOUDI- p. 94

### III. L'affectation

- L'opération d'affectation affecte ( assigne) une valeur à une variable. Elle est réalisée au moyen de l'opérateur « = ».

la syntaxe : *identificateur* = *valeur* ;

Signifie prendre la valeur du coté droit et la copier du coté gauche.

- *identificateur* : désigne une variable.
- *valeur* : est une constante, une variable ou une expression qui retourne une valeur du même type que la variable référencée par *identificateur* .

---

El Mostafa DAOUDI- p. 95

### Cas des types primitifs

- Soient a et b de type primitif. L'affectation a=b; signifie que le contenu de b est copié dans a.

#### Exemple:

a=2;

b=15;

a=b; // La nouvelle valeur de a est 15

⇒ après affectation a et b existent et ont tous la même valeur 15.

**Remarque:** si ensuite on modifie le contenu de a, alors le contenu de b ne sera pas modifié.

---

El Mostafa DAOUDI- p. 96



## Cas des Objets

Soient objA et objB deux objets,

L'affectation `objA=objB`; signifie qu'on copie la référence de `objB` dans `objA`.

⇒

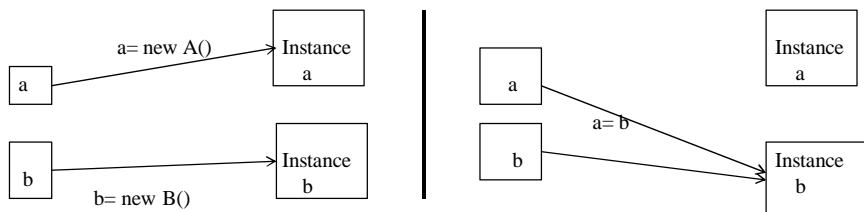
1. Après affectation, les références **objA** et **objB** pointent vers le même objet référencé par **objB**.
2. L'objet qui a été initialement référencé par **objA** existe toujours, mais on n'a aucun contrôle sur lui (il reste inaccessible).

---

El Mostafa DAOUDI- p. 97

### Exemple:

```
class A {  
    public int i;  
}  
public class TestAffectation {  
    public static void main(String[] args) {  
        A a=new A();  A b=new A();  
        a.i=6;        b.i=11;  
        a=b;          // a et b contiennent la même référence, on a: a.i=11  
        a.i=20;        // b.i = 20  
        b.i=13;        // a.i = 13  
    }  
}
```



---

El Mostafa DAOUDI- p. 98

## IV. Types de données en Java

2 groupes de types de données sont manipulés par Java:

- **types primitifs**
- **objets (instances de classe)**

---

El Mostafa DAOUDI- p. 99

### 1. Types primitifs

- Booléen:  
**Boolean** : prend deux valeurs **true** ou **false**
- Caractère (un seul caractère) :  
**char** : codé sur 2 octet par le codage Unicode
- Nombres entiers :  
**byte**: représenté sur 1 octet (8 bits),  
**short**: représenté sur 2 octets (16 bits),  
**int** : représenté sur 4 octets (32 bits),  
**long** : représenté sur 8 octets (64 bits).
- Nombres à virgule flottante :  
**float** : flottant (IEEE 754) 32-bit  
**double** : flottant (IEEE 754) 64-bit

---

El Mostafa DAOUDI- p. 100

## 1.1. Type booléen

- Sert à représenter une valeur logique du type vrai/faux:
  - Une variable de type booléen est déclaré par:  
*boolean* b;  
b prend une des deux valeurs *true* ou *false*
- C'est un véritable type:
  - Il est retourné par les opérateurs de comparaison
  - Il est attendu dans tous les tests
- ne peut pas être converti en entier

---

El Mostafa DAOUDI- p. 101

## 1.2. Type caractère

- Permet de manipuler des caractères: Java représente un caractère sur 2 octets (16 bits)  
16-bit => 65536 valeurs : presque tous les caractères de toutes les écritures.  
Ne sont affichables que si le système possède les polices de caractères adéquates.
- Une variable de type caractère est déclarée par: *char* c1,c2;
- Constantes de type caractère doivent être écrites entre simples quotes .  
**Exemple :** 'a' , 'Z' , 'E'

Les caractères disposant d'une notation spéciale.

Notation	Code Unicode	Abréviation usuelle	Signification
\b	0008	BS(Back Space)	Retour arrière
\t	0009	HT(Horizontal Tabulation)	Tabulation horizontale
\n	000a	LF(line Feed)	Saut de ligne
\f	000c	FF(Form Feed)	Saut de page
\r	000d	CR(Cariage Return)	Retour chariot
\"	0022		
\'	0027		
\\	005c		

---

El Mostafa DAOUDI- p. 102

### 1.3. Nombres entiers

- Les valeurs min/max

- **byte** : codé sur sur 1 octet = 8 bits
- **short** : compris entre -32 768 et 32 767
- **int** : compris entre -2.147 483 648 et 2 147 483 647
- **long** : compris entre -923 372 036 854 775 808  
et 923 372 036 854 775 807

---

El Mostafa DAOUDI- p. 103

- Constantes nombres: Une constante « entière » est de type **int**. Si elle est suffixée par « L » ou « l » alors elle est de type **long**.

Exemple:

108 // 108 constante de type int, représenté sur 32 bits

108L // 108 constante de type long, représenté sur 64 bits

- Représentation en octal (8bits) et hexadécimale (16 bits).

Exemple 1:

014 // 14 en octal (base 8) = 12 en décimal

0xA7 // A7 en hexadécimal (base 16) = 167 en décimal

- Conversion automatique (implicite) seulement vers les types entiers plus grands (taille plus grande).  
(int → long, short → int, byte → short) et vers les types flottants.

---

El Mostafa DAOUDI- p. 104

### Types des résultats des calculs

Avec des nombres entiers

- Tout calcul entre entiers donne un résultat de type **int**
- si au moins un des opérandes est de type **long**, alors le résultat est de type **long**

#### Exemple:

```
byte b1 = (byte)20;
```

```
byte b2 = (byte)15;
```

```
byte b3 = b1 + b2;
```

Provoquera une erreur à la compilation car :

(b1 + b2) est de type **int** (codé sur 32 bits) alors que b3 est de type **byte** (codé sur 8 bits seulement).

---

El Mostafa DAOUDI- p. 105

### 1.4. Les nombres flottants

- **float** : environ 7 chiffres significatifs ;
  - Valeur absolue (arrondie) maximal  $3,4 \times 10^{38}$  ( constante prédéfinie *Float.MAX\_VALUE*)
  - Valeur absolue (arrondie) minimal  $1,4 \times 10^{-45}$  (constante prédéfinie *Float.MIN\_VALUE*)
- **double** : environ 15 chiffres significatifs ;
  - Valeur absolue maximal (arrondie)  $1,8 \times 10^{308}$  (constante prédéfinie *Double.MAX\_VALUE*)
  - Valeur absolue minimal (arrondie)  $4,9 \times 10^{-324}$  (constante prédéfinie *Double.MIN\_VALUE*)

---

El Mostafa DAOUDI- p. 106

### Constantes nombres

- Une constante réelle est par défaut elle est de type **double**.
- Pour spécifier un **float**, il faut la suffixée par « F » ou « f ».
- Conversion automatique (implicite) : seulement **float** → **double**

### Exemple:

```
.567e2           // 56,7 de type double  
5.123E-2F       // 0,05123 de type float  
float x=2.5f    //ok  
double y = 2.5; //ok
```

```
float x = 2.5; // Erreur: conversion double à float
```

---

El Mostafa DAOUDI- p. 107

## 2. Type objet: Constante null

- **null** : valeur d'une "référence vers rien" (pour tous types de références).
- Elle indique qu'une variable de type non primitif ne référence rien) ; convient pour *tous les types non primitifs*.

---

El Mostafa DAOUDI- p. 108

### **3. Transtypage**

- Java est un langage fortement typé
- Dans certains cas, il est nécessaire de forcer le programme à changer le type d'une expression. On utilise pour cela le *cast* (transtypage) :

*(type-forcé) expression*

#### **Exemple:**

```
int i=13;    // i codé sur 32 bit; 13 constante , par défaut de type int
byte b=i;    // Erreur: pas de conversion implicite int →byte
```

Pour que ça marche, on doit faire un caste (un transtypage) c'est-à-dire on doit forcer le programme à changer le type de *int* en *byte*.

```
byte b=(byte)i; // de 32 bits vers 8 bit. b vaut 13 codé sur 8 bits
```

---

El Mostafa DAOUDI- p. 109

### ***Casts autorisés***

- En Java, deux seuls cas sont autorisés pour les *casts* :
  - entre types primitifs,
  - entre classes mère/ancêtre et classes filles.

---

El Mostafa DAOUDI- p. 110

### 3.1. Casts entre types primitifs

Un *cast* entre types primitifs peut occasionner une perte de données sans aucun avertissement ni message d'erreur.

- `int i=13; // i codé sur 32 bit; 13 codé sur 32 bits`  
`byte b=(byte)i;`

`// b vaut 13 car conversion de 32 bits vers 8 bit (On considère les 8 bits de poids le plus faible). i est un entier « petit »`

- `int i=256; // i codé sur 32 bit; 256 codé sur 32 bits`  
`byte b=(byte)i; // b = 0 ! Pourquoi ???`  
`// b vaut 0 car conversion d'un int vers un byte (de 32 bits vers 8 bits : on considère les 8 bits de poids le plus faible). i est un entier « grand »`

`int i = 130;`  
`b = (byte)i; // b = -126 ! Pourquoi ???`

---

El Mostafa DAOUDI- p. 111

### Transtypage implicite et explicite:

Une affectation types primitifs peut utiliser un *cast* implicite si elle ne provoque aucune perte

#### Exemple:

`byte =120; // cast implicite`

Ne provoque pas d'erreur car 120 est dans max/min pour les `byte`.

`byte b=130; // Provoque une erreur "cannot convert from int to byte" car 130 est hors max/min. Dans ce cas le cast explicite est obligatoire. Mais attention au perte d'information:`

`byte b=(byte)130; // cast explicite. Ca passe à la compilation mais provoque une erreur de calcul: renvoie -126.`

- Pour une affectation non statique, le *cast* est obligatoire :

`int i = 120;`

`byte b=i; // Provoque une erreur même si la valeur de i est dans l'intervalle max/min des byte. On doit faire un cast explicite:`

`byte b = (byte)i ; // cast obligatoire, sinon erreur`

---

El Mostafa DAOUDI- p. 112



- Les casts de types « flottants » vers les types entiers tronquent les nombres :

```
float x=1.99F;
```

```
int i = (int)x; /* i = 1, et pas 2. On fait une troncature et non un arrondi. */
```

**Attention:** la conversion peut donner un résultat totalement faux sans aucun avertissement ni message d'erreur :

```
int c = (int)1e+30; // c = 2147483647 !
```

```
int x = 10, y = 3;
```

```
double z;
```

```
z=x/y; // donne z=3.0; car x/y donne 3
```

si on veut que z=3.3333.. et pas 3.0, on doit écrire:

```
z = (double)x / y; // cast de x suffit
```

---

El Mostafa DAOUDI- p. 113

- De même, on peut affecter un entier à une variable de type nombre à virgule flottante

```
float x; int i;
```

```
x=i;
```

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

---

El Mostafa DAOUDI- p. 114

### 3.2. Casts entre entiers et caractères

- Ils font correspondre un entier à un caractère qui a comme code Unicode la valeur de l'entier
- La correspondance **char** → **int**, **long** s'obtient par *cast* implicite
- Le code d'un **char** peut aller de 0 à 65 535 donc **char** → **short**, **byte** nécessite un *cast* explicite (**short** ne va que jusqu'à 32 767)
- Les entiers sont signés et pas les **char** donc **long**, **int**, **short** ou **byte** → **char** nécessite un *cast* explicite

El Mostafa DAOUDI- p. 115

### Tableau récapitulatif de conversion de types

Conversion de	vers							
	boolean	byte	short	char	int	long	float	double
boolean		N	N	N	N	N	N	N
byte	N		O	C	O	O	O	O
short	N	C		C	O	O	O	O
char	N	C	C		O	O	O	O
int	N	C	C	C		O	O*	O
long	N	C	C	C	C		O*	O*
float	N	C	C	C	C	C		O
double	N	C	C	C	C	C	C	

N pas de conversion possible

O conversion automatique

C nécessite un transtypage (cast)

O\* conversion large (risque de pertes de précision)

El Mostafa DAOUDI- p. 116

### Exemple avec la surcharge des méthode

```
class Point {
    private int x,y;
    public void Point (int x, int y){ this.x=x; this.y=y; }
    public void deplace (int dx, int dy) { x+=dx; y+=dy; }
    public void deplace (int dx) { x+=dx; }
    public void deplace (short dx) { x+=dx; }
}

public class Surcharge {
    public static void main(String arg[]) {
        Point a=new Point(1,2);
        a.deplace(1,3); // appel deplace (int,int)

        a.deplace(2); // appel deplace (int)

        short p=3; // ok car 3 est dans le max/min de short
        a.deplace(p); // appel deplace (short)

        byte b=2; /// ok car 2 est dans le max/min de byte

        a.deplace(b); // appel deplace(short) après conversion de b en short
    }
}
```

El Mostafa DAOUDI- p. 117

**Attention: Il peut y avoir ambiguïté.** Supposons que la classe contient les deux méthodes deplace () suivantes:

```
public void deplace(int dx, byte dy) { x+=dx; y+=dy; }
public void deplace(byte dx, int dy) { x+=dx; }
```

Soit la classe exécutable suivante:

```
public class Surcharge {
    public static void main(String arg[]) {
        Point a =new Point(2,4);
        int n; byte b;

        a.deplace(n,b); // ok appel de deplace (int,byte)
        a.deplace(b,n); // ok appel de deplace (byte,n)
        a.deplace(b,b); // erreur de compilation, ambiguïté
    }
}
```

El Mostafa DAOUDI- p. 118

## V. Principaux opérateurs

- affectation : =
- arithmétiques : + - \* / %
- comparaisons : < <= > >= == !=
- booléens : && || ! ^
- opérations bit-à-bit (sur les entiers) :  
& | ^ ~ << >> >>>
- opération et affectation simultanées :  
+= -= \*= /= %= &= |=  
^= <<= >>= >>>=
- pré/post-incrémentation : ++  
pré/post-décrémentation : --
- opérateur ternaire : ?:
- création tableau ou objet (allocation mémoire) :  
new
- test de type des références : instanceof

---

El Mostafa DAOUDI- p. 119

## VI. Notion de portée et de durée de vie des objets

- Un bloc est un ensemble d'instructions délimité par les accolades { et }
- Les blocs peuvent être emboîtés les uns dans les autres.
- On peut grouper plusieurs instructions en un bloc délimité par des accolades { et }. Le concept de portée fixe simultanément la visibilité et la durée de vie des noms définis dans cette portée.

### Exemple 1:

```
int a=1, b=2;  
{ //début de bloc  
    int x=a;  
    x = 2*a+b;  
} //fin de bloc
```

```
x = 3; //ERREUR: x n'est pas connu ici
```

---

El Mostafa DAOUDI- p. 120

**Remarque :** La portée ( zone de validité de la déclaration)  
d'une variable va de sa déclaration jusqu'à la fin du bloc où  
elle est déclarée. Elle est fixée par les accolades { }.

**Exemple 2:**

```
{  
    int x=12; // x est accessible  
    {  
        int q;  
        q=x+100; // x et q tous les deux sont accessibles  
    }  
    x=6;  
        // x est accessible  
    q=x+2;  
        // Erreur: q est hors de portée  
}
```

---

El Mostafa DAOUDI- p. 121

**Attention:** Ceci n'est pas permis en Java

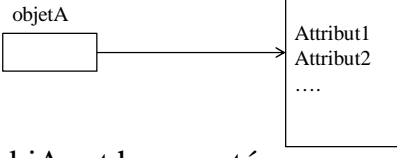
```
{  
    int x=12;  
    {  
        int x=96; // illégale en Java, valable en C, C++  
    }  
}
```

---

El Mostafa DAOUDI- p. 122

**Portée des objet :** Considérons une classe nommée ClasseA et qui a une méthode publique nommée f().

```
{  
    ClasseA objA=new ClasseA();  
} // fin de portée  
objA.f(); // Erreur car la référence objA est hors portée.
```



The diagram illustrates the scope issue. A small box labeled 'objetA' has an arrow pointing to a larger box representing the 'ClasseA' object. The 'ClasseA' box contains the text 'Attribut1', 'Attribut2', and '....'.

**Remarque:**

- La référence ObjA disparaît à la fin de la portée,
  - Par contre l'objet qui a été référencé par **objA** existe toujours, **mais on n'a aucun contrôle sur lui (il reste inaccessible)**.
- ⇒ Les objets n'ont pas la même durée de vie que les types primitifs.

---

El Mostafa DAOUDI- p. 123

**Gestion de la mémoire (Garbage Collector)**

Lorsqu'on perd le contrôle sur un objet. Il persiste et il occupe de la mémoire.

- Il n'existe aucun opérateur explicite pour détruire l'objet dont on n'a pas besoin,
- Mais il existe un mécanisme de gestion automatique de la mémoire, connu sous le nom de ramasse miettes (en anglais Garbage Collector). Son principe est le suivant:
  - A tout instant, on connaît le nombre de références à un objet donné.
  - Lorsqu'il n'existe plus aucune référence sur un objet, on est certains que le programme ne peut pas y accéder, donc l'objet devient candidat au ramasse miette.

---

El Mostafa DAOUDI- p. 124

Le ramasse-miettes (*garbage collector*) est une tâche qui

- travaille en arrière-plan
  - libère la place occupée par les instances non référencées
  - compacte la mémoire occupée
- 
- Il intervient
    - quand le système a besoin de mémoire
    - ou, de temps en temps, avec une priorité faible

---

El Mostafa DAOUDI- p. 125

## **VII. Instructions de contrôle**

### **1. Structure alternative « if... else »**

Soit `exprBool` une *expression Booléenne* qui retourne *true* (Vrai) ou *false* (faux)

```
if (exprBool) {  
    instructions; // exécutées si exprBool retourne true  
}  
  
if (exprBool) {  
    instructions1 ; // exécutées si exprBool retourne true  
}  
else {  
    instructions2; // exécutées si exprBool retourne false  
}
```

---

El Mostafa DAOUDI- p. 126

```

if (exprBool1) {
    instructions ;    // Exécutées si exprBool1 retourne true
}
else if (exprBool2) {
    instruction;
    // Exécutées si exprBool1 retourne false et exprBool2 retourne true.
}
else {
    // ...
}

```

**Remarque:** Un bloc serait préférable, même s'il n'y a qu'une seule instruction.

---

El Mostafa DAOUDI- p. 127

## **2. Expression conditionnelle**

*L'expression:*

*expressionBooléenne ? expression1 : expression2*

Est équivalente à:

```

if (expressionBooléenne) {
    expression1;
}
else {
    expression2;
}

```

---

El Mostafa DAOUDI- p. 128



### Exemple:

```
int x, y;  
if (x % 2 == 0)  
    y = x + 1;  
else  
    y = x;
```

Est équivalent à

```
int x;  
int y = (x % 2 == 0) ? x + 1 : x;
```

---

El Mostafa DAOUDI- p. 129

### 3. Cas multiple: Distinction de cas suivant une valeur

```
switch(expression) {  
    case val1: instructions; // exécutées si expression ==val1  
        break; // Attention, sans break, les instructions du cas suivant sont exécutées !  
    case val2: instructions; // exécutées si expression ==val2  
        break;  
    ...  
    case valn: instructions; // exécutées si expression ==valn  
        break;  
    default: instructions; // exécutées si aucune des valeurs prévues  
        break;  
}
```

- *Expression* est de type **char**, **byte**, **short**, ou **int**, ou de type énumération (ou type **énuméré défini avec enum**).
- S'il n'y a pas de clause **default**, rien n'est exécuté si *expression* ne correspond à aucun **case** (aucune valeur prévue).

---

El Mostafa DAOUDI- p. 130

#### **4. Boucles de répétitions**

Deux types de boucles :

- Répétitions « tant que »

```
while (expressionBooléenne) {  
    instructions;    // corps de de la boucle  
}
```

- Répétition « faire tant que »: le cors de la boucle est *exécuté au moins une fois*.

```
do {  
    instructions;    // corps de de la boucle  
} while (expressionBooléenne);
```

---

El Mostafa DAOUDI- p. 131

#### **5. Boucle d'itération: Répétition for**

```
for(init; test; incrément){  
    instructions; // corps de la boucle  
}
```

- **init** (initialisation): Déclaration et/ou affectations, séparées par des virgules
- **incréments** : expressions séparées par des virgules
- **Test**: expression booléenne

**Remarque:** Dans le cas où le corps de la boucle est formée d'une seule instruction pas besoin de mettre les accolades { } pour marquer le début et la fin du bloc

Elle est équivalente à

```
init;  
while (test) {  
    instructions;  
    incrément;  
}
```

---

El Mostafa DAOUDI- p. 132

## 6. Interruptions des boucles

- **Break** : sortie de la boucle : sort de la boucle et continue l'exécution des instructions se trouvant après la boucle.
- **continue** : passage à l'itération suivante: interrompt l'itération en cours et passe à l'itération suivante
- **break et continue** peuvent être suivis d'un nom d'étiquette qui désigne une boucle englobant la boucle où elles se trouvent (une étiquette ne peut se trouver que devant une boucle)

---

El Mostafa DAOUDI- p. 133

## Etiquette de boucles

```
etiq1: while (pasFini) { // etiq1: étiquette
...
for (int i=0; i < n; i++) {
    ...
    if (t[i] < 0)
        continue etiq1; // réexécuter à partir de etiq1
    ...
}
...
}
```

---

El Mostafa DAOUDI- p. 134

## **VIII. Entrées Sorties Standards**

### **1. Affichage standard (Ecriture sur la sortie standard)**

- L'affichage standard à l'écran est effectué à l'aide des méthodes `print()` et `println()` de la classe `PrintStream` (la classe de l'objet `System.out`).  
`print()` ne fait pas retour à la ligne .  
`println()` fait retour à la ligne suivante.

Dans le cas où:

- tous les arguments sont des types primitifs ou `String`:  
les types primitifs sont convertis vers le type `String` par l'appel des méthodes statiques `valueOf()` de la classe `String`.
- un argument est un objet, alors l'objet sera converti en un type `String` par l'appel de la méthode `toString()`.

---

El Mostafa DAOUDI- p. 135

### **Exemple 1:**

```
float z=1.732f;  
System.out.print("z=");  
System.out.println(z);  
System.out.println(" et 2z= " + (2*z));
```

### **Affiche:**

```
z=1.732  
et 2z= 3.464
```

### **Exemple 2:**

Soit « obj » est un objet, l'expression

```
System.out.println("résultat: " + obj) ;
```

est équivalente à l'expression:

```
System.out.println ("résultat: " + obj.toString());
```

---

El Mostafa DAOUDI- p. 136

## **2. Lecture (Saisi à partir du clavier: La classe Scanner:**

Les méthodes de la classe Scanner les plus intéressantes sont certainement :

- `byte nextByte()` : Lecture d'une valeur de type `byte`
- `short nextShort()` : Lecture d'une valeur de type `short`.
- `int nextInt()` : Lecture d'une valeur de type `int`.
- `long nextLong()` : Lecture d'une valeur de type `long`.
- `float nextFloat()` : Lecture d'une valeur de type `float`.
- `double nextDouble()` : Lecture d'une valeur de type `double`.
- `String nextLine()` : Lecture d'une chaîne jusqu'à une marque de fin de ligne. La marque de fin de ligne n'est pas incorporée à la chaîne produite.

---

El Mostafa DAOUDI- p. 137

### **Exemple:**

```
import java.util.Scanner;
public class TestScanner {
    public static void main(String[] args) {
        Scanner entree = new Scanner(System.in);
        System.out.print("nom et prénom? ");
        String nom = entree.nextLine();
        System.out.print("âge? ");
        int age = entree.nextInt();
        System.out.print("taille (en m)? ");
        float taille = entree.nextFloat();
        System.out.println("lu: "+nom+", "+age+" ans, "+taille+" m");
    }
}
```

---

El Mostafa DAOUDI- p. 138

## Chp. V. Les tableaux

### I. Introduction

- En Java
  - Les tableaux sont considérés comme des objets. Ils sont manipulés par des références.
  - les variables de type tableau contiennent des références aux tableaux.
  - Les éléments du tableau peuvent être d'un type primitif ou d'un type objet.
  - Les tableaux sont créés par l'opérateur **new**: allocation dynamique par **new**.
  - vérification des bornes à l'utilisation.
  - La taille du tableau est définie à l'exécution. Elle est fixée une fois pour toute.

---

El Mostafa DAOUDI- p. 139

### II. Déclaration et création des tableaux

**1. Déclaration :** La déclaration d'une référence à un tableau précise le type des éléments du tableau.

**int[] tab; Ou int tab[];**

Cette dernière forme permet de mélanger les tableaux de type int et les variables de type int.

**Exemple:**

```
int [] a, tabNotes, b; // a, tabNotes et b sont tous des tableaux de type int.  
double a, tabNotes[ ], b; /* a et b de type double,  
                           tabeNotes est un tableau de type double */
```

**Attention:**

En java la taille des tableaux n'est pas fixée à la déclaration. Elle est fixée quand l'objet tableau sera effectivement créé.

```
int[10] tabNotes; // Erreur car on a fixé la taille pendant la déclaration.  
// Le programme ne compile pas
```

---

El Mostafa DAOUDI- p. 140

## 2. Création: avec l'opérateur new

- Création après la déclaration

```
int [] tab; // déclaration du tableau d'entier tab  
tab = new int[5]; // création et fixation de la taille du tableau
```

- Déclaration et création.

```
int [] tab = new int[5];
```

La création d'un tableau par l'opérateur **new**:

- Alloue la mémoire en fonction du type et de la taille
- Initialise, par défaut, chaque élément du tableau à **0** ou **false** ou **null**, suivant le type de base du tableau.

---

El Mostafa DAOUDI- p. 141

## 3. Accès à la taille et aux éléments d'un tableau:

### Accès à la taille:

La taille est accessible par le "champ final" **length**: **final int length**

```
tab = new int[8];  
int l = tab.length; // la longueur du tableau tab est l = 8
```

**Indices du tableau:** Les indices d'un tableau **tab** commence à **0** et se termine à **(tab.length - 1)**.

### Accès aux élément du tableau:

Soit tab un tableau.

- **tab[i]**, avec  $0 \leq i \leq (\text{tab.length} - 1)$ , permet d'accéder à l'élément d'indice **i** du tableau **tab**.
- Java vérifie automatiquement l'indice lors de l'accès. Il lève une exception si tentative d'accès hors bornes.

```
tab = new int[8];  
int e = tab[8]; // tentative d'accès hors bornes  
/* L'exécution produit le message ArrayIndexOutOfBoundsException */
```

---

El Mostafa DAOUDI- p. 142

### Remarques:

- La taille peut être une variable.  
`int n=Clavier.lireInt( ); // saisir au clavier l'entier n.`  
`int tab[ ]=new int [n];`
- Après exécution, **la taille ne pourra plus être modifiée.**
- Par contre la référence tab peut changer. Elle peut par exemple référencer un autre objet de taille différente : par affectation ou par `new`.

### Exemple:

```
double [] tab1, tab2; // tab1 et tab2 deux tableaux de type double
tab1=new double [10]; // tab1.length retourne 10
tab2=new double [15]; // tab2.length retourne 15
tab1=new double [36]; /* tab1 référence (désigne) un nouveau tableau de taille
                        tab1.length = 36. */
tab1=tab2; /* tab1 et tab2 désignent le même tableau référencé par tab2.
            tab1.length retourne 15 */
```

El Mostafa DAOUDI- p. 143

## 4. Autres méthodes de création et d'initialisation.

On peut lier la déclaration, la création et l'initialisation explicite d'un tableau. La longueur du tableau ainsi créée est alors calculée automatiquement d'après le nombre de valeurs données (**Attention** : cette syntaxe n'est autorisée que dans la déclaration) :

- Création et initialisation explicite à la déclaration
  - `int tab[] = {5, 2*7, 8}; // la taille du tableau tab est fixé à 3`
  - `Etudiant [] etudiantsSMI = {`  
`new Etudiant("Mohammed", "Ali"),`  
`new Etudiant("Fatima", "Zahra")`  
`}`  
`// la taille du tableau etudiantsSMI est fixé à 2`
- Initialisation lors de la création.  
`int[] tab;`  
`tab = new int[] {6, 9, 3, 10}; // la taille du tableau est fixé à 4`

El Mostafa DAOUDI- p. 144



### **Exemple:**

```
public class ExempleTableau {  
    public static void main(String[] args) {  
        // DECLARATION  
        double[] tab;  
        // CREATION d'un tableau de dimension 100  
        tab = new double[100];  
        // AFFECTATION DES ELEMENTS DU TABLEAU  
        for (int i=0; i<tab.length; i++)  
            tab[i] = 1.;  
        // AFFICHAGE  
        for (int i=0; i<tab.length; i++)  
            System.out.print(tab[i]+" ");  
        System.out.println();  
    }  
}
```

---

El Mostafa DAOUDI- p. 145

### **Exemple de tableau de paramètres de la ligne de commande**

```
public class TestArguments {  
    public static void main(String[] args) {  
        for (int i=0; i < args.length; i++)  
            System.out.println("argument "+i+ " = "+args[i]);  
    }  
}  
$ javac TestArguments // compilation  
$ java TestArguments arg1 arg2 /* exécution en passant 2 arguments  
                                (deux chaînes de caractères arg1 et arg2)  
                                à la ligne de commande */
```

Le programme affichera:

```
argument 0 = arg1  
argument 1 = arg2
```

---

El Mostafa DAOUDI- p. 146

### III. Affectation de Tableaux

- Java permet de manipuler globalement les tableaux par affectation de leurs références.
- l'affectation ne modifie que la référence. Elle permet le changement de références.

Soient tab1 et tab2 deux tableaux  
tab1=tab2;

=> On perd l'objet initialement référencé par tab1.

=> La référence contenue dans tab2 est affectée à tab1. Maintenant tab1 et tab2 désignent le même objet tableau qui était initialement référencé par tab2.

```
tab1[2]=3;    // => tab2[2] =3.  
tab2[4]=6;    // => tab1[4] = 6;
```

**Attention:** Si on veut copier uniquement les valeurs de tab2 dans tab1, alors dans ce cas on peut faire une copie élément par élément

---

El Mostafa DAOUDI- p. 147

### IV. Copie de tableau

- Méthode « manuelle » : création d'un nouveau tableau puis copie élément par élément.
- Copie par appel de : System.arraycopy() :  
System.arraycopy(src, srcPos, dest, destPos, nb);  
**src** : tableau source.  
**srcPos** : indice du 1er élément copié à partir de src.  
**dest** : tableau destination  
**destPos** : indice de dst où sera copié le 1er élément  
**nb** : nombre d'éléments copiés

Copie nb éléments du tableau **src** (source) à partir de l'indice **srcPos** et les affecte dans le tableau **dest** (destination) à partir de l'indice **destPos**.

**Exemple:**

```
import java.lang.*  
System.arraycopy(tab1, 6, tab2, 2, 50);  
copie de 50 éléments de tab1 à partir de l'indice 6 et les affecte  
dans tab2 à partir de l'indice 2.
```

---

El Mostafa DAOUDI- p. 148

Allocation et copie par appel de: `Arrays.copyOf()` ou `Arrays.copyOfRange()`

**Exemple:** Soit `tab` un tableau d'entier déjà créé et initialisé. Création et initialisation du tableau `t2` à partir du tableau `tab`

```
import java.util.Arrays;
int[] t2= Arrays.copyOf(tab, 10);
// 1. crée le tableau t2
// 2. affecte au tableau t2 les 10 premiers éléments du tableau tab.

int[] t3= Arrays.copyOfRange(tab, debut, fin);
// 1. crée le tableau t3
//2. affecte à t3 les éléments de tab situés entre
// les indices: debut et (fin-1) :
```

---

El Mostafa DAOUDI- p. 149

## **V. Comparer 2 tableaux de type primitifs**

Comparer le contenu de deux tableaux:

1. On peut comparer les éléments des deux tableaux un à un .
2. On peut aussi utiliser la méthode **`equals()`** de la classe **`Arrays`** de la façon suivante:

```
import java.util. Arrays;
....
// soient tab1 et tab2 deux tableau
// On suppose que les deux tableaux sont déjà initialisés
boolean b=Arrays.equals(tab1,tab2);
/* compare les contenus des tableaux tab1 et tab2 et retourne
le résultat de la comparaison dans la variable booléenne b */

....
```

---

El Mostafa DAOUDI- p. 150

## VI. Tableaux d'objets

Les éléments d'un tableau peuvent être d'un type objet.

**Attention:** La déclaration d'un tableau d'objets crée uniquement des «cases» pour stocker des *références aux* objets, mais ne crée pas les objets eux-mêmes. Ces références valent initialement **null**. Il faut créer les objets avant de les utiliser.

---

El Mostafa DAOUDI- p. 151

### Exemple

Considérons l'exemple de **la classe Etudiant**.

Soit setCNE(int ) une méthode de la classe Etudiant.

```
Etudiants [] etudiantSMI = new Etudiant[100];
```

```
/* déclare un tableau de 100 étudiants. Chaque élément du tableau contient  
une référence vers un objet de type Etudiant */
```

```
etudiantSMI[0].setCNE(11225467);
```

```
/* génère une erreur, car pour le moment « etudiantSMI[0] » ne contient que  
la référence vers un objet de type Etudiant. Avant d'utiliser l'objet «  
etudiantSMI[0] » il faut le créer. */
```

```
etudiantSMI[0] = new Etudiant(); // Création de l'objet etudiantSMI[0]
```

```
etudiantSMI[0].setCNE(11225467); // ok! Car l'objet etudiantSMI[0] est créé.
```

---

El Mostafa DAOUDI- p. 152

**Attention:**

Pour les tableaux d'objets, la méthode **equals()** de la classe **Arrays** compare les références.

**Exemple:** Considérons la classe «Point» (voir TD).

```
public class TestTableauObjet {  
    public static void main(String[] args) {  
        Point p1[]=new Point[2];    // p1 est tableau de Point de taille 2  
        Point p2[]=new Point[2];    // p2 est tableau de Point de taille 2  
        Point p3[]=new Point[2];    // p3 est tableau de Point de taille 2  
        p1[0]=new Point(1,2); p1[1]=new Point(5,8);    // initialisation  
        p2[0]=new Point(1,2); p2[1]=new Point(5,8);    // initialisation  
        boolean b1=Arrays.equals(p1,p2);    // retourne false  
        p3[0]=p1[0]; p3[1]=p1[1]  
        boolean b2=Arrays.equals(p1,p3);    // retourne true  
        p1=p2;  
        boolean b3=Arrays.equals(p1,p2);    // retourne true  
    }  
}
```

---

El Mostafa DAOUDI- p. 153

**VII. Les tableaux en argument d'une méthode:**

- Le passage des tableaux se fait par spécification du tableau comme paramètre d'une méthode.
- On ne doit pas spécifier la taille de chaque dimension. Cette taille peut être obtenue dans la méthode à l'aide de l'attribut *length*.
- Le passage des tableaux comme paramètres des méthodes se fait par **référence** (comme les objets) et non par **copie** (comme les types primitifs)
- La méthode agit directement sur le tableau et non sur sa copie.

---

El Mostafa DAOUDI- p. 154

### VIII. Tableaux à deux dimensions

- Un tableau à deux dimensions, ou matrice, représente un rectangle composé de lignes et de colonnes.
- Si tab est un tableau à deux dimensions, l'élément de la ligne i et de la colonne j est désigné par tab[i][j].

- **Déclaration**

int[][] notes; // déclare un tableau à deux dimensions

Chaque élément du tableau contient une référence vers un tableau

- **Création:**

On utilise l'opération **new** de création de tableaux. Il faut donner au moins la première dimension

---

El Mostafa DAOUDI- p. 155

### **Exemple:**

```
notes = new int[30][3]; // créé un tableau de 30 étudiants, chacun a au plus 3 notes
```

Nous pouvons considérer le tableau *notes* comme un rectangle avec 30 lignes et 3 colonnes. Par convention, la première dimension est celle des lignes, et la deuxième, celle des colonnes. Les indices débutent à 0. Lors de sa création, les éléments du tableau sont initialisés par défaut.

```
notes = new int[30][]; // crée un tableau de 30 étudiants, chacun a un nombre variable de notes
```

### **Remarques:**

- Les tableaux à plusieurs dimensions sont définis à l'aide de tableaux des tableaux. Ce sont des tableaux dont les composantes sont elles-mêmes des tableaux.
- Un tableau à deux dimensions de tailles n et m, est en réalité un tableau unidimensionnel de n lignes, où chaque ligne est un tableau de m composantes.

---

El Mostafa DAOUDI- p. 156

- **Dimensions du tableau**

Soit t un tableau a deux dimensions:

- t.length : donne la longueur de la première dimension, c'est-à-dire le nombre de lignes du tableau t.
- t[i].length : donne la longueur de la ligne i de t, autrement dit, le nombre de colonnes de cette ligne.

- **Initialisation**

- Comme pour les tableaux à une dimension, on peut faire l'initialisation d'une matrice par énumération de ses composantes.

**Exemple 1:**

```
int [][] tab = { {1,2,3,4}, {5,6,8}, {9,10,11,12}};  
/* créé une matrice à 3 lignes (un tableau à 3 composante). Chaque ligne  
(chaque composante) est un tableau unidimensionnelle à un nombre variable de  
composantes:  
*/  
int [] t = tab[1]; // crée le tableau t= {5,6,8}  
  
tab[1][2] = 8;
```

---

El Mostafa DAOUDI- p. 157

**Exemple 2:**

```
int[][] t;  
t = new int[2][];  
int[] t0 = {0, 1};  
t[0] = t0; // t[0] est un tableau  
t[1] = new int[] {2, 3, 4, 5}; // Déclaration et initialisation  
for (int i = 0; i < t.length; i++) {  
    for (int j = 0; j < t[i].length; j++) {  
        System.out.print(t[i][j] + " ");  
    }  
    System.out.println();  
}  
}  
Affiche:  
0 1  
2 3 4 5
```

---

El Mostafa DAOUDI- p. 158

### **Exemple 3:**

Possibilité de forme non rectangulaire :

```
float triangle[][];  
int n;  
// ...  
triangle = new float[n][]; //créé un tableau à n lignes  
for (int i=0; i<n; i++) {  
    triangle[i] = new float[i+1]; // la ligne i est de dimension i  
}
```

Quelques (rares) fonctions à connaître sur les tableaux de tableaux :

```
Arrays.deepEquals(); // à explorer  
Arrays.deepToString(); //à explorer
```

---

El Mostafa DAOUDI- p. 159

## **Ch. VI. Héritage**

L'héritage est une notion fondamentale en Java et de manière générale dans les langages de programmation par Objets.

Il permet de créer de nouvelles classes par combinaison de classes déjà existantes sans toucher au code source des classes existantes.

Il permet de réutiliser une classe existante:

- en l'adaptant: ajouter de nouveaux attributs et méthodes à la nouvelles classes ou adapter (personnaliser) les attributs et les méthodes de la classes existante.
- en la factorisant: plusieurs classes peuvent partager les mêmes attributs et les mêmes méthodes.

---

El Mostafa DAOUDI- p. 160



### 1. Syntaxe:

`class <ClasseDerivee> extends <ClasseDeBase>`

### Interprétation:

Permet de définir un lien d'héritage entre deux classes:

- La classe <ClasseDeBase> est le nom de la classe de base. On l'appelle aussi une *classe mère*, une *classe parente* ou une *super-classe*. Le mot clef *extends* indique la classe mère.
- La classe <ClasseDerivee> est le nom de la classe dérivée. Elle hérite de la classe <ClasseDeBase>. On l'appelle aussi une *classe fille* ou une *sous-classe*.

### Remarques:

- On a seulement besoin du code compilé de la classe de base.
- Toutes les classe héritent de la classe *Object*. Par défaut, on ne met pas *extends Object* dans la définition d'une classe. **Exemple:**

```
public class TestClasse { ...} // équivalent à :  
public class TestClasse extends Objet { ...}
```

---

El Mostafa DAOUDI- p. 161

### **Lorsqu'une classe hérite d'une autre classe**

- La classe dérivée bénéficie automatiquement des définitions des attributs et des méthodes de la classe mère.
  - Elle hérite de tous les attributs (y compris "static") et méthodes membres de la classe mère (sauf les méthodes privées et les constructeurs) .
- En plus, la classe dérivée peut avoir des attributs et des méthodes supplémentaires correspondants à sa spécificité (ses propres attributs et méthodes).
- Elle peut y ajouter ses propres définitions.
  - redéfinir des méthodes (personnaliser les méthodes): exactement le même nom et la même signature.
  - surcharger des méthodes: même nom mais pas les mêmes arguments (signature).

---

El Mostafa DAOUDI- p. 162

### Exemple:

```
class ClasseDeBase { // classe mère
    // Définition des attributs et des méthodes
}
class ClasseDerivee extends ClasseDeBase{
    // ClasseDerivee : Sous-classe directe de la classe ClasseDeBase
    // Définition de nouveaux attributs et méthodes (propres à la classe ClasseDerivee)
    // Adaptation des méthodes qui sont déjà définies dans la classe mère
    // ClasseDerivee hérite des méthodes et attributs de la classe ClasseDeBase
}
class ClasseSousDerivee extends ClasseDerivee {
    /* ClasseSousDerivee: Sous-classe directe de la classe ClasseDerivee et sous-classe
    indirecte de la classe ClasseDeBase */
    // Définition de nouveaux attributs et méthodes (propres à la classe ClasseSousDerivee)
    // Adaptation des méthodes qui sont déjà définies dans les superclasses directes ou indirectes
    // ClasseSousDerivee hérite des méthodes et attributs des superclasses.
}
```

---

El Mostafa DAOUDI- p. 163

### Accès depuis l'extérieur:

Un objet (une instance ) d'une classe dérivée peut appeler depuis l'extérieur (accès depuis l'extérieur)

- ses propres méthodes et attributs publics et protected.
- Les méthodes et attributs publics et protected de la classe de **base**.

### Accès depuis l'intérieur:

- Une méthode d'une classe dérivée a accès aux membres publics et protected de sa classe de base.
- Une méthode d'une classe dérivée n'a pas accès aux membres privées de sa classe de base.

La classe dérivée ne peut accéder qu'aux membres (attributs ou méthodes) publics et protégés hérités.

---

El Mostafa DAOUDI- p. 164

### Remarques1:

Quand on écrit la classe dérivée on doit seulement:

- écrire le code (variables ou méthodes) lié aux nouvelles possibilités: offrir de nouveaux services.
- redéfinir certaines méthodes: enrichir les services rendus par une classe.

### Remarques2:

- Java, ne permet pas l'héritage multiple. Il permet l'héritage *simple uniquement*: chaque classe a une et une seule classe mère dont elle hérite les variables et les méthodes.
- C++ permet l'héritage multiple.

---

El Mostafa DAOUDI- p. 165

## 2. Construction des classes dérivées

Lorsqu'on construit une instance de la classe dérivée, on obtient un objet :

- dont une partie est construite grâce à la définition de la classe dérivée
- et une partie grâce à la définition de la super-classe

En Java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet.

⇒ Le constructeur de la classe dérivée doit faire appel au constructeur de la classe de base en appelant explicitement la méthode **super()** ainsi que la liste des paramètres appropriés.

### Remarques:

- les constructeurs ne sont pas hérités, mais on peut appeler ceux de la classe mère avec **super(...)**
- l'appel du constructeur de la classe de base (appel de la méthode **super()**) doit être **la première instruction** dans la définition du constructeur de la classe dérivée.

---

El Mostafa DAOUDI- p. 166

**Exemple 1:**

```
class ClasseDeBase{
    public ClasseDeBase(int i) { // constructeur de la classe de base
        System.out.println(" Classe de Base: " + i);
    }
}
class ClasseDerivee1 extends ClasseDeBase {
    public ClasseDerivee1(int i, int j) { // constructeur de la classe dérivée 1
        super(i+j); // appel du constructeur ClasseDeBase
        System.out.println(" Classe dérivée1: " + i+ " , "+j);
    }
}
class ClasseDerivee2 extends ClasseDerivee1{
    public ClasseDerivee2(int i) { // constructeur de la classe dérivée 2
        super(i , i+300); // appel du constructeur ClasseDerivee1
        System.out.println(" Classe dérivée2: " + i);
    }
}
```

---

El Mostafa DAOUDI- p. 167

```
public class TestHeritage{
    public static void main (String args[]) {
        ClasseDerivee2 objA=new ClasseDerivee2(7);
    }
}
```

**Sortie:**

Classe de Base : 314  
Classe dérivée1: 7 , 307  
Classe dérivée2: 7

---

El Mostafa DAOUDI- p. 168

### 3. Les cas possibles pour la construction des classes dérivées

#### Exemple 1:

```
class ClasseDeBase{
    public ClasseDeBase(arguments) { // Constructeur de la classe de
base
        ...
    }
    ...
}
class ClasseDerivee extends ClasseDeBase {
    // Pas de constructeur
    ....
}
```

On obtient une erreur de compilation. Un constructeur de la classe dérivée doit être défini et doit appeler le constructeur de la classe de base.

---

El Mostafa DAOUDI- p. 169

#### Exemple 2:

```
class ClasseDeBase{
    public ClasseDeBase(arguments1) { // constructeur de la classe de bas
        ...
    }
}
class ClasseDerivee extends ClasseDeBase {
    public ClasseDerivee(arguments2) { // constructeur de la classe dérivé
        /* Première instruction = Appel de super(arguments1)
        C'est obligatoire sinon une erreur
        */
        ....
    }
}
```

---

El Mostafa DAOUDI- p. 170

### Exemple 3:

```
class ClasseDeBase{
    public ClasseDeBase() { // constructeur sans paramètres de la classe de base
        ...
    }
}
class ClasseDerivee extends ClasseDeBase {
    public ClasseDerivee(arguments2) { // constructeur de la classe dérivée
        // l'appel de super() n'est pas obligatoire.
        /* si super() n'est pas explicitement appelé, alors le constructeur
           sans paramètres de la classe de base qui sera appelé par défaut.
        */
        ....
    }
}
```

---

El Mostafa DAOUDI- p. 171

### Exemple 4:

```
class ClasseDeBase{
    public ClasseDeBase(arguments1) { // constructeur de la classe de base
        ...
    }
    public ClasseDeBase() { // constructeur sans paramètres de la classe de base
        ...
    }
}
class ClasseDerivee extends ClasseDeBase {
    public ClasseDerivee(arguments2) { // constructeur de la classe dérivée
        // l'appel de super(avec ou sans arguments) n'est pas obligatoire.
        /* si super n'est pas explicitement appelé, alors le constructeur sans
           paramètres de la classe de base qui sera appelé par défaut. */
        ....
    }
}
```

---

El Mostafa DAOUDI- p. 172

### **Exemple 5:**

```
class ClasseDeBase{  
    // Pas de constructeur  
    ...  
}  
class ClasseDerivee extends ClasseDeBase {  
    public ClasseDerivee(arguments) {  
        /* appel implicite du constructeur par défaut de la  
        classe de base */  
        ....  
    }  
}
```

La création d'un objet de type « ClasseDerivee » entraîne l'appel du constructeur par défaut de la classe « ClasseDeBase ».

---

El Mostafa DAOUDI- p. 173

### **Exemple 6:**

```
class ClasseDeBase{  
    // Pas de constructeur  
    ...  
}  
class ClasseDerivee extends ClasseDeBase {  
    // Pas de constructeur  
    ....  
}
```

La création d'un objet de type « ClasseDerivee » entraîne l'appel de constructeur par défaut qui appelle le constructeur par défaut de la classe « ClasseDeBase ».

---

El Mostafa DAOUDI- p. 174

**Exemple pratique:**

```
class ClasseDeBase{
    public ClasseDeBase(int i) {
        System.out.println(" Classe de Base: " + i);
    }
}
class ClasseDerivee1 extends ClasseDeBase {
    public ClasseDerivee1() {
        // 1ère instruction obligatoire = appel explicite de super()
        super(4); // par exemple
        System.out.println(" Classe dérivée1: ");
    }
}
class ClasseDerivee2 extends ClasseDerivee1{
    public ClasseDerivee2() {
        // Appel explicite de super() n'est pas obligatoire
        // Appel implicite du constructeur sans paramètres de ClasseDerivee1
        System.out.println(" Classe dérivée2: ");
    }
}
```

---

El Mostafa DAOUDI- p. 175

```
public class TestHeritage{
    public static void main (String args[]) {
        ClasseDerivee2 objA=new ClasseDerivee2();
    }
}
```

**Sortie:**

Classe de Base : 4

Classe dérivée1:

Classe dérivée2:

---

El Mostafa DAOUDI- p. 176



### Résumé:

Si on n'appelle pas le constructeur de la super-classe, le constructeur par défaut est utilisé si:

- aucun constructeur n'est défini
  - au moins un constructeur sans paramètre est défini
- sinon le compilateur déclare une erreur

### Remarques

- Une classe déclarée **final** ne peut pas avoir de classes filles.
- La redéfinition d'une méthode **public** ne peut être **private**
- Une méthode **final** ne peut pas être redéfinie

---

El Mostafa DAOUDI- p. 177

### 4. Notion de la Redéfinition

– on redéfinit une méthode quand une nouvelle méthode a le même nom et la même signature qu'une méthode héritée de la classe mère.

⇒ la redéfinition d'une méthode d'une classe consiste à fournir dans une sous-classe une nouvelle implémentation de la méthode. Cette nouvelle implémentation masque alors complètement celle de la super-classe

**Attention:** Ne pas confondre redéfinition et surcharge des méthodes :

– on surcharge une méthode quand une nouvelle méthode a le même nom, mais pas la même signature, qu'une autre méthode de la même classe

---

El Mostafa DAOUDI- p. 178

```

class ClasseDeBase{
    public f(arg) {
        ...
    }
}
class ClasseDerivee1 extends ClasseDeBase {
    public f(arg) { // la méthode f() est redéfinie dans ClasseDerivee1
        ...
    }
}
class ClasseDerivee2 extends ClasseDeBase{
    public f(arg){ // la méthode f() est redéfinie dans ClasseDerivee2
        ....
    }
}

```

---

El Mostafa DAOUDI- p. 179

## Visibilité

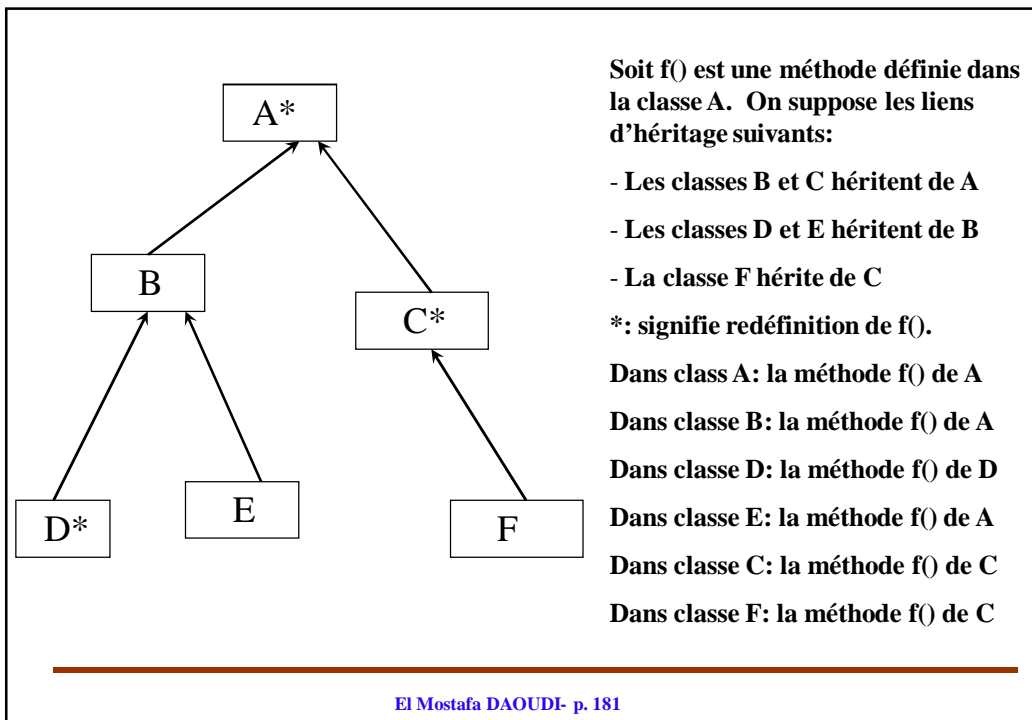
Lorsqu'un attribut ou une méthode ont été définis dans une classe et sont redéfinis dans une classe dérivée (qui en hérite) alors les éléments visibles dans la classe dérivée sont ceux redéfinis dans cette classe. Les éléments de la classe de base (héritée) sont alors masqués.

## Règles

On peut avoir les mêmes méthodes dans des classes héritant les unes des autres. Dans ce cas, c'est la classe la plus dérivée de l'objet qui détermine la méthode à exécuter, sans que le programmeur ait à faire des tests sur le type de l'objet à traiter (voir exemple suivant).

---

El Mostafa DAOUDI- p. 180



### Le mot clé super

- Grâce au mot-clé **super**, la méthode redéfinie dans la sous-classe peut réutiliser du code écrit dans la méthode de la super-classe, qui n'est plus visible autrement.
- **super** a un sens uniquement dans une méthode (comme le mot-clé this).

**Exemple:**

```
class Point {
    private int x,y;
    public void afficher(){
        System.out.println(" je suis en "+ x +" et " +y);
    }
}
class PointCol extends Point {
    public void afficher() { // redéfinition de la méthode afficher()
        super.afficher(); // appel de la méthode afficher() de la classe de base
        System.out.println (" et ma couleur est: "+ couleur);
    }
    private byte couleur;
}
```

---

El Mostafa DAOUDI- p. 183

**D'une manière générale,**

Soit **g()** une méthode définie dans une classe dérivée D.

Dans la méthode **g()**, l'instruction **super.** sert à désigner un membre d'une classe ancêtre (la première classe ancêtre de D contenant le membre désigné par **super**).

**Par exemple:**

**super.f()** désigne la méthode f() définie dans la première classe ancêtre de la classe D.

La recherche de la méthode f() commence dans la classe mère de D, ensuite dans la classe mère de cette classe mère, et ainsi de suite, jusqu'à trouver la définition d'une méthode f() qui sera alors exécutée

**Attention:** Une fois la première classe ancêtre trouvée, on ne peut pas remonter plus haut:

- pas de « **super.super.** »

---

El Mostafa DAOUDI- p. 184

**Exemple:**

```
class ClasseA {
    public String nom="Mostafa";
    public void f(){
        System.out.println("Je suis dans la classe de base ClasseA");
    }
}
class A extends ClasseA {
    public String nom="Ali";
    public void ff(){
        System.out.println("Je suis dans la classe de base A");
    }
}
class B extends A {
    public String nom = "Mohammed";
    public void g(){
        super.f(); // désigne la méthode f() de la classe ancêtre: classe ClasseA
        System.out.println( "Et je suis aussi dans la class dérivée B  : "+nom+ " " +super.nom);
        // super.nom désigne le champ nom de la classe mère : classe A.
    }
}
```

---

El Mostafa DAOUDI- p. 185

```
public class TestSuper {
    public static void main(String[] args) {
        B b = new B();
        b.g();
    }
}
```

**Affiche : ??**

Je suis dans la classe de base ClasseA

Et je suis aussi dans la class dérivée B : Mohammed Ali

---

El Mostafa DAOUDI- p. 186

## **5. Sous-type**

Le type **B** est un sous-type de **A** si on peut affecter une expression de type **B** dans une variable de type **A**.

### **Les type primitifs**

- **int** est un sous type de **float** (int i; float x=i;)
- **float** est un sous type de **double** (float y; double z=y;)
- ...

### **Les objets**

Les sous-classes d'une classe A sont des sous types de A. Dans ce cas on peut écrire:

```
A a = new B(...); // la variable a est de type A, alors que
                  // l'objet référencé par a est de type B.
```

```
A aa; B b=new B();
aa=b; // aa de type A, référence un objet de type B
```

### **Par contre on ne peut pas avoir:**

```
A a=new A();
B b;
b=a; // erreur: on ne peut pas convertir du type A vers le type B
```

---

El Mostafa DAOUDI- p. 187

## **Définitions:**

- **La classe réelle (ou le type réel)** de l'objet est la classe du constructeur qui a créé l'objet

**Exemple:** Soit B une sous classe de A

```
A a = new B(...); // B est la classe (type) réelle de l'objet a
```

- **Le type déclaré (ou la classe déclarée)** de l'objet est le type qui est donné au moment de la déclaration de la variable qui référence l'objet.

**Exemple:** Soit B une sous classe de A

```
A a = new B(...); // A est le type déclaré de l'objet a
```

---

El Mostafa DAOUDI- p. 188

## Cas des tableaux

Soit B une classe qui hérite de la classe A, alors on peut écrire :

```
A[] tab = new B[5];
```

**Attention:** Dans tab[] il faut des objets de la classe réelle et non celles du type déclaré (c'est à dire des objets de type B et non de type A). Par exemple, si on a les instructions suivantes:

```
A[] tab = new B[5]; // A est le type déclaré du tableau tab.
```

*// les éléments de tab doivent être du type réel c'est des instances*

*// de la classe B et non de la classe A.*

```
A a = new A();
```

```
tab[0] = a; // on affecte à tab[0] une instance de type A
```

*/\* Passe à la compilation mais provoquera une erreur à l'exécution car tab[0] reçoit une valeur de type A et non une valeur de type B. \*/*

---

El Mostafa DAOUDI- p. 189

## Ch. VII. Polymorphisme

### I. Exemple introductif:

Considérons l'exemple suivant où ClasseB est une classe qui hérite de la classe ClasseA. Soient f() une méthode qui est redéfinie dans B et g() une méthode définie dans A. On suppose que :

```
class ClasseA {  
    public void f(){  
        System.out.println("Méthode f(): Classe de base A");    }  
    public void g(){  
        System.out.println("Méthode g(): Classe de base A");    }  
}  
class ClasseB extends ClasseA { // ClasseB hérite de la classe ClasseA  
    public void f(){ // la méthode f() est redéfinie dans ClasseB  
        System.out.println("Méthode f(): Classe B dérivée de ClasseA");    }  
    public void h(){ // la méthode h() définie uniquement dans ClasseB  
        System.out.println("Méthode h(): Classe B dérivée de ClasseA");    }  
}
```

---

El Mostafa DAOUDI- p. 190

```

public class TestPolymorphisme {
    public static void main(String [] args ) {
        ClasseA a=new ClasseA();
        ClasseB b = new ClasseB();
        a.f(); // appelle la méthode définie dans ClasseA
        a=b; // le type déclaré de a est ClasseA. Le type réel de a est ClasseB
        a.f(); // appelle la méthode définie dans ClasseB
        a.g(); // appelle la méthode définie dans ClasseA
        b.g(); // appelle la méthode définie dans ClasseA
        b.h(); // appelle la méthode h() définie dans ClasseB
    }
}

```

Il faut noter que la même écriture a.f(); peut correspondre à des appels différents de la méthode f(). Ceci est réalisé grâce au polymorphisme.

---

El Mostafa DAOUDI- p. 191

- Le Polymorphisme veut dire que le même service peut avoir un comportement différent suivant la classe dans laquelle il est utilisé. C'est un concept fondamental de la programmation objet, indispensable pour une utilisation efficace de l'héritage
- Quand on manipule un objet via une référence à une classe mère, ce sont toujours les méthodes (non statiques) de la classe effective de l'objet qui sont appelées

Soit f() une méthode non static redéfinie dans la classe « ClasseB ».

```

ClasseA objA = new ClasseB();

```

```

objA.f(); // Appel de f() redéfinie dans ClasseB (classe effective=classe
          // réelle de l'objet), même si objA est une référence de type
          // ClasseA (Classe déclarée de l'objet).

```

---

El Mostafa DAOUDI- p. 192



**Attention c'est important:**

Si on a:

```
B b = new B();
```

```
b.h(); // appelle la méthode h() définie dans B
```

```
A a=new B();
```

```
a.h(); /* erreur à la compilation même si la classe réelle  
possède la méthode h(). En effet la classe déclarée  
(classe A) ne possède pas la méthode h(). */
```

- En effet, en Java, dès la compilation on doit garantir l'existence de la méthode appelée (typage statique) : la classe déclarée de l'objet qui reçoit le message (ici la classe A ou une de ces classes ancêtres (polymorphisme)) doit posséder cette méthode ().

---

El Mostafa DAOUDI- p. 193

- Le polymorphisme est obtenu grâce au mécanisme de la liaison retardée (tardive) « *late binding* »: la méthode qui sera exécutée est déterminée
  - seulement à l'exécution, et pas dès la compilation
  - par le type réel de l'objet qui reçoit le message (et pas par son type déclaré)

---

El Mostafa DAOUDI- p. 194

## **II. Mécanisme de la liaison retardée: cas des méthodes redéfinies**

Soit ClasseB la classe réelle d'un objet objB (ClasseB est la classe du constructeur qui a créé l'objet objB) et soit ClasseA la classe de déclaration de l'objet objB (ClasseA objB = new ClasseB();). Si on a l'instruction: objB.f();

quelle méthode f() sera appelée ?

1. Si la méthode f() n'est pas définie dans une classe ancêtre de la classe de déclaration (la classe ClasseA) alors erreur de compilation.
2. Si non
  - Si la méthode f() est redéfinie dans la classe ClasseB, alors c'est cette méthode qui sera exécutée
  - Sinon, la recherche de la méthode f() se poursuit dans la classe mère de ClasseB, puis dans la classe mère de cette classe mère, et ainsi de suite, jusqu'à trouver la définition d'une méthode f() qui sera alors exécutée.

---

El Mostafa DAOUDI- p. 195

La méthode appelée ne dépend que du type réel de l'objet et non du type déclaré.

- Soit B une classe qui hérite de la classe A, et soit f() une méthode définie dans A et redéfinie dans B.
- Soit C une classe qui hérite de la classe B.

**A a = new B();**

**a.f();** // La méthode f() appelée est celle définie dans B.

Maintenant si on:

**A a = new C();** // ok car C hérite de B qui hérite de A

**a.f();**

Si la méthode f() est redéfinie dans C alors la méthode appelée est celle définie dans C.

Si la méthode f() n'est redéfinie dans C alors c'est la méthode f() redéfinie dans B qui est appelée.

---

El Mostafa DAOUDI- p. 196

### III. Utilités du Polymorphisme:

Le polymorphisme permet d'éviter les codes qui comportent de nombreux embranchements et tests.

#### Exemple

Considérons une classe ClasseA. Supposons que les classes ClasseB et ClasseC héritent de la super classe ClasseA. Dans un tableau hétérogène, on range des objets de type ClasseB et ClasseC. Ensuite on affiche le contenu du tableau.

#### instanceof (voir plus loin):

Soit b une instance d'une classe B alors

- **(b instanceof B)** retourne **true** : signifie que b est une instance de B.
- si B est une sous classe de A alors **(b instanceof A)** retourne **true**.

---

El Mostafa DAOUDI- p. 197

#### Exemple

```
class ClasseA {  
    public void f() {} // méthode vide  
    public void g() {} // méthode vide  
}  
class ClasseB extends ClasseA {  
    public void f() { // traitement propre à ClasseB  
        System.out.println("traitement dans Classe B ");  
    }  
}  
class ClasseC extends ClasseA {  
    public void g() { // traitement propre à ClasseC  
        System.out.println("traitement dans ClasseC ");  
    }  
}
```

---

El Mostafa DAOUDI- p. 198

```

public class TestPolymorphisme {
    public static void main(String[] args) {
        ClasseA [] objA = new ClasseA[3]; // objA tableau d'objets
        objA[0]=new ClasseB(); // création de l'objet objA[0]
        objA[1]=new ClasseC(); // création de l'objet objA[1]
        objA[2]=new ClasseC(); // création de l'objet objA[2]
        // Pour chaque élément du tableau, faire le traitement correspondant
        for (int i=0; i < objA.length; i++) {
            if (objA[i] instanceof ClasseB)
                objA[i].f();
            else if (objA[i] instanceof ClasseC)
                objA[i].g();
        }
    }
}

```

---

El Mostafa DAOUDI- p. 199

### **Inconvénients dans le code précédent:**

1. Si on veut rajouter une nouvelle sous classe, par exemple ClasseD, dotée de la méthode h() pour faire un traitement propre à la sous classe ClasseD, alors on est obligé de:
  - changer le code source de la classe de base aux niveaux définition de la classe : rajouter la méthode vide `public void h() {}`.
  - Rajouter un branchement pour la sous classe ClasseD.
2. Si on traite plusieurs sous classes, alors le code comportera plusieurs tests.

En exploitant le polymorphisme, on peut améliorer le code précédent

- Eviter les tests
- Rendre le code extensible: rajouter de nouvelles sous-classes sans toucher au code source existant).

---

El Mostafa DAOUDI- p. 200

**L'exemple peut être réécrit de la manière suivante en exploitant le polymorphisme**

```
class ClasseA {
    public void f() {} // méthode vide
}
class ClasseB extends ClasseA {
    public void f() { // traitement propre à ClasseB
        System.out.println("traitement dans ClasseB ");
    }
}
class ClasseC extends ClasseA {
    public void f() { // traitement propre à ClasseC
        System.out.println(" traitement dans ClasseC");
    }
}
```

---

El Mostafa DAOUDI- p. 201

```
public class TestPolymorphisme {
    public static void main(String[] args) {
        ClasseA [] objA = new ClasseA[3];
        objA[0] = new ClasseB();
        objA[1] = new ClasseC();
        objA[2] = new ClasseC();
        for (int i=0; i < objA.length; i++)
            objA[i].f();

        // On n'a pas besoin de faire des tests: la méthode appelée
        // correspond à la méthode de la classe effective de l'objet :
        // principe du polymorphisme.
        //
    }
}
```

---

El Mostafa DAOUDI- p. 202

## Ch. VIII. La gestion des exceptions

### I. Définition et principe

- Une exception est un mécanisme pour traiter les anomalies qui se produisent pendant l'exécution.

#### **Principe**

- Principe fondamental = séparer la détection et le traitement des anomalies :
  - signaler tout problème dès sa détection.
  - mais regrouper le traitement des problèmes ailleurs, en fonction de leur type
- Plutôt que de compliquer le code du traitement normal, on traite les conditions anormales à part
- Le traitement « normal » apparaît ainsi plus simple et plus lisible

---

El Mostafa DAOUDI- p. 203

### II. Gestion des erreurs sans utiliser le mécanisme des exceptions

Considérons l'exemple de la classe point qui a:

- un constructeur a deux arguments qui permet de positionner le point dans le plan.

Le programme doit s'arrêter si au moins une des coordonnées du point est négative.

---

El Mostafa DAOUDI- p. 204

```

public class Point {
    private int x, y;
    public Point(int x, int y) {
        if ((x < 0) || (y < 0)) { // Détection de l'erreur (l'Exception)
            System.out.println("Erreur de Construction"); /* traitement
                                                             en cas d'erreur (traitement de l'exception) */
            System.exit(-1);
        }
        else {
            this.x = x ; this.y = y; // traitement normal
        }
    }
}

```

---

El Mostafa DAOUDI- p. 205

### Un 1<sup>er</sup> test

```

public class TestException {
    public static void main(String[] argv) {
        private int i=3;
        // début du bloc susceptible de générer une exception
        Point a = new Point(6,1);
        Point b = new Point(3, 4);
        // Fin du bloc susceptible de générer une exception
        System.out.println (" Exécution bien passée i= "+i);
    }
}

```

### Sortie du programme:

Exécution bien passée i=3

Les appels de Point(6,1) et Point(3,4) n'ont généré aucune exception..

---

El Mostafa DAOUDI- p. 206

### Un 2<sup>ème</sup> test

```
public class TestException {  
    public static void main(String[] argv) {  
        private int i=3;  
        // début du bloc susceptible de générer une exception  
        Point a = new Point(6,1);  
        Point b = new Point(-2, 4);  
        // Fin du bloc susceptible de générer une exception  
        System.out.println (" Exécution bien passée i= "+i);  
    }  
}
```

#### Sortie du programme:

Erreur de Construction

L'appel de Point(6,1) n'a pas généré d'exception, donc il a été exécuté normalement.

Par contre l'appel de Point(-2,4) a généré une exception et a provoqué l'arrêt du programme (**appel de System.exit(-1);**). Par conséquent l'instruction System.out.println() n'a pas été exécutée.

---

El Mostafa DAOUDI- p. 207

### III. Gestion des erreurs en utilisant le mécanisme des exceptions

Dans l'exemple précédent, le traitement normal, la détection de l'exception et son traitement ne sont pas séparés.

Java permet de séparer la détection de l'exception et de son traitement.

Le traitement normal et le traitement des exceptions sont dissociés.

- Au lieu de compliquer le code du traitement normal, on traite les conditions anormales à part.
- Le traitement « normal » apparaît ainsi plus simple et plus lisible.
- Le traitement des exceptions (erreurs) s'effectue dans une zone du programme spéciale (bloc « **catch** »).

---

El Mostafa DAOUDI- p. 208



### **Pratiquement:**

Le code dans lequel une exception peut se produire est mis dans un bloc `try`, c'est-à-dire on délimite un ensemble d'instructions susceptibles de déclencher une exception par des blocs `try {...}`

```
try {  
    /* Code dans lequel une exception peut se produire */  
}
```

La gestion des exceptions est obtenue par des blocs `catch`, où un bloc `catch` est associé à une exception donnée.

**Attention:** Le bloc `catch` doit être juste après le bloc `try`, sinon erreur de compilation.

```
catch ( Type1Exception e) { // de type Type1Exception qui hérite de Exception  
    /* code de la gestion des exceptions */  
}  
catch ( Type2Exception e) { // de type Type2Exception qui hérite de Exception  
    /* code de la gestion des exceptions */  
}  
.....
```

---

El Mostafa DAOUDI- p. 209

Lorsqu'une exception se produit dans le bloc `try`, alors :

- un saut est effectué vers un bloc (s) `catch`
- les blocs `catch` sont vérifiés un après l'autre jusqu'à ce qu'on trouve un bloc correspondant à l'exception
- le code du bloc `catch` est exécuté

**Exemple:** On reprend l'exemple précédent

---

El Mostafa DAOUDI- p. 210

### Un 1<sup>er</sup> Test

```
public class Test {  
    public static void main(String[] argv) {  
        try { // dans ce bloc une exception peut se produire  
            Point b = new Point(-2, 4);  
            Point a = new Point(6, 1);  
        }  
        catch (ErrConst e) { // Erreur de type ErrConst qui hérite de Exception  
            // dans ce bloc on traite les exceptions  
            System.out.println("Erreur de Construction");  
            System.exit(-1);  
        }  
        System.out.println(" Exécution bien passée");  
    }  
}
```

#### Sortie du programme:

Erreur de Construction

L'appel de Point(-2,4); génère une exception, donc un saut est effectué vers le bloc catch().

L'exécution du code du bloc catch() a provoqué l'arrêt du programme (**appel de System.exit(-1);**), par conséquent l'appel de Point(6,1) et les instructions après le bloc try { } n'ont pas été exécutées.

---

El Mostafa DAOUDI- p. 211

### Un 2<sup>ème</sup> Test

```
public class Test {  
    public static void main(String[] argv) {  
        try {  
            Point a = new Point(9, 5);  
            Point b = new Point(3, 7);  
        }  
        catch (ErrConst e) { // Erreur de type ErrConst qui hérite de Exception  
            System.out.println("Erreur de Construction");  
            System.exit(-1);  
        }  
        System.out.println(" Exécution bien passée");  
    }  
}
```

#### Sortie du programme:

Exécution bien passée

Les appels de Point(9,5) et Point(3,7) n'ont pas généré d'exceptions donc le code du bloc catch() { } n'a pas été exécuté par conséquent l'exécution est allée au-delà du bloc try.

---

El Mostafa DAOUDI- p. 212

#### **IV. Lancement (déclenchement) d'une exception**

Une méthode déclare qu'elle peut générer une exception par le mot clé **throws**. Ensuite la méthode lance une exception, en créant une nouvelle valeur (un objet) d'exception en utilisant le mot clé **throw**

##### **Exemple:**

```
public Point (int x, int y) throws ErrConst {  
    // Déclare que le constructeur Point() peut générer une exception  
    if ((x < 0) || (y < 0)) throw new ErrConst();  
    // Détection de l'exception et Création d'une nouvelle valeur d'exception  
    this.x = x ; this.y = y; // traitement normal  
}
```

« ErrConst » est une classe qui hérite de la classe « Exception ». Elle peut être définie de la manière suivante:

```
class ErrConst extends Exception {  
}
```

##### **N.B. Est-ce que cette définition est juste ??**

Si ça passe à la compilation, veut dire que la classe « Exception » n'admet aucun constructeur ou elle admet au moins un constructeur sans paramètres.

---

El Mostafa DAOUDI- p. 213

##### **Un Premier exemple complet:**

```
class ErrConst extends Exception {}  
class Point { private int x, y;  
    public Point(int x, int y) throws ErrConst { // déclare une exception  
        if ((x < 0) || (y < 0)) throw new ErrConst(); // déclenche une exception  
        this.x = x ; this.y = y; // traitement normal  
    }  
}  
public class Test {  
    public static void main(String[] argv) {  
        try {  
            Point a = new Point(6, 1);  
            Point b = new Point(-2, 4);  
        }  
        catch (ErrConst e) { // Erreur de type ErrConst qui hérite de Exception  
            System.out.println("Erreur de Construction");  
            System.exit(-1);  
        }  
    }  
}
```

---

El Mostafa DAOUDI- p. 214

### Un Deuxième exemple complet:

Supposons maintenant que la classe Point possède une méthode déplace() et que le point doit rester dans le plan positif. Dans ce cas, le constructeur Point() et la méthode déplace() doivent déclencher une exception.

```
class ErrConst extends Exception {}
class ErrDepl extends Exception {}

class Point {
    private int x, y;
    public Point(int x, int y) throws ErrConst { // déclare une exception
        if ((x < 0) || (y < 0)) throw new ErrConst(); // déclenche une exception
        this.x = x ; this.y = y;
    }
    public void déplace(int dx, int dy) throws ErrDepl{
        if ((x+dx < 0) || (y+dy < 0)) throw new ErrDepl();
        x = x+dx ; y = y+dy;
    }
}
```

---

El Mostafa DAOUDI- p. 215

```
public class Test {
    public static void main(String[] argv) {
        try {
            Point a = new Point(1,4);
            a.déplace(0,1);
            Point b = new Point(7, 4);
            b.déplace(3,-5);
        }
        catch (ErrConst e) {
            System.out.println("Erreur de Construction");
            System.exit(-1);
        }
        catch (ErrDepl ed) {
            System.out.println("Erreur de Déplacement");
            System.exit(-1);
        }
    }
}
```

---

El Mostafa DAOUDI- p. 216

## **V. Bloc finally**

- c'est une instruction optionnelle qui est exécutée quelle que soit le résultat du bloc try (c'est à dire qu'il ait déclenché ou non une exception)
- Il permet de spécifier du code dont l'exécution est garantie quoi qu'il arrive.

```
try {  
    ...  
}  
catch (...) {  
    ...  
}  
finally {  
    ...  
}
```

---

El Mostafa DAOUDI- p. 217

### **Exemple:**

```
public class TestFinally {  
    public static void main(String[] argv) {  
        try {  
            Point a = new Point(6, 1);  
            a = new Point(-2, 4);  
        }  
        catch (ErrConst e) { // Erreur de type ErrConst qui hérite de Exception  
            System.out.println("Erreur de Construction");  
        }  
        finally {  
            System.out.println("Fin du Programme");  
        }  
    }  
}
```

### **Affichage:**

```
Erreur de Construction  
Fin du Programme
```

Dans toutes les exécutions (déclenchement ou non d'exception), les instructions du bloc **finally** seront exécutées.

---

El Mostafa DAOUDI- p. 218

## VI. Constructeurs des exceptions

La création d'exception personnalisée peut être réalisée en rajoutant des constructeurs et des membres supplémentaires. Par convention, toutes les exceptions doivent avoir au moins 2 constructeurs :

- un sans paramètre
- un autre dont le paramètre est une chaîne de caractères utilisée pour décrire le problème

## Méthodes de la classe Throwable

- **Exception()**: constructeur sans argument
- **Exception(String)**: Constructeur avec Argument
- **getMessage()** retourne le message d'erreur décrivant l'exception
- **printStackTrace()** affiche sur la sortie standard la liste des appels de méthodes ayant conduit à l'exception

---

El Mostafa DAOUDI- p. 219

## Méthode printStackTrace()

```
class Point {  
    private int x, y;  
    public Point(int x, int y) throws Exception {  
        if ((x < 0) || (y < 0)) throw new Exception();  
        this.x = x ; this.y = y;  
    }  
}
```

---

El Mostafa DAOUDI- p. 220

```

public class ExceptionTestStack {
    public static void main(String[] argv) {
        try {
            Point a = new Point(6, 1);
            a = new Point(-2, 4);
        }
        catch (Exception e) { // Erreur de type Exception
            System.out.println("Erreur de Construction");
            e.printStackTrace();
            System.exit(-1);
        }
    }
}

```

#### **Sortie du Programme:**

Erreur de Construction

ErrConst

at Point.<init>(ExceptionTestStack.java:5)

at ExceptionTestStack.main(ExceptionTestStack.java:14)

---

El Mostafa DAOUDI- p. 221

#### **Méthode printStackTrace(): (avec exception personnalisée)**

```

class ErrConst extends Exception{ } // exception personnalisée

```

```

class Point {
    private int x, y;
    public Point(int x, int y) throws ErrConst {
        if ((x < 0) || (y < 0)) throw new ErrConst();
        this.x = x ; this.y = y;
    }
}

```

---

El Mostafa DAOUDI- p. 222

```

public class ExceptionTestStack {
    public static void main(String[] argv) {
        try {
            Point a = new Point(6, 1);
            a = new Point(-2, 4);
        }
        catch (ErrConst e) { // Erreur de type ErrConst qui hérite de Exception
            System.out.println("Erreur de Construction");
            e.printStackTrace();
            System.exit(-1);
        }
    }
}

```

#### **Sortie du Programme:**

Erreur de Construction

ErrConst

at Point.<init>(ExceptionTestStack.java:5)

at ExceptionTestStack.main(ExceptionTestStack.java:14)

---

El Mostafa DAOUDI- p. 223

#### **Méthode getMessage().**

```

class Point {
    private int x, y;
    public Point(int x, int y) throws Exception {
        if ((x < 0) || (y < 0)) throw new Exception(" Erreur aux points x=" + x + " et y=" + y);
        this.x = x ; this.y = y;
    }
}

```

---

El Mostafa DAOUDI- p. 224



```

public class Test {
    public static void main(String[] argv) {
        try {
            Point a = new Point(6, 1);
            a = new Point(-2, 4);
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
            System.exit(-1);
        }
    }
}

```

### **Sortie:**

Erreur au point x= -2 et y=4

---

El Mostafa DAOUDI- p. 225

### **Méthode getMessage(): Exception personnalisée**

On définit dans l'exception personnalisée, un constructeur qui a pour argument une chaîne de caractères:

```

class ErrConst extends Exception { // l'exception personnalisée
    public ErrConst() {}
    public ErrConst(String msg) {
        super(msg);
    }
}

class Point {
    private int x, y;
    public Point(int x, int y) throws ErrConst {
        if ((x < 0) || (y < 0)) throw new ErrConst(" Erreur aux points x=" +x+" et y=" +y);
        this.x = x ; this.y = y;
    }
}

```

---

El Mostafa DAOUDI- p. 226

```
public class Test {  
    public static void main(String[] argv) {  
        try {  
            Point a = new Point(6, 1);  
            a = new Point(-2, 4);  
        }  
        catch (ErrConst e) {  
            System.out.println(e.getMessage());  
            System.exit(-1);  
        }  
    }  
}
```

**Sortie:**

Erreur au point x= -2 et y=4

---

El Mostafa DAOUDI- p. 227

**Voir TD pour d'autres exemples d'exceptions personnalisées**

---

El Mostafa DAOUDI- p. 228

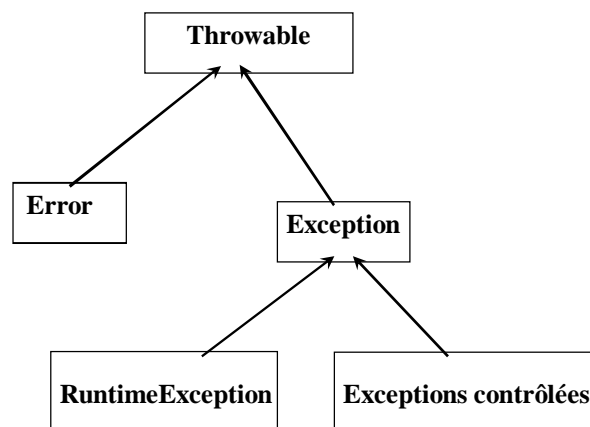
## V. Les classes d'erreurs/exceptions

- La classe **java.lang.Throwable** (classe fille de **Object**) est la superclasse de toutes les erreurs et exceptions rencontrées dans le langage de programmation Java.
- Java.lang est un package qui rassemble les classes de base de Java. Toutes les classes et interfaces de java.lang sont automatiquement importées par le compilateur.

---

El Mostafa DAOUDI- p. 229

## Arbre d'héritage des exceptions



---

El Mostafa DAOUDI- p. 230

### Quelques sous-classes de **RuntimeException** (Exceptions non contrôlées par le compilateur)

- **NullPointerException**: Tentative d'utiliser une référence null
- **ClassCastException** : Tentative de cast d'un objet dans un type incorrecte.
- **IndexOutOfBoundsException** : Un indice (sur un tableau, une chaîne) ou un intervalle défini par deux indices ont dépassé les limites inférieures ou supérieures. Ses sous-classes :
  - **ArrayIndexOutOfBoundsException**, pour les tableaux (indice négatif ou supérieur ou égal à la taille du tableau).
  - **StringIndexOutOfBoundsException**, pour les chaînes de caractères.

---

El Mostafa DAOUDI- p. 231

- **ArrayStoreException** : Tentative de stocker dans un tableau un élément qui n'est pas du type des éléments du tableau ou castable dans ce type.
- **ArithmeticException**: Une exception est survenue sur une opération arithmétique, comme une division d'un entier par zéro. )
- **NegativeArraySizeException** : Tentative de créer un tableau ou une chaîne avec une taille négative.
- **IllegalArgumentException** : Une méthode a été appelée avec un mauvais argument ou invoquée sur un mauvais objet. Sa sous-classe **NumberFormatException**
  - **NumberFormatException** : Tentative de convertir dans un type numérique une chaîne de caractères mal formatée.

---

El Mostafa DAOUDI- p. 232

## Ch. IX Interfaces et classes abstraites Abstractions et quelques classes utiles

### I. Classes abstraites

Une classe abstraite est une classe déclarée avec le mot clé **abstract**.

Elle est non instanciable . Elle sert uniquement de classe mère.

#### Exemple:

```
abstract class ClasseA {  
    ...  
}
```

Dans une classe abstraite on peut trouver:

- des champs,
- des méthodes ,
- **des méthodes abstraites.**

---

El Mostafa DAOUDI- p. 233

- Une méthode abstraite est déclarée avec le mot clé **abstract**. Dans ce cas on la déclare sans donner son implémentation (sans spécifier le corps de la méthode). On ne fournit que :
  - le type de la valeur de retour
  - et la signature (l'entête de la méthode).

#### Exemple:

```
abstract public void f(int i, float x);  
abstract public double g();
```

- Les méthodes abstraites peuvent être implémentées par les classes filles.

---

El Mostafa DAOUDI- p. 234

### Les règles

- On ne peut pas instancier une classe abstraite (on ne peut pas créer une instance (un objet) d'une classe abstraite).

Soit «ClasseA» une classe abstraite.

- On peut créer une référence sur un objet de type «ClasseA».
- Mais on ne peut pas créer un objet de type «ClasseA».

ClasseA objA; // autorisé

ClasseA objA = new ClasseA(); // n'est pas autorisé.

- Soit «ClasseB» une classe qui hérite de «ClasseA». Si «ClasseB» n'est pas abstraite alors on peut écrire:  
ClasseA a = new ClasseB();

---

El Mostafa DAOUDI- p. 235

- Une méthode static ne peut pas être abstraite. Pourquoi ??
- Une classe qui définit au moins une méthode abstraite doit être obligatoirement déclarée abstraite.
- Soit une classe fille qui hérite d'une classe mère qui définit des méthodes abstraites. Alors la classe fille doit implémenter toutes ses méthodes abstraites sinon elle doit être aussi déclarée abstraite.
- Une méthode déclarée abstraite doit obligatoirement être déclarée public.

---

El Mostafa DAOUDI- p. 236

### Intérêt des classe abstraites:

- Le principe est que la classe mère définit la structure globale d'un algorithme et laisse aux classes filles le soin de définir des points bien précis de l'algorithme.
- Par exemple quand on ne connaît pas à priori le comportement par défaut d'une opération commune à plusieurs sous-classes

---

El Mostafa DAOUDI- p. 237

## II. Interfaces

- Une classe est purement abstraite si toutes ses méthodes sont abstraites.
- Une interface est une classe purement abstraite. Elle est déclarée avec le mot clé `interface`, dont toutes les méthodes sont publiques.
  - ⇒ Une interface est une liste de noms de méthodes publiques. Dans l'interface on définit la signature des méthodes qui doivent être implémentées dans les classes qui les implémentent.
  - ⇒ Une interface est un modèle pour une classe

### Exemple d'interfaces

```
public interface Figure {  
    public void dessineFigure();  
    public void deplaceFigure(int dx, int dy);  
}
```

---

El Mostafa DAOUDI- p. 238

### Règles

- Toutes les méthodes d'une interface sont abstraites.
- Une interface n'admet aucun attribut.
- Une interface peut posséder des constantes publics

```
public interface NomInterface {  
    public static final int CONST = 2;  
}
```
- Les interfaces ne sont pas instanciables. Soit « I » une interface:

```
I obj;          // Juste  
I a = new I();   // Erreur
```
- Tout objet instance d'une classe qui implémente l'interface peut être déclaré comme étant du type de cette interface. Supposons que la classe « ClasseA » implémente l'interface « I ». Alors on peut avoir

```
I obj = new ClasseA();
```
- Les interfaces pourront se dériver

---

El Mostafa DAOUDI- p. 239

### Implémentation des Interfaces

- Les interfaces sont implémentées par des classes. Une classe implémente une interface « I » si elle déclare «**implements I**» dans son en-tête. Soit « ClasseA » qui implémente « I ».

```
public class ClasseA implements I { ... }
```

- Une classe peut implémenter une ou plusieurs interface(s) donnée(s) en utilisant une fois le mot clé **implements**.

```
public class ClasseA implements I1, I2, ... { ... }
```

- Si une classe hérite d'une autre classe elle peut également implémenter une ou plusieurs interfaces

```
public class ClasseB extends ClasseA implements I1, I3, ... {
```

---

El Mostafa DAOUDI- p. 240



### Implémente partiellement une interface

- Soit une interface «**I**» et une classe «**ClasseA**» qui l'implémente :  
**public class ClasseA implements I { ... }**
- « **ClasseA** » peut ne pas implémenter toutes les méthodes de «**I**». Dans ce cas «**ClasseA**» doit être déclarée **abstract** (elle lui manque des implémentations).
- Les méthodes manquantes seront implémentées par les classes filles de « **ClasseA** »

---

El Mostafa DAOUDI- p. 241

### III. La classe racine: classe Object

La classe **Object** ancêtre de toutes les classes.

- Toute classe autre que **Object** possède une **super-classe**.
- Toute classe hérite directement ou indirectement de la classe **Object**.
- Une classe qui ne définit pas de clause **extends** hérite de la classe **Object**.

La classe **object** fournit plusieurs méthodes qui sont héritées par toutes les classes.

- public boolean equals(Object obj)
- public String toString()
- public int hashCode()
- protected Object clone()
- public Class getClass()

Les plus couramment utilisées sont les méthodes **toString()** et **equals()**

---

El Mostafa DAOUDI- p. 242

## Comparer deux objets

### Les opérateurs == et !=

- Soient objA et objB deux objets, « objA==objB » retourne «true» si les deux objets référencent la même chose.
- **Attention:** Les tests de comparaison (== et !=) entre objets ne concernent que les références et non les attributs.

### Classe Object - equals()

public boolean equals(Object obj)

- Par défaut, retourne (this==obj). C'est-à-dire renvoie «true» si l'objet « obj » référence la même chose que l'objet qui appelle la méthode « equals » (objet courant « this »).
- Elle est prévue pour être redéfinie pour comparer les contenues.

---

El Mostafa DAOUDI- p. 243

### Exemple d'utilisation:

Considérons la classe Point et pA et pB deux instances de Point.

Point pA=new Point(5,8);    Point pB = new Point(5,8);

- pA.equals(pB) renvoie «false» car pA et pB référencent deux objet différents (appel de la méthode définie dans la classe Object).
- Si on définit la méthode « equals » dans la classe Point comme suit:

```
public boolean equals (Point p){  
    return ((this.x==p.x)&&(this.y==p.y));  
}
```

Dans ce cas pA.equals(pB) renvoie « true » car pA et pB ont le même contenu (pA et pB coïncident).

**Mais attention:** Que se passe-t-il si on a:

Object pA = new Point(2,3);

Object pB= new Point (2,3)

Que vaut : pA.equals(pB) ????

Voir TD pour la redéfinition de la méthode equals()

---

El Mostafa DAOUDI- p. 244

### Classe Object – méthode toString()

**public String toString();**

- Elle renvoie une description de l'objet sous la forme d'une chaîne de caractères de la forme:

nomDeLaClasse@AdresseMémoireDel'Objet

- Elle est prévue pour être redéfinie pour afficher le contenu.

**Exemple :** Soit pA une instance de la classe Point, alors

l'instruction:

System.out.println(pA) ;

Est équivalente à

System.out.println(pA.toString()) ;

// affiche par exemple Point@1a16869

---

El Mostafa DAOUDI- p. 245

Maintenant on la redéfinit dans la classe « Point » comme suit:

```
public String toString (){  
    return ("abscisse = "+this.x+" et ordonnée = " +this.y);  
}
```

Dans ce cas, si on a:

Point pA=new Point(10, 32);

Alors:

System.out.println(pA) ;  $\Leftrightarrow$  System.out.println(pA.toString()) ;

// appel de la méthode toString() redéfinie dans la classe « Point »

affiche : abscisse = 10 et ordonnée = 32

---

El Mostafa DAOUDI- p. 246