

# Chapitre 1 : Les objets et les classes

## Table des matières

1.Introduction.....	2
2.Description d'un objet.....	2
3.Caractéristiques d'un objet informatique.....	2
4.Représentation des objets en mémoire.....	3
5.Cycle de vie d'un objet.....	3
6.méthodes et changement d'état d'un objet.....	4
7.Introduction à la notion de classe.....	5
8.Relation entre classe et objets.....	5
9.Syntaxe Java pour l'écriture de classe et manipulation des objets.....	6
9.1Rédaction d'une classe.....	6
9.1.1Déclaration des attributs.....	6
9.1.2Déclaration des méthodes .....	7
9.1.3Surcharge de méthodes.....	7
9.2Création et manipulation des objets.....	8
9.2.1Déclaration des objets.....	8
9.2.2Création et instanciation d'un objet.....	8
9.3Manipulation des objets.....	9
9.3.1accès aux attributs d'un objet.....	9
9.3.2Appels de méthodes.....	10
9.3.3Affectation d'objets.....	12
9.4Le mot clé this.....	13
9.5 Attributs et méthodes de classe.....	15
9.5.1Attributs de classe.....	15
9.5.2méthodes de classe.....	16
9.6utilisation du mot-clé final avec des variables.....	16
10.Exemple complet.....	18

## 1. Introduction

Comparé à la programmation classique, où la programmation privilégie les procédures et les traitements par rapport aux données (que veut-on faire ?), la programmation orientée objet privilégie les données par rapport aux traitements et considère un programme comme étant un ensemble d'objets informatiques dont chacun est décrit par une liste de variables (les données) et caractérisés par des opérations qu'il connaissent (les traitements)

Dans le cas de la conception et la programmation OO, on ne pose la question que veut-ton faire ? Mais plutôt, on posera la question de quoi parle-t-on ? Ou qu'est ce qu'on manipule comme entités ? Un programme OO sera donc constitué d'un ensemble d'objets interagissant entre eux et chacun disposant d'une partie données et d'une partie traitement.

## 2. Description d'un objet

un objet se décrit par un trio ayant la forme suivante :

<objet, attribut, valeur> l'attribut est aussi appelé propriété

Exemples

- l'objet « arbre1 » est décrit par : <arbre1, type, sapin>, <arbre1, taille, 5m>
- l'objet « voiture1 » est décrit par : <voiture1, couleur, rouge>, <voiture1, marque, toyota>, <voiture1, modèle, yaris>
- l'objet « passant1 » est décrit par : <passant1, taille, grand>, <passant1, age, 50>

Dans les langages informatiques OO, on stocke et on manipule les objets en mémoire sous forme de couples (attribut, valeur).

## 3. Caractéristiques d'un objet informatique

un objet informatique est caractérisé par :

- **un identifiant unique:** valeur unique et invariante qui identifie l'objet. Un objet informatique étant stocké en mémoire, l'adresse mémoire représente son identifiant unique.
- **un état :** l'état d'un objet est l'ensemble des valeurs des attributs de cet objet à un moment donné. Un objet évolue et peut changer d'état durant son existence, ce changement se réalise par la modification de la valeur d'au moins d'un seul de ses attributs.
- **un comportement :** le comportement d'un objet est définie par le traitement qu'on peut effectuer sur les attributs de cet objet. Il se réalise par un ensemble de fonctions ou procédures appelées méthodes. Les méthodes d'un objet permettent la définition de la manière de changer l'état d'un objet.

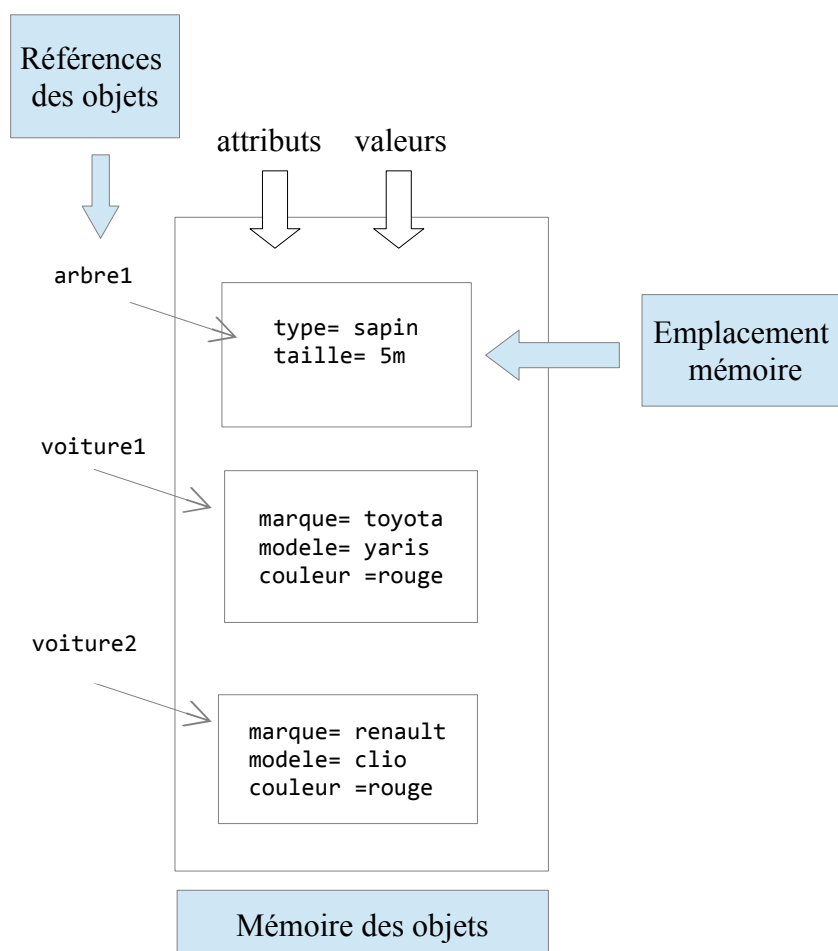
## 4. Représentation des objets en mémoire

Lors de la création d'un objet, un espace mémoire sera alloué et réservé. Il contiendra les attributs de cet objet. Les attributs sont généralement de types primitifs (entier, réel, caractères...) et donc les tailles des valeurs à encoder sont connues à l'avance, par conséquent, la taille alloué pour l'espace de stockage d'un objet est aussi connu d'avance.

**L'adresse** mémoire de chaque espace réservé pour **un objet**, représente son **identifiant unique**.

Dans les programmes, **un objet est accessible par un variable spéciale** qui contiendra l'adresse mémoire où cet objet est stocké. Cette variable est appelé **référence de l'objet**.

Exemple



## 5. Cycle de vie d'un objet

Les objets existent mémoire durant l'exécution d'un programme (comme c'est le cas des variables). Tout objet passe par trois étape lors de son existence :

1. **création de l'objet** : dans cette étape, l'espace mémoire de l'objet est alloué puis les valeurs des attributs sont initialisé et enfin l'adresse mémoire est retourné sous forme de référence de l'objet.

2. **changement d'état éventuel** : l'objet évolue et peut changer d'état plusieurs fois
3. **destruction de l'objet** : l'objet arrive en fin de vie lorsqu'il n'est plus référencé. Si son identifiant unique qui est sa référence ne contient plus son adresse mémoire alors l'objet n'est plus accessible et donc l'objet est dit détruit.

## 6. méthodes et changement d'état d'un objet

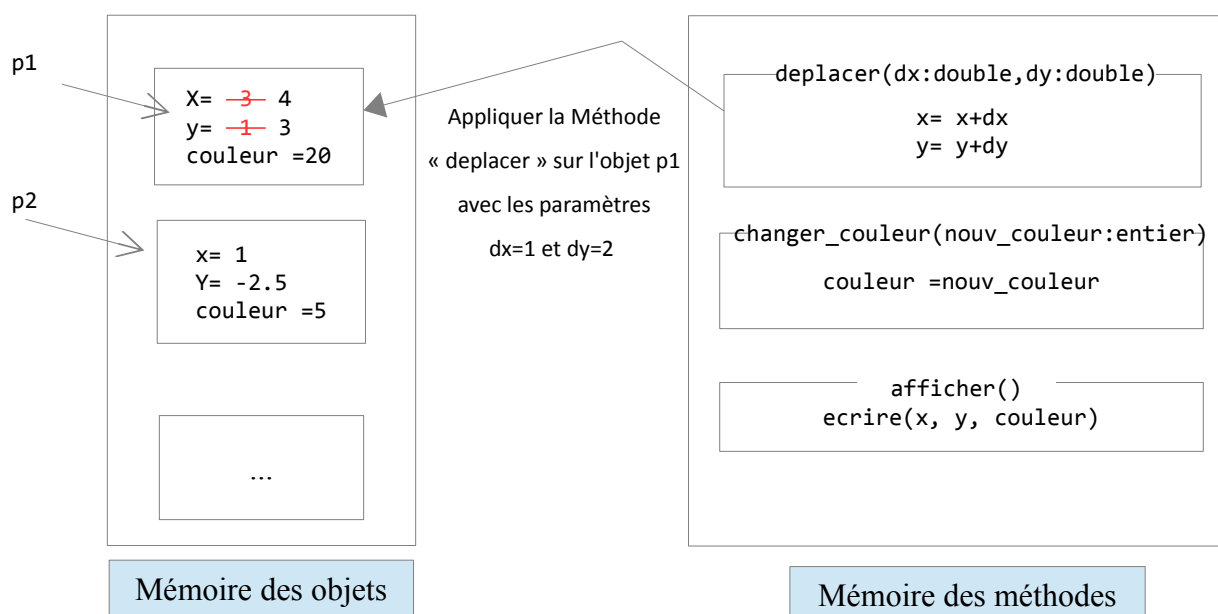
Le changement d'état des objets va permettre de les rendre vivants et dynamique, et ainsi il seront capable de simuler des objets de la réalité.

On a vu précédemment que seules des opérations spéciales, qu'on appelle dans la POO méthodes, sont autorisés à manipuler les attributs d'un objet, il sont donc les seules responsables de changer l'état d'un objet.

Dans la mémoire, en parallèle du stockage des objets avec leurs attributs, les méthodes occupent aussi un espace mémoire séparé.

L'exemple suivant montre une représentation des attributs et des méthodes en mémoire afin de mettre en évidence la différence entre l'implémentation des traitements entre le mode la programmation procédurale et la POO.

Soit un objet représentant un Point et qui est référencé en mémoire par « p1 ». il est défini par les attributs suivants : « x » et « y » de type double pour les coordonnées et « couleur » un entier pour la couleur. Un objet point peut être déplacé sur le plan avec "dx" et "dy", il peut changer de couleur et il peut être affiché.



On remarque bien que les méthodes « déplacer », « changer\_couleur » et « afficher » utilisent les attributs "x", "y" et "couleur" sans que ces derniers ne soient passés en paramètres ou déclarés comme variables locales.

Les questions qui se posent sont les suivantes :

Comment relier un ensemble de méthodes à un ensemble d'attributs ? Comment la fonction « déplacer » va reconnaître la valeur « x » et la valeur « y » en tant qu'attributs de l'objet « p1 » ?

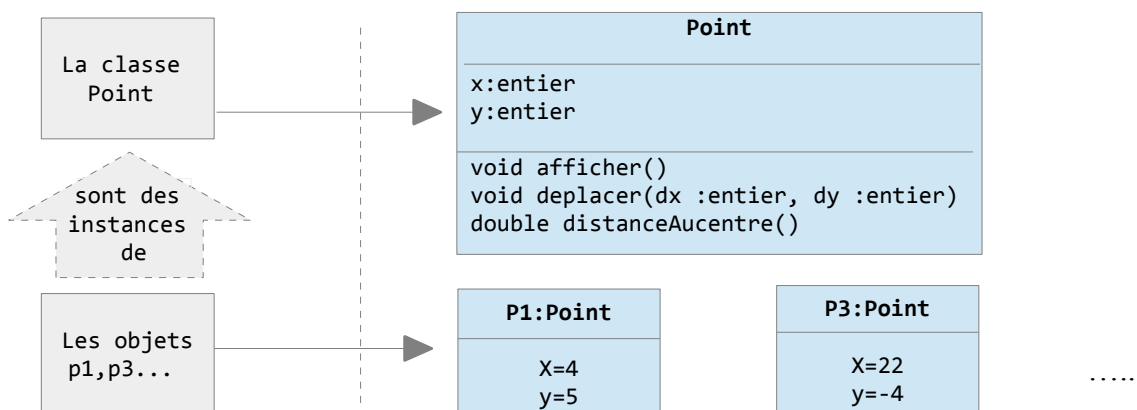
## 7. Introduction à la notion de classe

Pour que la méthode " déplacer " reconnaisse les attributs x et y il faut assurer que "déplacer" soit rattaché à des objets de type Point, la meilleure manière d'associer les méthodes et les objets sur lesquelles elles s'appliquent est de regrouper la déclaration des attributs définissant la structure des objets et les méthodes dans une seule unité de programmation qu'on appellera **classe**

La POO définit donc, une nouvelle structure : la Classe, qui permet de définir la description de tous les objets qui la prennent comme modèle (partie attribut) mais aussi elle permet de définir la dynamique de ces objets en englobant les méthodes qui regroupent à leur tour les instructions portant sur les attributs des objets comme si ils sont passés en paramètres de ces méthodes

## 8. Relation entre classe et objets

- Une classe est un modèle (sorte de moule) qui permet de définir les objets :
  - ayant la même description : ensemble d'attributs
  - ayant le même comportement: ensemble de méthode
  - ayant la même sémantique : sens commun
- l'ensemble des attributs et des méthodes d'une classe sont appelés membres d'une classe
- l'objet est pour une classe ce que représente une variable pour un type
- l'objet est une instance d'une seule classe
- une classe peut instancier plusieurs objets



## Exemples

Le tableau suivant propose quelque classe (1ere colonnes) avec des objets instances de ces classes (2eme colonne)

Classe	Objets (instances)
Humain	Ali, Nadia, Mohamed...
Voiture	Toyota Yaris, Renault Clio, Peugeot 206, ...
Matière	Algèbre, Algo, Analyse....

## 9. Syntaxe Java pour l'écriture de classe et manipulation des objets

### 9.1 Rédaction d'une classe

Une classe java a la syntaxe suivante (sans notion d'héritage)

```
[modificateurs] class nom_de_classe
{
    //corps_de_classe :
    // déclaration des attributs
    // déclaration des méthodes
}
```

- **Modificateurs** : optionnel, il s'agit d'un ensemble de mots clés indiquant la nature de la classe et sa portée

Les modificateurs, seront traiter un peut plus loin dans le cours.

- **class** : indiquant qu'on déclare une classe
- **nom\_de\_classe** : un nom significatif qu'on utilisera pour référencer la classe
- **corps\_de\_classe** : représente l'ensemble des attributs (champs ou données) et des méthodes.

#### 9.1.1 Déclaration des attributs

les attributs se déclarent comme des simples variables, sauf qu'avec les attributs on peut utiliser un modificateur

**Syntaxe :**

[ <i>modificateur</i> ] <i>type nomAttribut</i> [=valeur_initialisation]	
<i>Modificateur</i>	optionnel, il permet de spécifier la visibilité de l'attribut (on revient sur cette notion ultérieurement)
<i>type</i>	peut être de type primitif (int, double, ...) ou de type référent (tableau, classe, ....)
<i>nomAttribut</i>	identificateur de l'attribut

Exemples

```
public double x=6.2 ;
int couleur ;
private Point p ;
```

Un attribut qui n'est pas initialisé explicitement, prendra une valeur par défaut :

- les attributs de type *numérique* ou caractère prendront la valeur *0*
- les attributs de type *boolean* prendront la valeur *false*
- les attributs de type référence prendront la valeur *null*

### 9.1.2 Déclaration des méthodes

#### Syntaxe

<pre>[modificateur] typeRetour nomMethode (liste_Paramètres) {     // Corps de la méthode : ensemble d'instructions }</pre>	
<i>Modificateur</i>	optionnel, il permet de spécifier la visibilité de la méthode (on revient sur cette notion ultérieurement)
<i>typeRetour</i>	c'est le type de retour de la méthode. Si la méthode ne retourne rien le type Retour est remplacé par le mot « void ».
<i>nomMethode</i>	identificateur de la méthode
<i>listeParamètres</i>	liste (éventuellement vide) des paramètres. On spécifie les variables qu'on souhaite passé à la méthode lors de son appel.

### 9.1.3 Surcharge de méthodes

Sous le langage Java, on peut utiliser le même nom pour plusieurs méthodes tant que leurs listes d'arguments sont différentes. Cette technique s'appelle surcharge (overloading en anglais).

Des listes d'arguments différentes sont des listes dont le nombre d'arguments est différent ou des listes dont le nombre d'arguments est le même mais dont au moins un des arguments est de types différents dans chaque liste.

Ainsi `f(int a, int b, int c)` et `f(int a, int b)` sont des méthodes dont le nombre d'arguments est différent et `g(int a, int b, int c)`, `g(int a, float b, int c)` et `g(float a, float b, float c)` sont des méthodes possédant le même nombre d'arguments mais dont au moins un des arguments est de types différents dans chaque liste.

Le nom d'une méthode et sa liste d'arguments s'appelle sa **signature**.

Exemple :

```
public class Surchage
{ public static void main(String args[ ])
  { int i=5,j=6;
    carre(i);
    carre(j);
    carre(i,j);
  }

  static void carre(int n)
  { System.out.println(n*n);
  }

  static void carre(int m,int n)
  { System.out.println(n*n+"\t"+m*m);
  }
}
```

## 9.2 Création et manipulation des objets

### 9.2.1 Déclaration des objets

Une classe est un type de données : à la suite d'une déclaration classe Point , on pourra déclarer des variables de type Point, c'est-à-dire des variables dont les valeurs sont des instances de la classe Point. Exemple :

```
Point p; // déclaration d'une variable Point
```

Par défaut, à la déclaration une référence vaut null, cela veut dire, qu'elle ne « pointe » sur aucun objet, et donc :

```
Point p;<==>Point p=null;
```

### 9.2.2 Création et instanciation d'un objet

Comme on a déjà dit dans une section précédente, la création d'un objet consiste à **allouer son espace mémoire** et puis **initialiser les valeurs des attributs**.

Sous Java, pour allouer l'espace nécessaire pour un objet on utilisera le mot clé **new**, et pour initialiser les valeurs des attributs on utilisera une méthode particulière appelé **constructeur**

Un constructeur à caractéristiques suivantes :

- un constructeur doit avoir le même nom que la classe, il n'a aucun type de retour. Donc un constructeur est une méthode ayant la forme **NomClass(liste\_de\_parametres)**
- les valeurs des paramètres passés à un constructeurs servent à initialiser les attributs de l'objet



- dans une classe on peut avoir plusieurs constructeurs de signatures différentes (surcharge de méthodes)
- toute classe JAVA possède au moins un constructeur, si une classe ne définit pas explicitement de constructeur, un constructeur par défaut sans paramètres et qui n'effectue aucune initialisation particulière est invoqué

La syntaxe de création d'un objet est donc la suivante :

```
Refobjet= new NomClasse(liste_de_parametre);
```

Exemple de constructeurs :

pour la classe **Point**

```
class Point{
    double x,y ;
    Point(){ x=3 ; y=4 ;}
    ...
}
```

pour la classe **personne**

```
class Personne{
    String nom,prenom ;
    personne(String nomPer,String prenomPer){
        nom=nomPer ;
        prenom=prenomPer ;
    }
    ...
}
```

Exemples d'instanciation d'objet :

```
Point p;// déclaration d'une référence p
p= new Point() ;
Personne pers ;
pers= new Personne("lyes","Djaafri")
```

## 9.3 Manipulation des objets

### 9.3.1 accès aux attributs d'un objet

pour accéder aux attributs d'un objet on utilise une notation pointée :

```
nom_de_objet.nom_de_attribut
```

Exemple

```
Point p1; p1 = new Point();
```

```
Point p2 = new Point();
p1.x = 10;
p1.y = 10;
p2.y=p2.y+ 14;
```

### 9.3.2 Appels de méthodes

Dans la POO, pour appeler une méthode sur une objet à l'extérieure de la classe, en utilise une notion fondamentale de la POO qui est l'**envoi de message**.

Pour appeler la méthode **afficher()** sur l'objet **p1**, on dit qu'il faut "**envoyer à l'objet p1 le message "afficher avec une liste de paramètre vide"**"

On dit que l'objet **p1** est **destinataire du message**.

Sous Java l'implémentation de la notion d'envoi de message est l'utilisation de la notation pointée :

```
nom_de_objet.nom_methode(liste_de_parametres)
```

#### Exemple

le code suivant représente la déclaration d'une classe "Point", qui définit un point dans plan par ses coordonnées, "x" représentant son ordonnée et "y" son abscisse

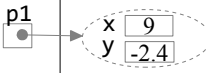
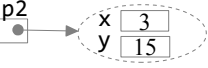
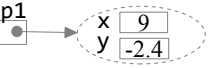
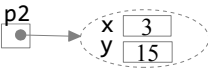
```
class Point {
    // déclaration des attributs
    double x;
    double y;

    //déclaration des méthodes
    void deplacer(double dx,double dy){
        x = x + dx;
        y = y + dy;
    }

    void afficher(){
        System.out.println(" (" + x + " , " + y + " ) " );
    }
}
```

soit le code suivant à l'extérieur de la classe Point :

N° ligne	Lignes de codes	Valeurs de x et y dans les méthodes	Mémoire des objets
1	Point p1,p2 ;		p1 → p2 →
2	p1=new Point(4,-1.4);		p1 → x: 4, y: -1.4
3	p2=new Point(-5,12);		p2 → x: -5, y: 12

4	<code>p1.deplacer(5,-1);</code>	La méthode "deplacer" est appelée avec <b>p1</b> , donc les valeurs de <b>x</b> et <b>y</b> utilisées sont celles de p1 : <b>p1.x=4</b> et <b>p1.y=-1.4</b>	$x=4+5$ et $y=-1.4+(-1)$ 
5	<code>p2.deplacer(2,3);</code>	La méthode "deplacer" est appelée avec <b>p2</b> , donc les valeurs de <b>x</b> et <b>y</b> utilisées sont celles de p2 : <b>p2.x=-5</b> et <b>p2.y=12</b>	$x=-5+2$ et $y=12+3$ 
6	<code>p1.afficher();</code>	La méthode "afficher" est appelée avec <b>p1</b> , donc les valeurs de <b>x</b> et <b>y</b> utilisées sont celles de p1 : <b>p1.x=9</b> et <b>p1.y=-2.4</b>	
7	<code>p2.afficher();</code>	La méthode "afficher" est appelée avec <b>p2</b> , donc les valeurs de <b>x</b> et <b>y</b> utilisées sont celles de p2 : <b>p2.x=3</b> et <b>p2.y=15</b>	

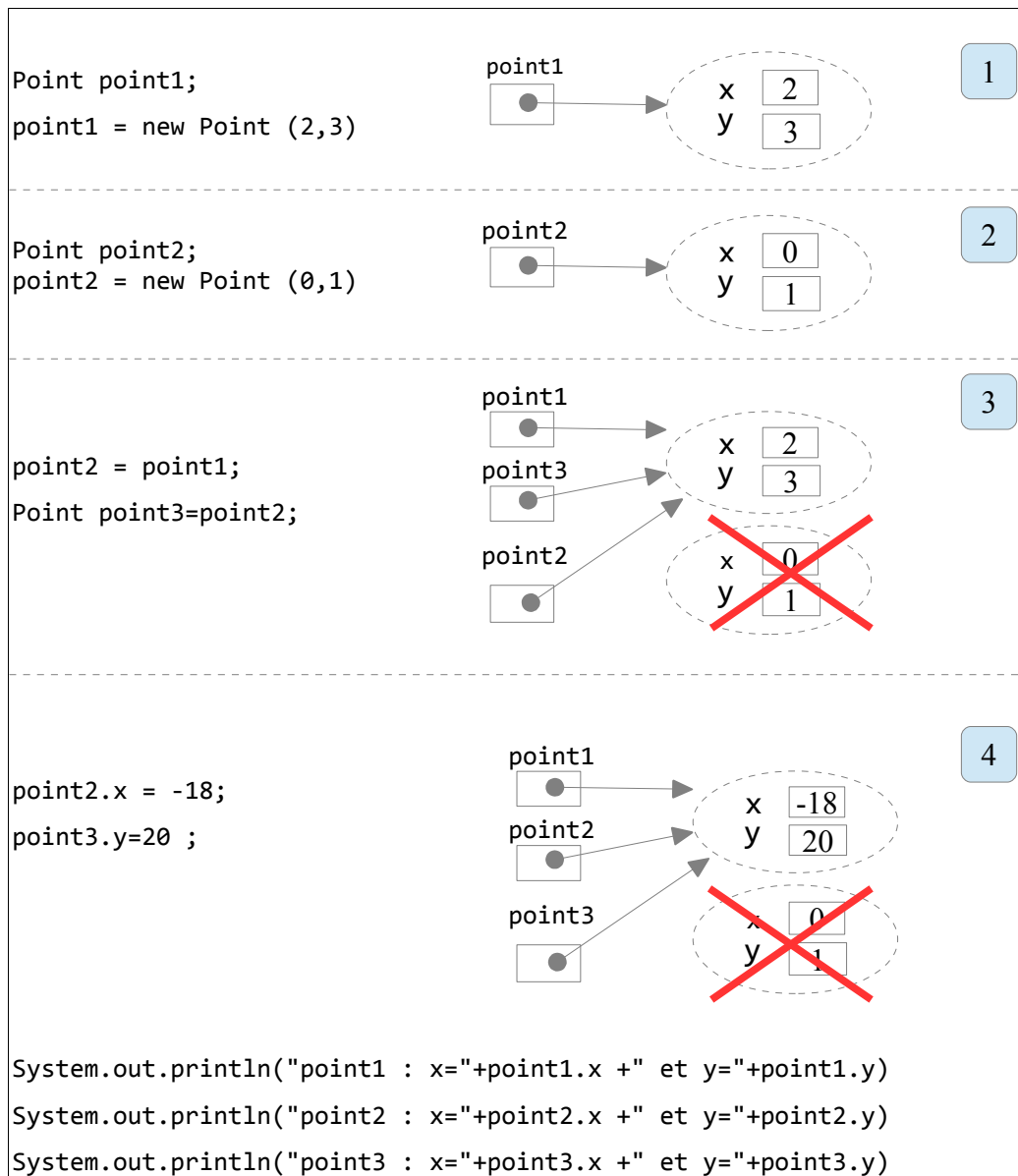
## Conclusion

Lors d'un appel de méthode avec un objet donné, les attributs de cet objet sont connus dans cette méthode. Les valeurs de ces attributs sont celles de l'objet qui appelle la méthode (l'objet récepteur du message)

### 9.3.3 Affectation d'objets

L'affectation entre objets, n'est en fait, qu'une affectation entre références.

soit le code suivants :



#### Explication

- La 1ere partie du code permet de créer un nouvel objet de type Point référencé par point1 et ayant les valeur x= 2 et y=3
- De la même manière, la partie2 du code permet de créer un nouvel objet de type Point référencé par point2 et ayant les valeur x= 0 et y=1
- Dans la 3eme partie l'affectation point2=point1 ; représente une affectation entre références, l'adresse contenu dans point1 est maintenant contenu aussi dans point2 ce qui

implique schématiquement que point1 et point2 pointent sur le même objet (x=2, y=3) et que l'objet pointé par point2 auparavant est maintenant non accessible parce ce qu'il est sans référence.

Aussi dans la 3eme partie du code, on déclare une référence point3 et on lui affecte point2, cela implique que point3 pointe sur le même objet que point2. *Il n'y a donc pas création un nouvel objet point3 à partir d'une simple affectation de référence.*

- La 4eme partie du code , l'instruction `point2.x = -18;` modifie l'a valeur de l'attribut x de l'objet référencé par `point2`, puis l'instruction `point3.y = 20;` modifie l'a valeur de l'attribut y de l'objet référencé par `point3`.

Les instructions d'affichage des attributs des références pointées par `point1`, `point2` et `point3` donnent le même résultat et l'écran affichera :

```
point1 : x=-18 et y=20  
point2 : x=-18 et y=20  
point3 : x=-18 et y=20
```

## Conclusion

L'affectation entre objets ne crée pas un nouvel objet mais elle crée une nouvelle référence qui pointe sur le même objet.

## 9.4 Le mot clé this

Dans le corps d'une méthode d'instance ou d'un constructeur, le mot clé **this** désigne l'objet courant. L'objet courant est l'objet à travers lequel la méthode d'instance a été appelé.

Vous pouvez vous référer aux membres d'instance (propriétés ou méthodes) de l'objet courant à partir d'une méthode d'instance ou d'un constructeur en utilisant le mot clé **this**.

La valeur de **this** est une référence constante sur l'objet. La référence **this** ne peut pas être affectée, par contre elle peut être retournée.

### Utilisation de this

lever l'ambiguïté sur l'utilisation des attributs

Exemple

```
class Point {  
    double x;  
    double y;
```

```

Point() {
    this.x = 0; this.y = 0; // utilisation de this facultative
}

Point(double x, double y) {
    /*-- utilisation nécessaire de this pour
       différencier les attributs des paramètres
    --*/
    this.x = x; this.y = y;
}
...
}

```

### this en tant que constructeur

Si on utilise la surcharge de constructeur, on peut également utiliser la notation **this(liste\_de\_paramètres)**, à partir d'un constructeur, pour appeler un autre constructeur de la même classe. Le constructeur appelé doit respecter la **liste\_de\_paramètres**.

Pour jouer ce rôle, **this(liste\_de\_paramètres)** désigne le constructeur de la même classe ayant la même liste d'arguments.

Exemple de deux programmes équivalents, permettant la surcharge de constructeurs de la classe Point. Le premier programme utilise une syntaxe conventionnelle, le second utilise la notation **this(..)**

```

class Point{
    double x,y ;

    Point(){
        this.x=0; this.y=0 ;
    }

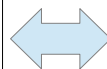
    Point(double x){
        this.x=x; this.y=x ;
    }

    Point(Point p){
        this.x=p.x; this.y=p.y ;
    }

    Point(double x,double y){
        this.x=x; this.y=y ;
    }
    ...
}

```

Déclaration de constructeurs sans utiliser la notation this()



```

class Point{
    double x,y ;

    Point(){
        this(0) ;
    }

    Point(double x){
        this(x,x) ;
    }

    Point(Point p){
        this(p.x,p.x) ;
    }

    Point(double x,double y){
        this.x=x; this.y=y ;
    }
    ...
}

```

Appel du constructeur Point(double x)

Appel du Constructeur Point(double x, double y)

Même déclaration de constructeurs mais avec la notation this()

### Remarques

- l'appel de **this** en tant constructeur ne peut se faire qu'à l'intérieur d'un constructeur

- l'utilisation de **this** en tant que constructeur **est un appel de méthode**, et il n'a pour rôle que la réutilisation d'un code déjà existant, pour cela il doit se faire sans l'opérateur **new**.
- L'appel d'un constructeur avec la notation **this(...)** doit être toujours la première instruction du corps du constructeur.

## 9.5 Attributs et méthodes de classe

Par défaut lorsque l'on crée des objets d'une même classe, chacun de ces objets possède ses propres **attributs d'instance**. Cependant, Java permet aussi de définir des attributs qui sont partagés par tous les instances (objets) d'une même classe,

De même, java permet de définir des méthodes de classe qui peuvent être appelées indépendamment de tout objet de la classe

### 9.5.1 Attributs de classe

Soit le code suivant

```
class A { double x ; }
...
A a1=new A();
A a2=new A();
...
A an=new A();
```

Mémoire :

a1.x → 0.0   a2.x → 0.0   ...   an.x → 0.0

Dans ce code, l'attribut x est un attribut d'instance, il existe une variable x pour chaque objet instance (a1, a2, ...an) de la classe A

La question qui se pose: Dans la classe A, comment déclarer une variable qui soit indépendante de tous les objets instances de la classe A ?

Java permet de répondre à cette question, en utilisant le mot clé **static**. Les attributs déclarés comme **static** on les appelle **attributs de classe** ou **attributs statiques**

Exemple : on veut modifier le code précédent pour qu'il nous fournit le nombre d'objets crée dans la classe A.

```
class A {
    double x ; // attribut d'instance (par défaut)
    static int nbr=0 ; //attribut de classe
    A(){
        nbr++ ;
        System.out.println("nouvel objet créé,nombre total d'objet :"+nbr) ;
    }
}
```

Mémoire :

A.nbr → 2  
a1.x → 0.0   a2.x → 0.0

```

    }
}

class Test{
    public static void main(String[] args){
        A a1=new A();
        A a2=new A();
        System.out.println("nombre d'objet de
la
        classe A:"+A.nbr) ;
        /* pour invoqué la variable statique nbr
on utilise A.nbr, mais on peut utiliser
a1.nbr ou a2.nbr*/
    }
}

```

Exécution :

```

nouvel objet créé, nombre total d'objet :1
nouvel objet créé, nombre total d'objet :2
nombre d'objet de la classe A :2

```

### 9.5.2 méthodes de classe

De la même manière que les attributs de classe, java permet de créer des méthodes de classe qui réalisent des traitements indépendamment de tout objet de la classe. La déclaration d'une méthode de classe doit être précédée par le mot clé **static**

Règles et remarques :

- une méthode de classe ne pourra agir sur l'invocation d'un attribut d'instance ou l'appel d'une méthodes d'instance n'est pas autorisé dans le corps d'une méthode classe. Par conséquent, le mot clé **this** n'est pas autorisé. Une erreur de compilation sera déclenché, le cas échéant.
- les attributs de classe (bien sûr, sans compter le paramètres et les variables locales) peuvent être utilisés dans le corps d'une méthode de classe.
- Les variables et les méthodes de classe appartiennent à la classe elle-même, on peut ainsi les utiliser sans créer des objets.
- De nombreuses classes de bibliothèque standard java, ont que des membres statiques, par exemple la classe **java.lang.Math** :

```

System.out.println(Math.PI) ;
double a=Math.abs(x)

```

### 9.6 utilisation du mot-clé final avec des variables

Quand le mot clé **final** est utilisé avec une variable, cela signifie que cette variable **ne peut plus être modifiée après son initialisation**, et doit être **obligatoirement** initialisée une fois (et une seule fois).

Remarques :

- l'utilisation du mot clé final a pour effet de rendre une variable constante.
- Java n'exige pas que la déclaration d'une variable avec final soit suivie directement par son



initialisation.

- Java donne la possibilité de différer l'initialisation dans une instruction ultérieure
- Le comportement du mot clé final devant une objet (une référence), n'interdit pas la modification de ses attributs. Par contre, elle lui interdit l'affectation d'un autre objet.

Exemple :

```
final Point p1 ;p1=new Point(1,3) ;
Point p2 ;p1=new Point(-2,3.5) ;
p1.x=2 ;//ne pose pas de problème malgré que p1 est final
p1=p2 ;// erreur de compilation : la référence p1 est déjà initialisée
```

Les cas de figure suivants sont possibles avec le mot clé final avec des variables :

### 1. cas de variable local à une méthode

```
void methode() {
    final double var = 0.2;
    var = 0.5 ; // interdit et provoque une erreur de compilation.
}
```

La valeur de la variable `var` ne pourra pas être modifiée.

### 2. cas de variable paramètre d'une méthode

```
void methode( final double param1) {
    double var = 4.0;
    param1 = var/2 ; //affectation interdite et provoque une erreur de
compilation.
}
```

Le paramètre `param1` ne pourra pas être modifié

### 3. cas d'un attribut d'instance

Si un attribut d'instance est déclaré avec le mot clé final, cela implique deux contraintes :

- il doit être initialisé obligatoirement :
  - soit lors de sa déclaration ;
  - soit dans un constructeurs ;
- si l'initialisation doit se faire dans un constructeurs, et que la classe possède plusieurs constructeurs, alors l'initialisation doit se faire dans chaque constructeur.

```
public class Test {
    final double var; //déclaration d'un attribut d'instance final
```

```

    final double a=2; //déclaration et initialisation d'un attribut d'instance
final
    Test() {
        var=2.0; //l'initialisation de var est obligatoire
        a=2; //erreur de compilation : a est déjà affecté
    }
    Test(int valeur) { var=valeur; //l'initialisation de var est obligatoire }
    Test(double x) { var=x; //l'initialisation de var est obligatoire }

    public static void main(String [] args) {
        Test t1=new Test(2.8);
        t1.var=22; //erreur de compilation : var est déjà affecté
    }
}

```

#### 4. cas d'une variable de classe (statique)

Une variable statique déclarée avec **final** doit être initialisée. L'initialisation est interdite dans les constructeurs.

```
private static final int X = 5;
```

Dans la classe **Math**, de la bibliothèque standard java, la constante PI est déclarée **static** et **final** :

```
public static final double PI=3.141592653589793;
```

## 10. Exemple complet

Le code suivant représente deux classe, la première : la classe Point qui représente un point dans un plan, la deuxième : la classe Test qui permet de tester la classe Point, en créant des instances (objets de types Point) et en réalisant quelques traitements par le biais d'appel de méthodes.

```

/*-----*/
/*-----classe Point-----*/
/*-----*/

class Point {
//déclaration des propriétés
    private double x,y;

// déclaration de constructeurs

    public Point() {
        x = 0; y = 0;
        //ou faire un appel au code d'un autre constructeur:
        //this(0,0) ;
    }

    public Point(double a) {
        x = a; y = 0;
    }
}

```

```
//ou faire un appel au code d'un autre constructeur:
//this(a,0) ;
}

public Point(double a, double b) {
    x = a; y = b;
}

public Point(Point p) {
    x = p.x; y = p.y;
    //ou faire un appel au code d'un autre constructeur:
    //this(p.x,p.y) ;
}

//déclaration des méthodes
public void deplacer(double dx, double dy) {
    x += dx ; y += dy;
}

// surcharge de la méthode « deplacer »
public void deplacer(double dx) {
    x += dx;
}

//la méthode afficher affiche un objet de Point
public void afficher() {
    System.out.println(" Point : [x= " + x + " , y= " + y + " ] ");
}

/*la méthode distance calcule la distance un objet
de type Point et le centre (0,0)
*/
public double distance(){
    return Math.sqrt(x*x+y*y);
    //ou faire un appel à la méthode statique :
    //Point.distance(this,new Point(0,0) ;
    //ou encore un appel à la méthode d'instance :
    //return this.distance(new Point(0,0)) ;// this est facultatif
}

public double distance(Point p){
    double d;
    d=Math.sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y));
    return d ;
    //ou faire un appel à la méthode statique :
    // return Point.distance(this,p) ;
}

public static double distance(Point p1,Point p2){
    double d;
    d=Math.sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
    return d;
}

public Point PointPlusProcheDuCentre(Point p){
    if (this.distance()>p.distance()) then return p;
    else return this ;
}
}
```

```
/*-----*/
/*-----classe TestPoint-----*/
/*-----*/

public class TestPoint {

    public static void main(String[] tabParam) {
        //création d'un objet point p1 par le constructeur sans paramètres
        Point p1;
        p1 = new Point();
        p1.deplacer(2,5.8);

        //création d'un objet point p2 par 2eme constructeur
        Point p2 = new Point(3);

        //création d'un objet point p3 par 3eme constructeur
        Point p3;
        p3 = new Point(3, 4);

        //appel de la méthode « deplacer » sur l'objet p3
        p3.deplacer(2, -6.2);
        //affichage de p3
        System.out.println("p3 :"); p3.afficher();
        //appel de la méthode « deplacer » sur l'objet p2
        p2.deplacer(-5);
        //affichage de p2
        System.out.println("p2 :"); p2.afficher();

        //appel de la méthode distance sur l'objet p2
        double dis1=p2.distance();
        System.out.println("la distance entre p2 et le centre (0,0)
est :"+dis);

        //appel de la méthode PointPlusProcheDuCentre sur l'objet p2

        System.out.println("Le point le plus proche du centre est :");
        p2.PointPlusProcheDuCentre(p1).afficher() ;

    }
}
```