



P.O.O. (Programmation Orientée Objet)

CHOUTI Sidi Mohammed

Cours pour L2 en Informatique
Département d'Informatique
Université de Tlemcen
2017-2018

1. Introduction à la Programmation Orientée Objet
2. Classes et objets
3. Héritage et polymorphisme
4. Interface et implémentation
5. Interface graphique et Applet
6. ...

Fichier "Carre.java"

```
class Carre{
    Point2D centre;
    double longueur;

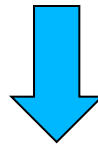
    Carre(){ centre=new Point2D(); longueur=1; }
    Carre(Point2D c, double l){ centre=c; longueur=l; }

    void deplacer(Vecteur2D vecteur) {
        centre.x += vecteur.x; centre.y += vecteur.y; }
    void deplacer(double x,double y) {
        centre.x += x; centre.y += y; }
    void deplacerH(double x) { centre.x += x; }
    void deplacerV(double y) { centre.y += y; }
    void afficher(){ System.out.print("Objet Carre :\n\tcentre : ");
        centre.afficher();
        System.out.println("\n\tlongueur : " + longueur); }
}
```

le **code** de la classe *Cercle* et celui de la classe *Carre*, est presque **identique**

Factoriser le code

un moyen de regrouper les parties
de code identiques



ce que propose le concept
d'héritage

L'héritage est un principe propre à la programmation orientée objet, permettant de créer une nouvelle classe à partir d'une classe existante. Appelé aussi dérivation de classe provient du fait que la classe dérivée ou filie (la classe nouvellement créée) contient les attributs et les méthodes de sa superclasse ou mère (la classe dont elle dérive).

En java

Class Cercle **extends** Forme

ou

Class Carre **extends** Forme

1- L'héritage de classe porte à la fois sur l'état et le comportement

- Une sous-classe hérite de tous les attributs de sa superclasse
- Une sous-classe hérite de toutes les méthodes de sa superclasse
- Une sous-classe hérite de toutes les propriétés statiques de sa superclasse

2- en Java, si vous ne spécifiez pas de lien d'héritage, la classe en cours de définition hérite alors de la classe **Object**.

Fichier « Forme.java »

```
class Forme{
    Point2D centre;

    Forme(){ centre=new Point2D(); }
    Forme(Point2D c){ centre=c; }

    void deplacer(Vecteur2D vecteur) {
        centre.x += vecteur.x; centre.y += vecteur.y; }
    void deplacer(double x,double y) { centre.x += x; centre.y += y; }
    void deplacerH(double x) { centre.x += x; }
    void deplacerV(double y) { centre.y += y; }

    void afficher(String nature){
        System.out.print("Objet " + nature+" :\n\tcentre : ");
        centre.afficher();    }
    }
```


Fichier « Carre.java »

```
class Carre extends Forme {  
    double longueur;  
  
    Carre(){super(); longueur=1; }  
    Carre(Point2D c,double l){ super(c);  longueur=l; }  
  
    void afficher(){  
        super.afficher("Carre");  
        System.out.println("\n\tlongueur : " + longueur);  
    }  
}
```

Fichier « Cercle.java »

```
class Cercle extends Forme {  
    double rayon;  
  
    Cercle(){ rayon=1; }  
    Cercle(Point2D c,double l){ super(c); rayon=l; }  
  
    void afficher(){  
        super.afficher("Cercle");  
        System.out.println("\n\trayon : " + rayon);  
    }  
}
```

- Supprime les **redondances** dans le code.
- On peut très facilement **rajouter, après coup, une classe**, et ce à **moindre coup**, étant donné que l'on peut réutiliser le code des classes parentes.
- Si vous n'aviez pas encore modélisé un comportement dans une classe donnée, et que vous vouliez maintenant le rajouter, une fois l'opération terminée, ce comportement sera alors directement utilisable dans l'ensemble des sous-classes de celle considérée.

super sert à accéder les définitions de classe au niveau de la classe parente de la classe considérée

this sert à accéder à la classe courante.

Règles sur l'utilisation des constructeurs de la classe mère

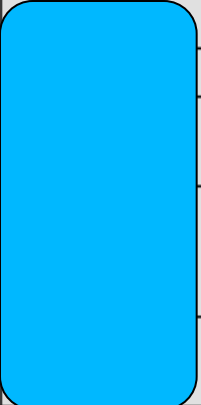
Règle 1 : si vous invoquez `super()`, cela signifie que le constructeur en cours d'exécution passe la main au constructeur de la classe parente pour commencer à initialiser les attributs définis dans cette dernière. Ensuite il continuera son exécution.

Règle 2 : un appel de constructeur de la classe mère peut uniquement se faire qu'en première instruction d'une définition de constructeur.

Règle 3 : si la première instruction d'un constructeur ne commence pas par le mot clé **super** le constructeur par défaut de la classe mère est appelé.

Règle 4 : si vous invoquez `this()`, le constructeur considéré passe la main à un autre constructeur de la classe considérée.

Exemples

fichier "Classe1.java"	fichier "Classe2.java"
<pre>class Classe1 { Classe1() { System.out.println("Classe1"); } Classe1(int val) { this(); System.out.println(val); } }</pre>	<pre>class Classe2 extends Classe1 { Classe2() { super(5); System.out.println("Classe2"); } Classe2(int val) { System.out.println(val); } }</pre>
exemple de création d'objet	résultats
new Classe1();	
new Classe1(3);	
new Classe2();	
new Classe2(2);	

Le polymorphisme

Un langage orienté objet est dit **polymorphique**, s'il offre la possibilité de pouvoir (**percevoir**) un objet en tant qu'instance de classes **variées**.

fichier "A.java"	fichier "B.java"
<pre>class A { ... }</pre>	<pre>class B extends A { ... }</pre>
<pre>B b=new B(); A a=b; // Utilisation du polymorphisme (A) b; // On utilise le casting</pre>	

Le polymorphisme

fichier "A.java"	fichier "B.java"
<pre>class A { void m() { System.out.println("Mother"); } }</pre>	<pre>class B extends A { void m() { System.out.println("Son"); } }</pre>
<pre>public class Polym { public static void main(String args[]){ B b=new B(); A a=b; a.m(); // Utilisation du polymorphisme } }</pre>	

- Racine de l'arbre d'héritage des classes : `java.lang.Object` □
- Héritée par toutes les classes sans exception
- Object n'a pas de variable d'instance ni de variable de classe
- Object fournit plusieurs méthodes
- Couramment utilisées sont les méthodes `toString` et `equals`

`public String toString()` renvoie

- Description de l'objet sous la forme d'une chaîne de caractères ☐
- Nom de la classe, suivie de "@" et de la valeur de la méthode

`hashCode()`

- `hashCode()` renvoie la valeur hexadécimale de l'adresse mémoire de l'objet
- Pour être utile, `toString()` doit être redéfinie
- Si `p1` est un objet, `System.out.println(p1)` affiche la chaîne de caractères `p1.toString()`

public boolean equals(Object obj) renvoie

- true si et seulement si l'objet courant this a « la même valeur » que l'objet obj □
- La méthode equals de Object renvoie true si this référence le même objet que obj □
- Elle peut être redéfinie dans les classes pour lesquelles on veut une relation d'égalité différente
- 2 objets égaux au sens de equals doivent renvoyer le même entier pour hashCode

Paramètre args de main

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

- Le tableau **args** est constitué d'arguments passés au programme via la ligne de commande :

Exemple : Soit **ArgsTest** une classe principale et son exécution

```
>java ArgsTest un deux trois
```

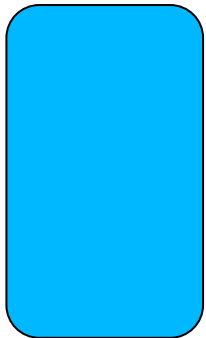
- `args[0] = "un", args[1] = "deux", args[2] = "trois".`

Les éléments du tableau `args` sont des chaînes de caractères

Paramètre args de main

```
class ArgsTest{  
    public static void main(String[] args) {  
        for (int i=0; i<args.length; i++) {  
            System.out.println(args[i]);  
        }  
    }  
}
```

Après exécution, on aura :



Paramètre args de main

```
class Somme {  
    public static void main(String[] args) {  
        int somme = 0;  
        for (int i=0; i < args.length ; i++) {  
            somme = somme + Integer.parseInt(args[i]); }  
        System.out.println(somme);  
    }  
}
```

Après exécution :

>java Somme 12 23 34, on aura :



Il est souvent nécessaire de convertir les args[i], qui sont toujours de type String

Classe et méthode abstraites

Fichier « Forme.java »

```
class Forme{
    Point2D centre;

    Forme(){ centre=new Point2D(); }
    Forme(Point2D c){ centre=c; }
    void deplacer(Vecteur2D vecteur) {
        centre.x += vecteur.x; centre.y += vecteur.y; }
    void deplacer(double x,double y) { centre.x += x; centre.y += y; }
    void deplacerH(double x) { centre.x += x; }
    void deplacerV(double y) { centre.y += y; }

    double surface() { return 0; }

    void afficher(String nature){
        System.out.print("Objet " + nature+" :\n\tcentre : ");
        centre.afficher();    }
}
```

Classe et méthode abstraites

Fichier « Carre.java »

```
class Carre extends Forme {  
    double longueur;  
  
    Carre(){ longueur=1; }  
    Carre(Point2D c,double l){ super(c); longueur=l; }  
  
    double surface() { return longueur * longueur; }  
  
    void afficher(){  
        super.afficher("Carre");  
        System.out.println("\n\tlongueur : " + longueur);  
    }  
}
```


Classe et méthode abstraites

Fichier « Cercle.java »

```
class Cercle extends Forme {  
    double rayon;  
  
    Cercle(){ rayon=1; }  
    Cercle(Point2D c,double l){ super(c); rayon=l; }  
  
    double surface() { return rayon * rayon; }  
  
    void afficher(){  
        super.afficher("Cercle");  
        System.out.println("\n\trayon : " + rayon);  
    }  
}
```

Il peut donc, dans certains cas, être utile de définir une méthode sans en donner le code. Utilisation du mot clé abstract

```
abstract double area();
```

au lieu de

```
double area(){ return 0; }
```

Classe et méthode abstraites

Fichier « Forme.java »

```
abstract class Forme{
    Point2D centre;

    Forme(){ centre=new Point2D(); }
    Forme(Point2D c){ centre=c; }
    void deplacer(Vecteur2D vecteur) {
        centre.x += vecteur.x; centre.y += vecteur.y; }
    void deplacer(double x,double y) { centre.x += x; centre.y += y; }
    void deplacerH(double x) { centre.x += x; }
    void deplacerV(double y) { centre.y += y; }

    abstract double surface() ;

    void afficher(String nature){
        System.out.print("Objet " + nature+" :\n\tcentre : ");
        centre.afficher();    }
}
```

Faire usage de l'abstraction

On ne peut plus créer d'objet de la classe *Forme*

Ecrire une classe **Volaille** . Une volaille est caractérisée par un « **numéro** » et un « **poids** ».

Ajouter à cette classe un constructeur et une méthode **changePoids** qui permet de modifier son poids.

Ecrire **deux autres méthodes**, l'une retourne son prix, et l'autre si la volaille est assez grosse pour être abattue.

Le prix et l'estimation de la grosseur dépend du type de la volaille (Poulet, canard, dinde, etc.).

Ecrire une classe **Poulet** qui spécifie la classe Volaille .

Tous les poulets ont le même prix de vente au kilo,
et le même poids d'abattage.

Cependant, pour des raisons de marché,
le prix de vente et le poids d'abattage peuvent changer.

Il existe une autre technique pour introduire de l'abstraction

```
interface I1 {  
    void m();  
}  
  
abstract class C1 {  
    abstract void g();  
}  
  
class C2 extends C1 implements I1 {  
    void m(){ // Le code de m }  
    void g(){ // Le code de g }  
}
```

Héritage entre interfaces

```
interface I2 extends I1 {  
    void n();  
}  
  
abstract C3 implements I2 {  
    void m(){  
        // Le code de m();  
    }  
}
```


Héritage simple

```
class MyClass
    extends MotherClass
    implements Interface1, Interface2 {
    ...
}
```