Chapitre 4
Le polymorphisme

Module15: La programmation orientée objet JAVA

Plan

Introduction

- Nous avons certainement tous appris à l'école qu'il était impossible d'additionner des pommes et des oranges ! ! !
- Mais, avec le concept de la POO on peut écrire l 'expression suivante :

$$3\ \textit{pommes} + 5\ \textit{oranges} = 8\ \textit{fruits}$$

- **Lidée de base** : Pour admettre cette ecriture il suffit d'imaginer que nous disons pommes et oranges mais que nous manipulons en fait des fruits.

Polymorphisme

- C'est un concept puissant de la POO qui complète l'héritage
- Est un mécanisme par lequel un nom peut désigner des objets de nombreuses classes différentes, tant qu'elles sont reliées par une super-classe commune
- Il explique comment une méthode peut se comporter différemment suivant l'objet sur lequel elle s'applique
- Plus précisément, quand une même méthode est définie à la fois dans la classe mère et dans la classe fille, son exécution est réalisée en fonction de l'objet associé à l'appel et non plus suivant le nombre de paramètres, comme c'est le cas lors de la surcharge de méthodes

Polymorphisme

- Étudions l'exemple :

```
public class Forme {
 protected int x, y, couleur ;
 public Forme(int nx, int ny) {
    x = nx ;
    y = ny ;
    couleur = 0;
 }
 public void afficher() {
    System.out.println("Position en " + x + ", "
    + y);
    System.out.println("Couleur : " + couleur);
 }
 public void échangerAvec(Forme autre) {
    int tmp;
    tmp = x;
    x = autre.x;
    autre.x = tmp;
    tmp = y;
     y = autre.y;
    autre.y = tmp;
 }
 public void déplacer(int nx, int ny)   {
    x = nx;
    y = ny;
 }
} // Fin de la classe Forme
```

```
public class Cercle extends Forme {
    public final static int TailleEcran = 600 ;
    private int  r ;
    public Cercle(int xx, int yy){
            super(xx, yy);
           couleur = 10;
           r = rayonVérifié();
    }
    public void afficher()  {
           super.afficher();
           System.out.println("Rayon : " + r);
    }
    ...
}// Fin de la classe Cercle
```

Polymorphisme

⇒ **Explications**

- La méthode afficher() est décrite dans la classe Forme et dans la classe Cercle
- Cette double définition ne correspond pas à une véritable surcharge de fonction
- En effet, les deux méthodes afficher() sont définies sans aucun paramètre
- Le choix de la méthode ne peut donc s'effectuer sur la différence de paramètres
- Il est effectuée par rapport à l'objet sur lequel la méthode est appliquée

Polymorphisme

⇒ **Observons l'exécution du programme suivant**

```
public class FormerDesCercles {
 public static void main(String [] arg) {
   Cercle A  = new Cercle1(5, 5);
   A.afficher(); // appliquée à A
   Forme F = new Forme (10, 10);
   F.afficher(); //appliquée à F
 }
}
```

Polymorphisme

⇒ Exemple : la situation

- Considérons cette situation dans laquelle les classes Point et Poincol sont censées disposer chacune d'une méthode affiche()

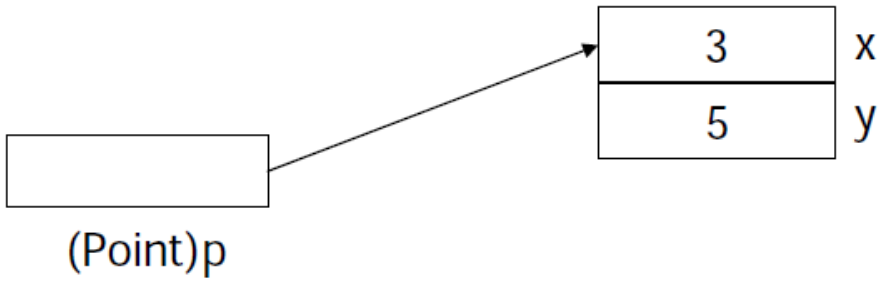
```
class Point {
   public Point(int x, int y){...}
   public void affiche() {...}
}

Class Pointcol extends Point {
   public Pointcol(int x, int y, byte couleur){...}
   public void affiche() {...}
}
```

Polymorphisme

⇒ **Exemple : référence**

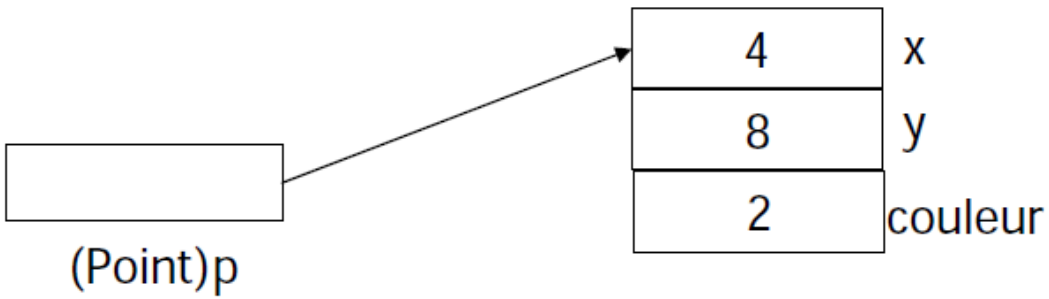
- Avec ces instructions
  Point p ;
  p = new Point(3,5) ;
- On aboutit tout naturellement à cette situation



Polymorphisme

⇒ **Exemple : changement de référence**

- Mais il se trouve que Java autorise ce genre d'affectation (p est toujours de type Point !)
  p = new Pointcol(3,5, (byte)2)
- La situation correspondante est :



Polymorphisme

⇒ **Exemple : constat**

- Java permet d'affecter à une variable objet non seulement la référence à un objet du type correspondant, mais aussi une référence à un objet d'un type dérivé
- On est en présence d'une conversion implicite d'une référence à un type de classe T en une référence à un type ascendant de T

Polymorphisme

⇒ **Exemple : et pour affiche ?**

- Considérons maintenant ces instructions

```
Point p = new Point(3,5);
p.affiche(); //appelle la méthode affiche de la classe Point
p = new Pointcol(4,8,2);
p.affiche(); //appelle la méthode affiche de la classe Pointcol
```

- Constat
  - Dans la dernière instruction, p est de type Point, alors que l'objet référencé par p est de type Pointcol
  - p.affiche appelle alors la méthode affiche de la classe Pointcol

⇒Java ne se fonde pas sur le type de la variable p mais bel et bien sur le type effectif de l'objet référencé au moment de l'appel

Ligature dynamique (Late Binding)

- Dans la plupart des langages, lorsque le compilateur rencontre un appel de méthode, il doit être à même de savoir exactement de quelle méthode il s'agit.
- Le lien entre l'appel et la méthode est alors établi au moment de la compilation. Cette technique est appelée early binding, que l'on pourrait traduire par liaison précoce.
- Java utilise cette technique pour les appels de méthodes déclarées final.
- Elle a l'avantage de permettre certaines optimisations.
- En revanche, pour les méthodes qui ne sont pas final, Java utilise la technique Late Binding du late binding (liaison tardive).

Ligature dynamique (Late Binding)

- Dans ce cas, le compilateur n'établit le lien entre l'appel et la méthode qu'au moment de l'exécution du programme.
- Ce lien est établi avec la version la plus spécifique de la méthode.
- Dans notre cas, nous la méthode affiche() possède 3 versions définies dans les classes Fruit, Pomme et Orange.
- Grâce au late binding, java est capable de déterminer, au moment de l'exécution, quelle version de méthode qui sera appelée ce que nous pouvons vérifier par le programme suivant :

Polymorphisme

- Exemple complet intégrant les situations exposées ci-dessus

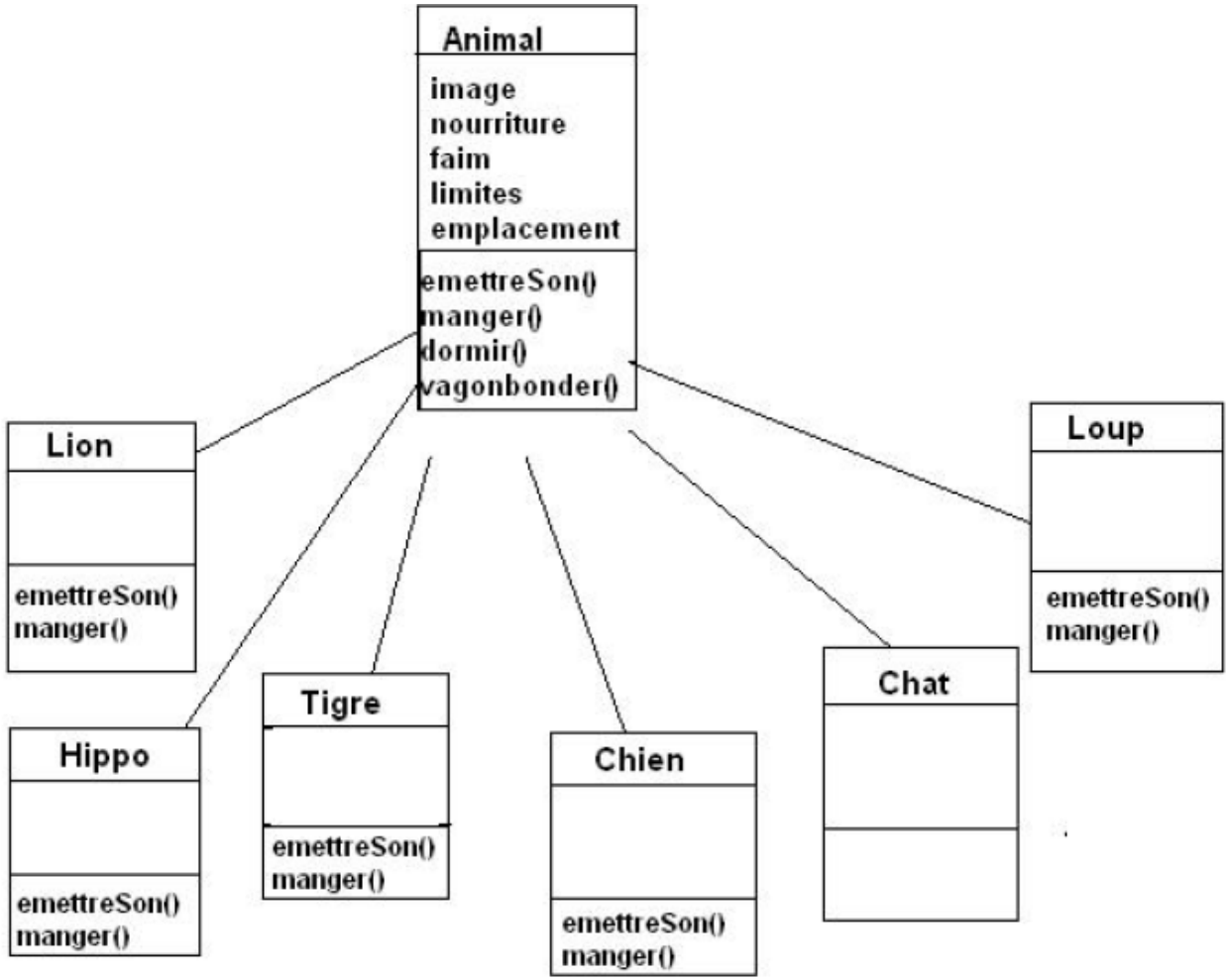
```
class Point2{
    private int x, y;
    public Point2 (int x, int y) {
      this.x = x ; this.y = y ; }
    public void deplace (int dx, int dy) {
      x += dx; y+=dy;}
    public void affiche(){
      System.out.println("Je suis en " + x +
    " " + y);}
}

class Pointcol extends Point2{
    private byte couleur;
    public Pointcol (int x, int y, byte
    couleur){
      super (x, y) ;
      this.couleur = couleur ;
    }
  public void affiche(){
        System.out.println("et ma couleur est :
        " +          couleur);
  }
}
```

```
public class Poly{
    public static void main (String
    args[]){
        Point2 p = new Point2 (3, 5) ;
        p.affiche(); //appelle affiche de
        Point
        Pointcol pc = new Pointcol(4,
        8, (byte)2) ;
        p = pc; //p de type Point,
        référence à un objet de type
        Pointcol
        p.affiche(); //appelle affiche de
        Pointcol
        p = new Point2 (5, 7); // p
        référence à nouveau un objet
        de type Point
        p.affiche(); //appelle affiche de
        Point
    }
}
```

Polymorphisme

⇒ Exemple 1



Polymorphisme

- Création d'un tableau contenant 5 animaux.

```
Animal[] animaux=new Animal[5]
Animaux[0]= new Chien();
Animaux[1]= new Chat();
Animaux[2]= new Loup();
Animaux[3]= new Hippo();
Animaux[4]= new Lion();
```

Déclare un tableau de type animal

On peut placer un objet de n'importe quelle sous-classe d'Animal dans le tableau

```
for (int i=0;i<animaux.length;i++) {
    Animaux[i].manger(); //Java saura faire la différence
    Animaux[i].vagonbonder(); //Java saura faire la différence
}
```

Polymorphisme

⇒ **Exemple 2**

```
public class Figure {
public void dessineToi() { }
}
public class Rectangle extends Figure {
public void dessineToi() {
. . .
}
public class Cercle extends Figure {
public void dessineToi() {
. . .
}
```

Polymorphisme

⇒ **Le code du fichier .java**

```
public class Dessin { // dessin composé de plusieurs figures
private Figure[] figures ;
. . .
public void afficheToi() {
for (int i=0 ; i < nbFigures ; i++)
figures[i].dessineToi() ;
}
```

⇒ **Le code du fichier TestApplication.java**

```
public class TestApplication {
public static void main(String[] args) {
Dessin dessin = new Dessin(30) ;
. . . // création des points centre, p1, p2
dessin.ajoute(new Cercle(centre, rayon)) ;
dessin.ajoute(new Rectangle(p1, p2)) ;
dessin.afficheToi() ;

. . .
```

Polymorphisme

Problème

Sans utilisation du polymorphisme, la méthode **dessineToi()** correspond à quel objet ?

```
for (int i=0 ; i < figures.length ; i++)
{ figures[i].dessineToi()
}
```

Solution

On uiltise l'opérateur instanceof, le code devient

```
for (int i=0 ; i ¡ figures.length ; i++) {
if (figures[i] instanceof Rectangle) {
. . . // dessin d'un rectangle
}
else if (figures[i] instanceof Cercle) {
. . . // dessin d'un cercle
}
}
```

⇒le polymorphisme évite les codes qui comportent de nombreux embranchements et

L'opérateur instanceof

- Si **x** est une instance d'une sous-classe **B** de **A**,
  x instanceof A
  renvoie **true**
- Pour tester si un objet **o** est de la même classe que l'objet courant, il ne faut donc pas utiliser **instanceof** mais le code suivant :

```
if (o != null && o.getClass() == this.getClass())
```

public Class getClass()

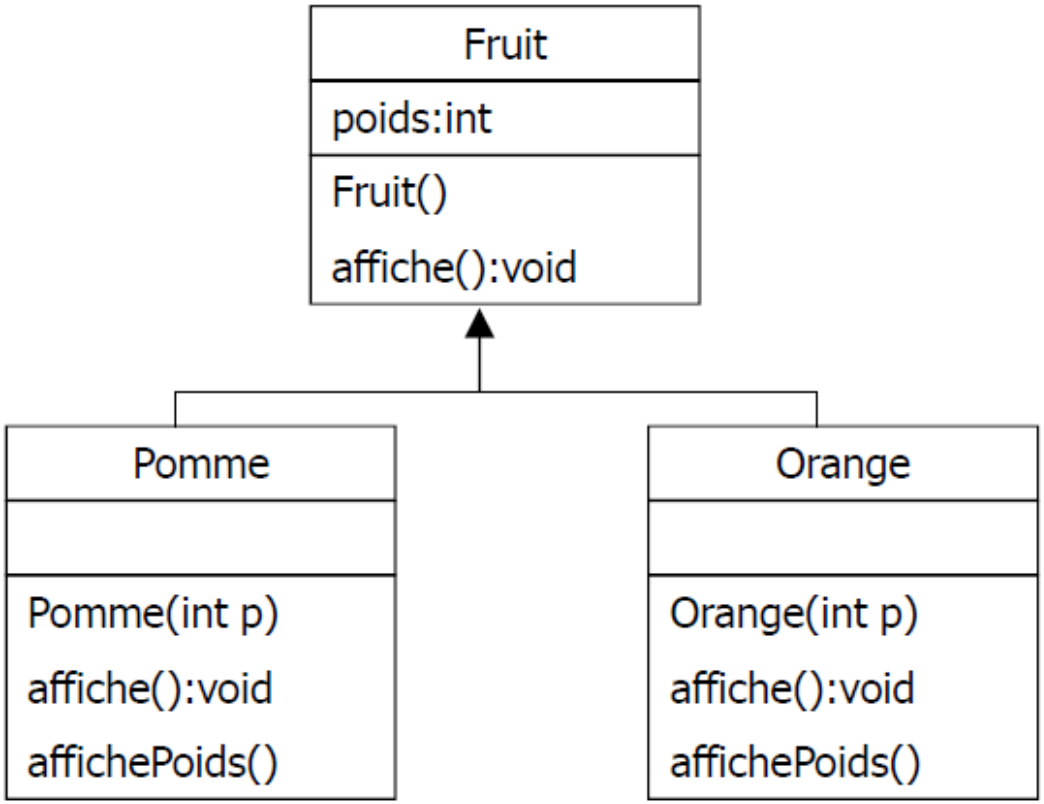
renvoie la classe de l'objet (le type retour est un peu plus complexe comme on le verra dans le cours sur la généricité)

Définition

- Le « cast » est le fait de forcer le compilateur à considérer un objet comme étant d'un type qui n'est pas le type déclaré ou réel de l'objet
- En Java, les seuls casts autorisés entre classes sont les casts entre classe mère et classes filles
- On parle de **Sur-casting** (upcast) et de **Sous-casting** (downcast) suivant le fait que le type est forcé de la classe fille vers la classe mère ou inversement

Sur-casting des objets

- Considérons l'exemple suivant :



Sur-casting des objets

⇒**fichier Fruit.java**

```
public class Fruit{
int poids;
public Fruit(){
System.out.println("Création d'un fruit");}
public void affiche(){
System.out.println("c'est un fruit"); }
}
```

⇒**fichier Pomme.java**

```
public class Pomme extends Fruit{
public Pomme(int p){
poids=p;
System.out.println("création d'une pomme de "+ poids+" grammes ");
}
public void affiche(){
System.out.println("C'est une pomme"); }
public void affichePoids(){
System.out.println("le poids de la pomme est :"+poids+" grammes"); }
}
```

Sur-casting des objets

⇒**fichier Orange.java**

```
public class Orange extends Fruit{
public Pomme(int p){
poids=p;
System.out.println("création d'une pomme de "+ poids+" grammes ");
}
public void affiche(){
System.out.println("C'est une orange");
}
public void affichePoids(){
System.out.println("le poids de l'orange est :"+poids+" grammes");
}
}
```

Sur-casting des objets

- Considérons l'exemple suivant :

⇒**fichier Polymorphisme.java**

```
public class Polymorphisme{
public static void main(String[] args){
// Sur-casting implicite
Fruit f1=new Orange(40);
// Sur-casting explicite
Fruit f2=(Fruit)new Pomme(60);
// Sur-casting implicite
f2=new Orange(40);
}
}
```

Sur-casting des objets

- Un objet de type Pomme peut être affecté à un handle de type fruit sans aucun problème :
  - Fruit f1 ;
  - f1=new Pomme(60) ;
- Dans ce cas l'objet Pomme est converti automatiquement en Fruit.
- On dit que l'objet Pomme est sur-casté en Fruit.
- Dans java, le sur-casting peut se faire implicitement.
- Toutefois, on peut faire le sur-casting explicitement sans qu'il soit nécessaire.
- La casting explicit se fait en précisant la classe vers laquelle on convertit l'objet entre parenthèse. Exemple :
  - f2=(Fruit)new Orange(40) ;

Sous-casting des objets

- Considérons l'exemple suivant :

```
public class Polymorphisme3{
public static void main(String[] args){
  Fruit f1;
  Fruit f2;
  f1=new Pomme(60);
  f2=new Orange(40);
f1.affichePoids();
  ((Pomme)f1).affichePoids();
}
}
```

- Erreur de compilation:

```
Polymorphisme3.java:5: cannot resolve symbol
    symbol  : method affichePoids ()
    location: class Fruit
    f1.affichePoids();
      ^
1 error
```

- Solution : Sous-casting explicit

- Ce message indique que l'objet f1 qui est de type Fruit ne possède pas la méthode affichePoids().
- Cela est tout à fait vrai car cette méthode est définie dans les classes Pomme et Oranges et non dans la classe Fruit.

Sous-casting des objets

- En fait, même si le handle f1 pointe un objet Pomme, le compilateur ne tient pas en considération cette affectation, et pour lui f1 est un Fruit.
- Il faudra donc convertir explicitement l'objet f1 qui de type Fruit en Pomme.
- Cette conversion s'appelle Sous-casting qui indique la conversion d'un objet d'une classe de base vers un autre objet d'une classe dérivée.
- Dans ce cas de figure, le sous-casting doit se faire explicitement.

Sous-casting des objets

- L'erreur de compilation peut être évité en écrivant la syntaxe suivante :

```
((Pomme)f1).affichePoids();
```

- Cette instruction indique que l'objet f1 , de type Fruit, est converti en Pomme, ensuite la méthode affichePoids() de l'objet Pomme est appelé ce qui est correcte.

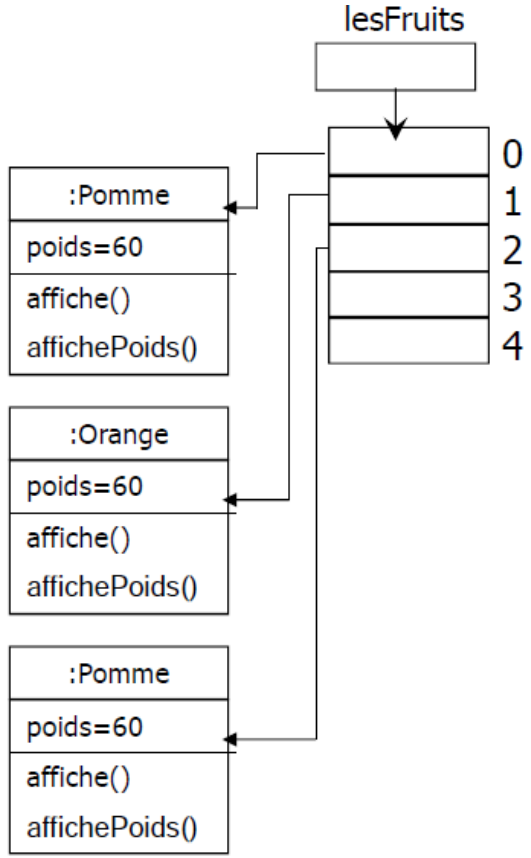
Polymorphisme

Pour résumer, un objet est une instance de :

- sa classe,
- toutes les classes parentes de sa classe,
- toutes les interfaces qu'il implémente, toutes les interfaces parentes des interfaces qu'il implémente,
- toutes les interfaces qu'implémentent les classes parentes de sa classe,
- toutes les interfaces parentes des précédentes

Tableaux d'objets

- Déclaration :
  - Exemple : Tableau d'objets Fruit
  - Fruit[] lesFruits ;
- Création du tableau
  - lesFruits = new Fruit[5] ;
- Création des objets :
  - lesFruits[0]=new Pomme(60) ;
  - lesFruits[1]=new Orange(100) ;
  - lesFruits[2]=new Pomme(55) ;
- Manipulation des objets :
  for(int i=0 ;i<lesFruits.length ;i++){
  lesFruits[i].affiche() ;
  if(lesFruits[i] instanceOf Pomme)
  ((Pomme)lesFruits[i]).affichePoids() ;
  else
  ((Orange)lesFruits[i]).affichePoids() ; }
- Un tableau d'objets est un tableau de handles



Classe final (et autres au final)

- **Classe final :**
  - ne peut avoir de classes filles (String est final)
- **Méthode final :**
  - ne peut être redéfinie
- **Variable (locale ou d'état) final :**
  - la valeur ne pourra être modifiée après son initialisation
- **Paramètre final (d'une méthode ou d'un catch) :**
  - la valeur (éventuellement une référence) ne pourra être modifiée dans le code de la méthode