Université Moulay Ismaïl
École Normale Supérieure de Meknès
Département des sciences

Programmation oriontée objet en JAVA

CH4 (polymorphisme, classes abstraites et interfaces)

Pr. Mehdi Moukhafi

Héritage et Polymorphisme

Classes et Sous-classes

Ou

Extension de Classes

POO en JAVA

Héritage et Polymorphisme

POO en Java

Héritage et **Polymorphisme**

Plusieurs formes

Poly ==Plusieurs

Morph==Forme

Polymorphisme

Les Classes Shape

- Soit la classe Shape
  - supposons tous les shapes ont des coordonées x et y
  - redéfinissons la version **`toString`** de la classe Object

- Deux sous-classes de la classe Shape
  - Rectangle définit une nouvelle méthode changeWidth
  - Circle

Une Classe *Shape*

```
public class Shape
{  private double dMyX, dMyY;

   public Shape() {  this(0,0);}

   public Shape (double x, double y)
   {  dMyX = x;
      dMyY = y;
   }

   public String toString()
   { return "x: " + dMyX + " y: " + dMyY;
   }

   public double getArea() {
       return 0;
   }
}
```

Une Classe *Rectangle*

```
public class Rect extends Shape
{  private double dMyWidth, dMyHeight;

   public Rect(double width, double height)
   {
   dMyWidth = width;
      dMyHeight = height;
   }
   public String toString()
   {  return
       " width " + dMyWidth
       + " height " + dMyHeight;
   }
   public double getArea()  {
       return dMyWidth * dMyHeight;
   }
   public void changeWidth(double width){
        dMyWidth = width;
}  }
```

Une Classe *Circle*

```
class Circle extends Shape {
   private double radius;
   private static final double PI = 3.14159;

   public Circle(double rad){
   radius = rad;
   }

   public double getArea() {
      return PI * radius * radius;
   }

   public String toString() {
      return " radius " + radius;
   }
}
```

Polymorphisme

- Si la classe *Rect* est derivée de la classe *Shape*, une operation qui peut être exécutée sur un objet de la classe *Shape* peut aussi être exécutée sur un objet de la classe *Rect*

- Quand une requête d'utiliser une méthode est effectuée à travers une reférence de la superclasse, Java choisit la méthode redéfinie correcte "polymorphiquement" dans la sous-classe appropriée associée à l'objet

- Polymorphisme signifie “différentes formes”

Références aux objets

```
Rect r = new Rect(10, 20);
Shape s = r;
System.out.println("Area is " + s.getArea());
```

- Etant donné que la classe Rect est une sous-classe de la classe Shape, et elle redéfinit la méthode *getArea()*.
- Cela va t-il fonctionner?

Références aux objets

```
Rect r = new Rect(10, 20);
Shape s = r;
System.out.println("Area is " + s.getArea());
```

- Le code fonctionne si **Rect extends Shape**
- Une référence à un objet peut faire référence à un objet de son type de base ou un descendant dans la chaîne d'héritage
  - La relation *est-un* est vérifiée. Un Rect *est-un* shape alors **s** peut faire référence à Rect
- C'est une forme de *polymorphisme* et est utilisée extensivement dans le JCF "Java *Collection* Framework"
  - Vector, ArrayList, LinkedList sont des listes d'objets

Polymorphisme

```
Circle c = new Circle(5);
Rect r = new Rect(5, 3);
Shape s = null;
if( Math.random(100) % 2 == 0 )
     s = c;
else
     s = r;
System.out.println( "Shape is "
     + s.toString() );
```

- Supposons maintenant que Circle et Rect sont toutes les deux des sous-classes de Shape, et toutes les deux ont redéfinis *toString()*, quelle version sera appelée?

Polymorphisme

```
Circle c = new Circle(5);
Rect r = new Rect(5, 3);
Shape s = null;
if( Math.random(100) % 2 == 0 )
      s = c;
else
      s  =  r;
System.out.println( "Shape is "
      + s.toString() );
```

- Circle et Rect ont **redéfinis** toString quelle version sera appelée?
  - Le code fonctionne parce que **s** est **polymorphique**
  - L'appel de méthode determiné à l'exécution par le ***dynamic binding (liaison dynamique)***

Compatibilité de Type

```
Rect r = new Rect(5, 10);
Shape s = r;
s.changeWidth(20);
```

- Etant donné la classe Rect qui est une sous-classe de la classe Shape, et qui a une nouvelle méthode *changeWidth(double width)*, que sa superclasse n'a pas.
- Cela va t-il fonctionner?

Compatibilité de Type

```
Rect r = new Rect(5, 10);
Shape s = r;
s.changeWidth(20); // erreur de syntaxe
```

- Le polymorphisme permet à **s** de faire reférence à un objet Rect, mais il y a des limitations
- Le code en haut ne fonctionne pas
- Comment le modifier un peu pour le faire fonctionner sans changer les définitions des classes ?

Compatibilité de Type

```
Rect r = new Rect(5, 10);
Shape s = r;
s.changeWidth(20); // erreur de syntaxe
```

- Le polymorphisme permet à **s** de faire reférence à un objet Rect, mais il y a des limitations
- Le code en haut ne fonctionne pas
- Statiquement **s** est declarée pour être un shape
  - pas de méthode changeWidth dans la classe Shape
  - doit faire un ***cast*** de s à un *rectangle*;

```
Rect r = new Rect(5, 10);
Shape s = r;
((Rect)s).changeWidth(20); //Okay
```

Problémes avec le Casting

```
Rect r = new Rect(5, 10);
Circle c = new Circle(5);
Shape s = c;
((Rect)s).changeWidth(4);
```

- Est ce que cela fonctionne?

Problémes avec le Casting

- Le code suivant compile mais une exception est lancée à l'exécution

```
Rect r = new Rect(5, 10);
Circle c = new Circle(5);
Shape s = c;
((Rect)s).changeWidth(4);
```

- Le Casting doit être fait soignesement et correctement
- Si l'on est pas sûr de quel type l'objet sera alors utilise l'operateur **`instanceof`**

L'opérateur instanceof

```
Rect r = new Rect(5, 10);
Circle c = new Circle(5);
Shape s = c;
if(s      instanceof      Rect)
    ((Rect)s).changeWidth(4);
```

- syntaxe: **expression instanceof NomClasse**

Casting

- Il est toujours possible de convertir une sous-classe à une superclasse. pour cette raison, le casting explicite peut être omit. Par exemple,
  - `Circle c1 = new Circle(5);`
  - `Shape s = c1;`

est equivalent à

  - `Shape s = (Shape)c1;`

- Le casting explicite doit être utilisé quand on fait le casting d'un objet d'une superclasse à une sous-classe. Ce type de casting peut ne pas aboutir.
  - `Circle c2 = (Circle) s;`

```
class Point {
    ...
    public String toString(){
        return "["+getX()+","+getY()+"]";
    }

    public boolean equals(Object o){
        if(o == this){
            return true;
        }
        else if(o == null){
            return false;
        }
        else if(o instanceof Point){
            Point p = (Point) o;
            return getX() == p.getX() && getY() == p.getY();
        }
        else{
            return false;
        }
    }
    ...
}
```

Héritage et Classes abstraites

Classes abstraites

- Il peut être nécessaire au programmeur de créer une classe déclarant une méthode sans la définir (c'est-à-dire sans en donner le code). La définition du code est dans ce cas laissée aux sous- classes.
- Une telle classe est appelée classe abstraite.
- Elle doit être marquée avec le mot réservé abstract.
- Toutes les méthodes de cette classe qui ne sont pas définies doivent elles aussi être marquées par le mot réservé abstract.
- Une classe abstraite ne peut pas être instanciée.

Classes abstraites

- Par contre, il est possible de déclarer et d'utiliser des variables du type de la classe abstraite.
- Si une sous-classe d'une classe abstraite ne définit pas toutes les méthodes abstraites de ses superclasses, elle est abstraite elle aussi.

```
public abstract class Polygone
  {
     private int nombreCotes = 3;
     public abstract void dessine (); // méthode non définie
     public int getNombreCotes()
       {
          return(nombreCotes);
       }
  }
```

Classes abstraites

Remarque:

Une classe abstraite (présentée par le mot clé abstract) peut ne pas contenir de méthodes abstraites. Cependant, une classe contenant une méthode abstraite doit obligatoirement être déclarée abstract.

Interfaces

Interfaces : comment classifier ?

- Java ne permet pas l'héritage multiple

- Or, il existe parfois différentes classifications possibles selon plusieurs critères

Exemple de classification

Selon la forme

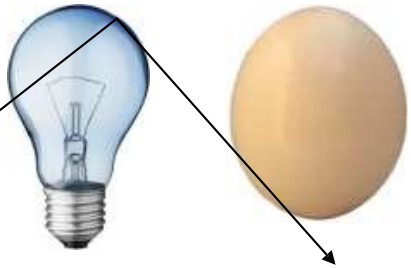
Solides convexes

Polyèdres

Parallélépipède

Cube

Solides de révolution



Cylindres

Sphères

Autres critères

D'autres critères qui pourraient servir à réaliser une classification décrivent des comportements ou des capacités

- « électrique »
- « comestible »
- « lumineux »

Or ces « mécanismes » peuvent être commun à différentes classes non reliées entre elles par une relation d 'héritage

Notion d'« Interfaces »

Pour définir qu'une certaine catégorie de classes doit implémenter un ensemble de méthodes, on peut regrouper les déclarations de ces méthodes dans une interface.

Le but est de décrire le fait que de telles classes pourront ainsi être manipulées de manière identique.

Exemple :

- Tous les appareils électriques peuvent être allumés ou éteint
- Tous les objets comestibles peuvent être mangés
- Tous les objets lumineux éclairent

Définition d'Interface

Une interface est donc la description d'un ensemble des procédures (méthodes) que les classes Java peuvent mettre en oeuvre.

Les classes désirant appartenir à la catégorie ainsi définie

- déclareront qu'elles implémentent cette interface,
- fourniront le code spécifique des méthodes déclarées dans cette interface.

Cela peut être vu comme un contrat entre la classe et l'interface

- la classe s'engage à implémenter les méthodes définies dans l'interface

Codage d'une interface en Java

Mot réservé : interface

Dans un fichier *Nom_interface*.java, on définit la liste de toutes les méthodes de l'interface

```
interface NomInterface {
type_retour methode1(paramètres);
type_retour methode2(paramètres);
 ... }
```

Les méthodes d'une interface sont abstraites : elles seront écrites spécifiquement dans chaque classe implémentant l'interface

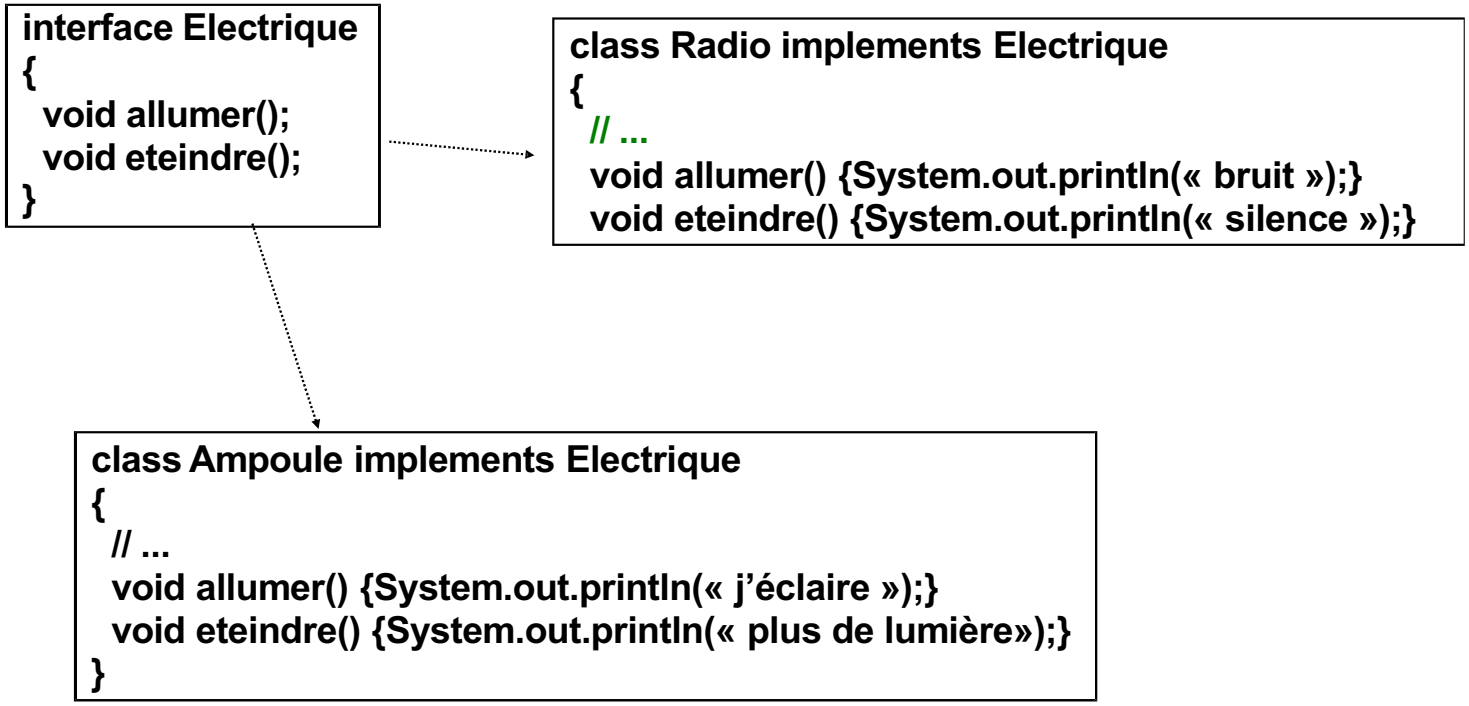
Le modificateur abstract est facultatif.

Implémentation d'une interface dans une classe

- Mot réservé : **implements**
- La classe doit expliciter le code de chaque méthode définie dans l'interface

```
class MaClasse implements nomInterface {
                                    …
   type_retour methode1(paramètres)
{code spécifique à la méthode1 pour cette classe};
…
```

Exemple d'Interface (1)



Exemple d'Interface (2)

```
// …

Ampoule monAmpoule = new Ampoule();
Radio maRadio = new Radio();
Electrique c;
Boolean sombre;

// …

if(sombre  == true)
   c = monAmpoule;
else
   c = maRadio;

c.allumer();
…
c.eteindre();

// …
```

Utilisation des interfaces

- Une variable peut être définie selon le type d'une interface
- Une classe peut implémenter plusieurs interfaces différentes
- L'opérateur instanceof peut être utilisé sur les interfaces

Exemple :

```
interface Electrique
…
interface Lumineux
…
class Ampoule implements Electrique, Lumineux
…
Electrique e;
Object o = new Ampoule();
if (o instanceof Electrique) {e=(Electrique)o;e.allumer();}
```

Conclusion sur les interfaces

Un moyen d'écrire du code générique

Une solution au problème de l'héritage multiple

Un outil de concevoir d'applications réutilisables