P.O.O. (Programmation Orientée Objet)

CHOUITI Sidi Mohammed

Cours pour L2 en Informatique

Département d'Informatique

Université de Tlemcen

2019-2020

Plan du cours

1. Introduction à la Programmation Orientée Objet
2. Classes et Objets
3. Héritage, polymorphisme
4. Abstraction, déclaration finale
5. Interface, implémentation et Paquetage
6. Classes Courantes en Java
7. Gestion des Exceptions
8. Interfaces graphiques

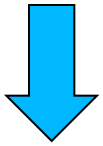
Fichier "Carre.java"

```
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

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();        }
                  }
```

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 fille (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 « 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(){super(); rayon=1; }
      //overloading
      Cercle(Point2D c,double l){ super(c);   rayon=l; }

      //overriding
      void afficher(){       //Redéfinition
            super.afficher("Cercle");
            System.out.println("\n\trayon : " + rayon);
      }
}
```

La **surcharge (overloading)** survient lorsque deux méthodes ou plus dans une classe ont le même nom de méthode mais des paramètres différents.

La **redéfinition (overriding)** signifie avoir deux méthodes avec le même nom et les mêmes paramètres, l'une des méthodes est dans la classe parente et l'autre dans la classe fille.

La redéfinition permet à une classe fille de fournir une implémentation spécifique d'une méthode déjà fournie à sa classe parente.

Intérêts de l'héritage

- 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"

```
class Classe1 {
    Classe1(){
       System.out.println(" Classe1");}
    Classe1(int val){
       this();
       System.out.println(val );    }
}
```

fichier "Classe2.java"

```
class Classe2 extends Classe1 {
       Classe2(){
             super(5);
             System.out.println(" Classe2");}
       Classe2(int val){
             System.out.println(val );    }
}
```

exemple de création d'objet	résultats
new Classe1();	
new Classe1(3);	
new Classe2();	
new Classe2(2);	

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"

class A {
…
}
	class B extends A {
	…
	}

```
B b = new B();

A a = b;      // Utilisation du polymorphisme
(A) b;        // Utilisation du Casting
```

Polymorphisme

En redéfinissant une méthode dans une sous-classe, on peut **spécialiser (spécifier)** le comportement d'une méthode.

fichier "A.java"

```
class A {
      void m() {
            System.out.println (" Mère " ); }
}
```

fichier "B.java"

```
Class B extends A {
   void m() {
     System.out.println ("Fille" );}
}
```

```
class Polym {
      public tatic void main( String [] args) {
            B b = new B();
            A a = b;
            a.m();  //polymorphisme: la méthode la plus spécifique est appelée}
}
```

Règles de transtypage ou de Casting (UpCasting + DownCasting)

- Dans une instruction d'affectation (refVar1 = refVar2;),
  le compilateur vérifie que :
  le type de la variable refVar1 est le **même** que celui de refVar2
  ou **un type parent** du type de la variable refVar2.

"**UpCasting**" implicite/explicite : permet de convertir le type d'une référence vers un type parent.

UpCasting

Exemple

```
Object obj1 = new String();          // UpCasting implicite

Object obj2 = (Object) new String();  // UpCasting explicite
```

UpCasting

La conversion (implicite ou explicite) vers un type parent est toujours acceptée par le compilateur et elle ne posera aucun problème à l'exécution du programme.

Une référence d'un type parent peut, sans risque, lire/modifier les attributs ou invoquer les méthodes dont elle a accès, indépendamment si l'instance référée et du même type ou un sous-type du type de la référence.

DownCasting

Permet de convertir le type d'une référence vers un sous type.

DownCasting implicite

Exemple

```
  String str1 = new Object(); // DownCasting implicite
(erreur de compilation : Type mismatch: cannot
convert from Object to String)
```

La conversion implicite vers un sous type est toujours refusée par le compilateur.

DownCasting explicite

Exemple

```
String str2 = (String) new Object(); //DownCasting explicite

(erreur à l'exécution : java.lang.ClassCastException:
java.lang.Object cannot be cast to java.lang.String)
```

La conversion explicite vers un sous type est toujours acceptée par le compilateur. Cependant à l'exécution du programme, la JVM va vérifier si le type de l'instance (Object) est le même ou un sous-type du type qu'on a spécifiée pour la conversion (String) : si ce n'est pas le cas, la JVM déclenchera une exception.

DownCasting explicite

Exemple

```
String str2 = (String) "chaine"; //DownCasting explicite

// qui fonctionne
```

Classe Object

- 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**

Classe Object

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()

Classe Object

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