جامعة هواري بومدين
للعلوم والتكنولوجيا
U S T H B

Chapitre 5 (suite)

Héritage et Polymorphisme

S. BOUKHEDOUMA

USTHB – FEI – département d'Informatique
Laboratoire des Systèmes Informatiques -LSI
sboukhedouma@usthb.dz

Exercice

1- On demande d'implémenter une classe **Forme** qui décrit les formes géométriques régulières. On suppose qu'une forme possède un centre (Point) et une couleur.

2- Implémenter deux classes **Carré** et **Cercle** en donnant les spécificités de chacune d'elles. Les traitements à implémenter doivent permettre entre autres, de déplacer, redimensionner, calculer le périmètre et la surface de la forme.

Héritage - Exercice

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

//constructeur
 public Point (double x, double y)
   { this.x = x; thix.y = y;}
  // méthodes

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

public void déplacer (double a, double b)
 { x = x+a; y = y+b;}

} // fin de la classe Point
```

Héritage – Exercice

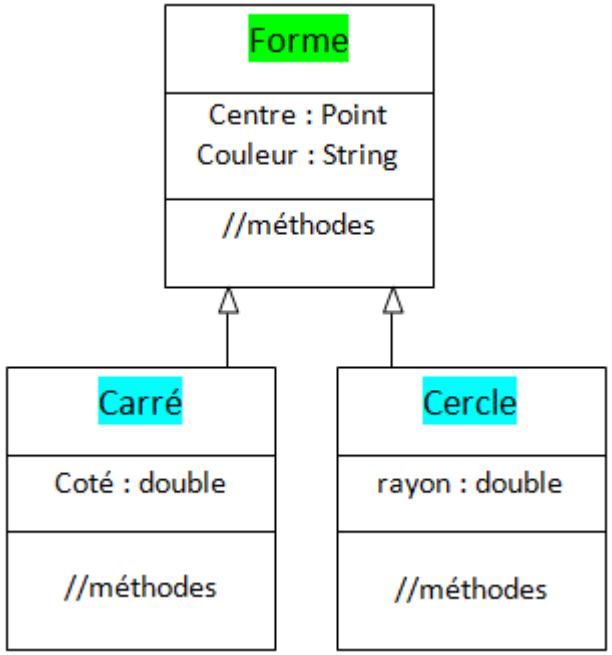
```
public class Forme
{ Point centre;
  String couleur;
                                        // constructeur
public Forme ( Point C, String coul)
      { centre = new Point (C.x, C.y); couleur = coul; }

                                        // méthode Modifier
public void afficher ()
      {System.out.println ("centre:"); centre.afficher();  // de la classe Point
       System.out.println ("\t couleur:"+ couleur); }

                                        // méthode déplacer
public void déplacer()
      { centre. déplacer (1.0, 2.0);}    // de la classe Point
}
```

Héritage – Exercice

```
public class Carré extends Forme
{ double coté;  //attribut supplémentaire
                        // constructeur
public Carré ( Point C, String coul, double coté)
      { super (C, coul);  this.coté = coté; }

                        // méthode afficher redéfinie
public void afficher ()
      {System.out.println ("Carré:");  super.afficher();
            System.out.println ("\t coté:"+ coté); }

                        // méthode redimensionner
public void redimensionner( double cot)
      { coté = cot;}

public void zoomer( float v)
      { coté = coté*v;}
```

Héritage – Exercice

```
public double surface ()
            {return (coté*coté);}

public double périmètre ()
            {return (coté*4);}

}  // fin de la classe Carré
```



Héritage – Exercice

```
public class Cercle extends Forme
{ double rayon;  //attribut supplémentaire
                        // constructeur
public Cercle ( Point C, String coul, double rayon)
      { super (C, coul);  this.rayon = rayon; }

                        // méthode afficher redéfinie
public void afficher ()
      {super.afficher(); System.out.println ("\t rayon:"+ rayon); }

                        // méthode redimensionner
public void redimensionner ( double r)
      { rayon = r;}

public void zoomer ( float v)
      { rayon = rayon*v;}
```

Héritage – Exercice

```
public double surface ()
        {return (Math.PI* Math.pow(rayon,2));}

public double périmètre ()    {return (2*Math.PI*rayon);}
}  // fin de la classe Cercle
```

```
class ProgForme
{ public static void main (String arg[ ])
   { Point P1 = new Point (3, 2) ; Point P2 = new Point (-5, 0);
   Carré CA = new Carré (P1, "bleu", 4.0);    Cercle   C   = new  Cercle (P2, " noir ", 2.5);
   CA.afficher();   CA.déplacer (2.0, 3.2);  CA. zoomer (0,75); CA.afficher();
   System.out.println (" le périmètre du carré est: "  +  CA.périmètre());
   C.afficher();   C.déplacer (4.5, 3.0);
 System.out.println (" le périmètre du cercle est: "  +  C.périmètre()); } }
```

Remarque: **Une méthode invoquée sera déterminée en fonction de la référence de l'objet sur lequel on fait l'appel → pas d'ambigüité.**

Héritage – Redéfinir une méthode

```
public class Produit
{ String ref;
  String libellé;
  float prix;
                                        // constructeur
public Produit ( String ref, String lib, float px)
        { this.ref = ref ; libellé = lib ; prix = px ; }

                                        // méthode Modifier
public void modifier (String newlib, float newpx)
        { libellé = newlib ; prix = newpx ; }

                                        // méthode afficher
public void afficher()
        { System.out.print (ref + "\t"+ libellé+ "\t"+ prix);}
}
```

Héritage – Redéfinir une méthode

```
public class Produit-Alimentaire extends Produit
{  Date DatFab, DatExp ; // attributs supplémentaires
   String Conditions;
                                    //constructeur
public Produit-Alimentaire (String ref, String lib, float px, Date DF, Date DE, String  cond)
{ …}
                                    // méthode modifier
public void Modifier (String newlib, float newpx, Date newDF, Date newDE)
{ …}
                    // méthode setCond  qui n'existe pas dans la superclasse

public void  setCond (String  cond)  { Conditions = cond;}

                                // méthode afficher  *** redéfinition***
public void afficher()
      { super.afficher ();
        DatFab.affiche();          DatExp.affiche();  // méthode affiche de la classe Date
        System.out.println (Conditions); }
}
```

Héritage – Redéfinir une méthode

- Une méthode est dite redéfinie si elle garde la **même signature** dans la superclasse et dans la sous-classe

La méthode **afficher est redéfinie**

La méthode **modifier n'est pas redéfinie (signatures différentes dans la superclasse et la sous-classe)**

- On parle d'une **méthode polymorphe**

se comporte de manière différente selon le type de l'objet (sous-classe ou superclasse)

Héritage – Redéfinir une méthode

```
class ProgProduit
{ public static void main (String arg[ ])
   { Date DF = new Date (3, 2, 2017) ; Date DF = new Date (3, 3, 2017) ;
   Produit-Alimentaire PA =
   new Produit-Alimentaire ("Ref028", "Lait", 90.0, DF, DE, "Secfrais- 10°C") ;
System.out.println("Il s'agit du produit suivant :") ;
   PA.afficher();        // la méthode afficher redéfinie
System.out.println (" on va modifier le produit ");
   PA.modifier ("Lait entier", 110.0, DE, DF );  // la méthode modifier
                                                    non redéfinie
 System.out.println ("Il s'agit maintenant du produit:");
   PA.afficher(); }
}
```

Héritage – La classe Object

```
public class Object
{ // méthodes de la classe Object
   public String toString ()
   public boolean equals (Object O)
   protected void finalize();
   public final Class getClass ();
  …
}
```

La méthode **toString de la classe Object** appliquée sur n'importe quel objet, renvoie le nom de la classe de l'objet concaténé à la référence de l'objet (adresse en hexadécimal)

Héritage – la méthode toString

```
class ProgProduit
{ public static void main (String arg[ ])
  {Produit P = new Produit ("Ref012", "savon liquide", 200.00)
   System.out.println (P.toString());
    // affiche par exemple : Produit@006352.

   Date DF = new Date (3, 2, 2017) ; Date DF = new Date (3, 3, 2017) ;
 Produit-Alimentaire PA = new Produit-Alimentaire ("Ref028", "Lait", 90.00, DF,
                                                    DE, "Sec-frais-10°C")
System.out.println (PA.toString());
// affiche par exemple : Produit-Alimentaire@003E52.
 }
```

Héritage – redéfinition de la méthode toString

➢ On pense toujours à **redéfinir la méthode toString** dans chaque classe pour **retourner une description textuelle de l'objet**

```
public class Produit
{String ref;   String libellé;  float prix;
public Produit ( String ref, String lib, float px) // constructeur
      { this.ref = ref ; libellé = lib ; prix = px ; }
                              // méthode Modifier
…
                              // méthode afficher
…
                        // méthode toString - redéfinition
public String toString()
      { return (ref + "\t"+ libellé+ "\t"+ prix);}
}
```

Héritage – redéfinition de la méthode toString

```
public class Produit-Alimentaire extends Produit
{  Date DatFab, DatExp ; String Conditions; // attributs supplémentaires
                                        //constructeur
public Produit-Alimentaire (String ref, String lib, float px, Date DF, Date DE, String  cond)
{ …}
                                        // méthode modifier
public void modifier (String newlib, float newpx, Date newDF, Date newDE)
{ …}
                        // méthode setCond  qui n'existe pas dans la superclasse
public void  setCond (String  cond)  { Conditions = cond;}
                                    // méthode afficher -redéfinition
public void afficher()  {…}

                                    // méthode toString - redéfinition
public String toString()
      { return (super.toString() + "\t"+ Conditions );}
                    // appel de la méthode toString de la classe Produit
}
```

Héritage – la méthode toString redéfinie

```
class ProgProduit
{ public static void main (String arg[ ])
  {Produit P = new Produit ("Ref012", "savon liquide", 200.00)
   System.out.println (P.toString());
    // affiche      Ref012        savon liquide       200.00

   Date DF = new Date (3, 2, 2017) ; Date DF = new Date (3, 3, 2017) ;
 Produit-Alimentaire PA = new Produit-Alimentaire ("Ref028", "Lait", 90.00, new
DF, DE, "Sec-frais-10°C")
System.out.println (PA.toString());
   // affiche      Ref028       Lait       90.00       Sec-frais-10°C
 }
```

Héritage – redéfinition de la méthode toString

Depuis Java 5, on peut annoter par **@Override** les redéfinitions de méthodes.

Permet de repérer des fautes de frappe dans le nom de la méthode : le compilateur envoie un message d'erreur si la méthode ne redéfinit aucune méthode de la superclasse.

Le modificateur **« final »**

- Si une méthode est déclarée **final** dans une classe, il est impossible de la redéfinir dans les classes dérivées.

- Si une classe A est déclarée **final**, il est impossible de la dériver (i.e impossible de créer des sous-classes de A)

Héritage – redéfinition de la méthode toString

```
public class Fruit
{ private String nom;  private String vitamine; private String couleur;

                                        //constructeur
public Fruit (String n, String v, String c)
        { nom = n; vitamine = v; couleur = c;}
                                         // méthodes
 public String getVitamine ()
       { return (vitamine);}

            // redéfinition de la méthode toString de la classe Object
@Override public String toString()
{return (" Il s'agit d'un fruit appelé " + nom+ " de couleur "+ couleur
+"riche en " + vitamine ) ; }
}
```

Héritage – redéfinition de la méthode toString

```
class ProgFruit
{ public static void main (String arg[ ])
{ Fruit F = new Fruit ("Fraise", "Vitamine C", "rouge");
  System.out.println (F.toString());
  // on peut écrire tout simplement
  System.out.println (F); }   // correspond à un appel à toString
}
```

L'affichage donnera:

Il s'agit d'un fruit appelé Fraise de couleur rouge riche en Vitamine C

Héritage – Transtypage

Le transtypage (conversion de type ou ***cast*** en anglais) consiste à modifier le type d'une variable, d'une expression ou d'un objet.

Transtypage entre types d'objets

Il faut que les objets à transtyper soient liés par l'héritage

Transtypage implicite

Transtypage explicite

Transtypage implicite (upcasting)

Classe fille vers classe mère

Utiliser une **référence** de la classe mère pour désigner un objet de la classe fille, c'est toujours possible car:

Un objet de type sous-classe (classe fille) **est un** objet de type superclasse (classe mère)

Un objet de type Carré **est un** objet de type Forme
Un objet de type Cercle **est un** objet de type Forme

Un objet de type Produit-Alimentaire **est un** objet de type Produit

Transtypage implicite

Classe fille vers classe mère

On peut alors écrire:

Produit-Alimentaire PA ; Produit P ; *//déclaration de références*

PA= **new** Produit-Alimentaire ("Ref006", "Lait", "90.0", DF, DE, « sec-frais-10°C) ;

// on crée un objet PA de la classe Produit-Alimentaire

P = PA ; *//affectation de références*

// une référence de Produit peut référencer un objet de type Produit-Alimentaire

On peut aussi écrire:

Produit P = new **Produit-Alimentaire** ("Ref006", "Lait", "90.0", DF, DE, « sec-frais-10°C) ;

Transtypage implicite

Classe fille vers classe mère

Sur un autre exemple:

```
Point P1 = new Point (3, 2) ; Point P2 = new Point (-5, 0); Point P3 = new Point(0,0);
   Forme CA1 = new Carré (P1, "bleu", 4.0);
   Cercle  C1   = new  Cercle (P2, " noir ", 2.5);
   Forme C2 = new  Cercle (P1, " vert ", 5.0);
   Carré  CA2 = new Carré (P3, "noir ", 6.2);
```

On peut regrouper ces objets dans une même structure (comme un vecteur) dont les éléments seront de type **Forme**:

```
Forme T[] = new Forme [5]; //vecteur de références vers des objets de type Forme
T[0] = CA1;  T[1] = C1; T[2] = C2; T[3] = CA2; //affectations de références
 T[4] = new Carré (P2, "orange", 1.5);
                        // T[4] est une référence de type superclasse (Forme)
```

Transtypage implicite

Classe fille vers classe mère

Suite de l'exemple:

```
//affichage des objets du vecteur
For (int i = 0; i <5; i++)
    T[i].afficher();
        //selon le type de l'objet, la méthode afficher de la classe Carré ou
          Cercle sera invoquée, c'est une méthode redéfinie
```

L'opérateur instanceof

instanceof appliqué à une référence d'objet permet de dire si l'objet est du type (nom de classe) spécifié ou non.

Syntaxe if (référence d'objet **instanceof** nom de classe = =true) …

Par exemple (diapo précédente)

if (C1 **instanceof** Cercle) → true
if (CA1 **instanceof** Forme) → true
if (C1 **instanceof** Forme) → true
if (CA1 **instanceof** Carré) → true
if (CA1 **instanceof** Cercle) → **false**
if (C1 **instanceof** Carré) → **false**

L'opérateur instanceof

```
// par exemple, zoomer  à 75% les objets du vecteur  T  qui sont de type Cercle
    For (int i = 0; i <5; i++)
        if (T[i] instanceof Cercle)    T[i].zoomer (0.75);

// par exemple, afficher la surface des objets du vecteur  T  qui sont de type Carré
     For (int i = 0; i <5; i++)
        if (T[i] instanceof Carré)
                System.out.println ("Surface de l'objet n° "+ i  +":"+  T[i].surface());
```

La méthode getClass

getClass définie dans la classe Object de java.lang permet de retourner la classe d'un objet (sa propre classe pas sa superclasse)

Syntaxe **Class** nomdeClasse = référence d'objet. **getClass()**

Class A = C1.getClass(); → class Cercle *//C1 est de type Cercle*
Class B = CA1.getClass(); → class Carré

Dans la classe Class de java.lang sont définies des méthodes qui s'appliquent à un objet de type Class (retourné par getClass):

getName () : permet de retourner le nom complet de la classe (package.nomclasse)
getSimpleName () : permet de retourner le nom de la classe(nomclasse)

Les méthodes getClass, getName, …

Par exemple

Class A = C1.getClass(); → class Cercle
A.**getName ()** : retourne le nom de la classe (avec le package)
A.**getSimpleName () : retourne le nom de la classe** → Cercle

Class B = CA1.getClass(); → class Carré
B.**getSimpleName () : retourne le nom de la classe** → Carré

La méthode getClass

Autre exemple:

Scanner e = new scanner (System.in);

Class X = e.**getClass()** → class Scanner

X.getName → java.util.Scanner

X.getSimpleName → Scanner

Méthodes de la class **Class**

D'autres méthodes permettent de connaitre la structure interne d'une classe, retourner les noms d'attributs/méthodes, constructeurs, la classe mère d'une classe via une référence de type **Class**.

getDeclaredFiels()
getDeclaredMethods()
getDeclaredConstructors ()
getSuperclass ()