جامعة هواري بومدين
للعلوم والتكنولوجيا
U S T H B

Chapitre 5 (suite – Part3)

Héritage et Polymorphisme

S. BOUKHEDOUMA

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

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 *//CA1 est de type 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 ()

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 explicite (downcasting)

Classe mère vers classe fille

Utiliser une **référence** de la classe fille pour désigner un objet de la classe mère, c'est possible moyennant un **cast explicite**

Forcer une référence de type super-classe à « devenir » une référence de type sous-classe.

Convertir une référence de type super-classe vers une référence de type sous-classe.

Un objet de type Forme **n'est pas forcément** un carré
Un objet de type Produit **n'est pas forcément un** produit alimentaire

Syntaxe du cast: **(type sous-classe)référence de superclasse**

Transtypage explicite (downcasting)

Classe mère vers classe fille

```
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);}
}
```

Transtypage explicite (downcasting)

Classe mère vers classe fille

```
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()
    {…}
}
```

Transtypage explicite (downcasting)

Classe mère vers classe fille

Exemple

```
Produit P;
  ((Produit-Alimentaire)P).setCond("abri-lumière,20°C");  // cast explicite
//on accède à une méthode spécifique à la sous-classe via une référence de type superclasse
```

Si on avait écrit…

```
P. setCond ("abri-lumière,20°C");    // sans effectuer un cast explicite
```

le compilateur génère une erreur car P (de type superclasse) n'a pas accès à une méthode spécifique (ou un attribut spécifique) de la classe Produit-Alimentaire (la sous-classe).

Transtypage explicite (downcasting)

Classe mère vers classe fille

Exemple *de transtypage explicite impossible*

créer un objet Produit (classe mère) à partir d'une référence de type Produit-Alimentaire (classe fille)

```
Produit-Alimentaire PA = new Produit ("Ref008", "XXX", 1250.0);
                    // impossible → erreur
```

Transtypage explicite (downcasting)

Classe mère vers classe fille

Exemple *de transtypage explicite impossible*

Le *downcasting ne permet pas de convertir une instance d'une superclasse en une instance* d'une sous-classe !

```
class A {…}
class B extends A {…}
A a = new A();   // on a créé une instance de type A (superclasse)
B b = (B)a;  // on force l'objet a de type A (superclasse) à être de type B (sous-classe)
```

ce code compile, mais plantera à l'exécution

Liaison dynamique

La liaison dynamique est une notion liée au polymorphisme; aux méthodes redéfinies dans les sous-classes.

La méthode à exécuter sur un objet est déterminée au moment de l'exécution du code et non au moment de la compilation (liaison statique)

A l' invocation d'une méthode redéfinie, le choix de l' implémentation à exécuter ne se fait pas en fonction du type déclaré de la référence à l'objet, mais en fonction du type réel de l'objet. Le type réel de l'objet est celui qui a été utilisé pour sa création (new <type>…).

Exemple: Liaison dynamique

```
public class A { ... // méthode Meth()}

public class B extends A
{ ... // méthode Meth() redéfinie dans B}

public class C extends A
{ ... // méthode Meth() redéfinie dans C}

public class Test
{public static void main (String args [])
{ // on crée un tableau tab de trois références de type A
A tab [] = new A[3];
```

Exemple: Liaison dynamique

```
tab [0] = new B(…); // on crée trois objets de type B, A, C stockés dans tab
tab [1] = new A(…);
tab [2] = new C(…);
tab[0].Meth();
tab[1].Meth();
tab[2].Meth();

// on effectue un cast explicite
((A)tab[0]).Meth() ; ((B)tab[1]). Meth(); ((C)tab[1]).Meth() ; }
```

Question: Quelle méthode Meth() sera exécutée à chaque invocation ? Expliquez.

Héritage – redéfinition de la méthode equals

La méthode equals de la classe Object

```
public boolean equals(Object obj) {
  return (this == obj);      // vérifie l'égalité des références
}
```

La méthode equals est redéfinie dans la classe **String** par exemple, pour les chaines de caractères pour vérifier l'égalité des chaines.

Exemple

String S1 = new String("test") ; String S2 = new String("test") ;

If (S1.equals (S2)) → true

If (S1 = = S2) → false

Faire la différence entre la méthode « equals » et le symbole ==

Héritage – redéfinition de la méthode equals

Exemple de la classe Point

```
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;}
```

```
@override  public String toString()
 {return ("("+"x"+","+"y"+ ")"); }
          //redéfinition de la méthode equals
@override  public boolean equals (Object obj)

{if (this == obj)     return true;
   if (obj == null)    return false;
   if (this. getClass() != obj.getClass())
                       return false;
    Point  P = (Point) obj;  // cast explicite
   if (P.x == this.x && P.y == this.y) return true;
              else return false; }

} // fin de la classe Point
```

Héritage – redéfinition de la méthode equals

La méthode equals de la classe Object

Dans la classe Produit, on pourra écrire (rédéfinir la méthode equals):

```
…
@override  public boolean equals (Object obj)
 {if (this == obj)     return true;
  if (obj == null)    return false;
  if (this. getClass() != obj.getClass())
                      return false;
   Point  P = (Produit) obj;  // cast explicite
  if (P.ref == this.ref) return true;
            else return false; }
```

L'utilisateur peut définir sa propre méthode **equals** sans redéfinir celle de la classe Object.

public boolean equals (Produit P)

On aura alors ***une surcharge*** de la méthode equals

Héritage – méthode de classe (static)

Cacher une méthode de classe

- Redéfinir une méthode de classe (déclarée **static**) , c'est **cacher la méthode** (en anglais hide).

- Contrairement aux méthodes d'instance, les appels aux méthodes de classe sont résolus *statiquement à* la compilation (pas de liaison dynamique).

- Si une instance est utilisée pour l'invocation, la classe utilisée sera celle du **type déclaré** et non du type réel de l'objet.

Il n'y a pas de polymorphisme sur les méthodes de classe (les méthodes static)

Héritage – Avantages

- L'héritage **supprime**, en grande partie, **les redondances** dans le code des applications.
- l'héritage **favorise la réutilisation** de classes
- Possibilité de **rajouter facilement une classe**, et ce à moindre coût, puisque l'on peut réutiliser le code des classes parentes.
- Si un comportement n'a pas été prévu dans une classe donnée, une fois qu'il est ajouté, **ce comportement sera hérité** dans l'ensemble des sous-classes de la classe en mère.