Programmation Objet. Cours 4

Marie-Pierre Béal

DUT 1

Héritage. Transtypage. Interface. Classe abstraite. La classe `Object`.

Héritage

L'héritage consiste à faire profiter tacitement une classe dérivée D des attributs et des méthodes d'une classe de base B.

Héritage

- La classe dérivée possède les attributs de la classe de base (et peut y accéder sauf s'ils sont privés).
- La classe dérivée possède les méthodes de la classe de base (même restriction).
- La classe dérivée peut déclarer de nouveaux attributs et définir de nouvelles méthodes.
- La classe dérivée peut redéfinir des méthodes de la classe de base. La méthode redéfinie masque la méthode de la classe de base.
- Dérivation par `extends`.
- Toute classe dérive, directement ou indirectement, de la classe `Object`.
- L'arbre des dérivations est visible dans les fichiers créés par `javadoc`, sous eclipse,...
- La relation d'héritage est transitive.

Exemple 1

```
public class Base {
  private int p = 2;
  int x = 3, y = 5;
  @Override public String toString(){
    return "B "+p+" "+x+" "+y;
  }
  int sum(){return x+y;}
}
public class Der extends Base {
  int z=7;
  @Override public String toString(){
    return "D "+x+" "+y+" "+z;
  }
}
```

Exemple 1

```
public class Test{
  public static void main(String[] args) {
    Base b = new Base();
    Der d = new Der();
    System.out.println(b);
    System.out.println(b.sum());
    System.out.println(d);
    System.out.println(d.sum());
  }
}
```

Le résultat est :

```
B 2 3 5
8
D 3 5 7
8
```

Exemple 2

```
public class Segment {
   private Pixel start;
   private Pixel end;
   public Segment(Pixel start, Pixel end){
      this.start = start;
      this.end = end;
   }
   @Override public String toString() {
       return (start.toString() + end.toString());
  }
}
```

Exemple 2

```
public class ColoredSegment extends Segment{
 private final String color;
 public ColoredSegment(Pixel start, Pixel end, String color)
    super(start,end); //en premiere ligne du constructeur
    this.color = color;
 }
 @Override public String toString() {
    return(super.toString() + color);
 }
 public String getColor(){
    return color;
 }
}
```

Exemple 2

```
public class Test{
  public static void main(String[] args) {
    Pixel p1 = new Pixel(0,0);
    Pixel p2 = new Pixel(1,1);
    ColoredSegment s = new ColoredSegment(p1,p2,"red");
    System.out.println(s);
  }
}
```

Exemple 2

Dans l'exécution d'un constructeur, le constructeur de la classe de base est exécuté en premier. Par défaut, c'est le constructeur sans argument de la classe de base. On remonte récursivement jusqu'à la classe `Object`.

Dans un constructeur d'une classe dérivée, l'appel d'un autre constructeur de la classe de base se fait au moyen de `super(...)`. Cette instruction doit être la première dans l'écriture du constructeur de la classe dérivée. Si cette instruction est absente, c'est l'instruction `super()` par défaut qui est exécutée en premier.

Transtypage

Le transtypage donne la possibilité de référencer un objet d'un certain type avec un autre type. Le transtypage

- modifie le type de la référence à un objet ;
- n'affecte que le traitement des références : ne change jamais le type de l'objet ;
- est implicite ou explicite.

En Java,

- le sous-typage coïncide avec la dérivation.
- Il n'y a pas de relation de sous-typage entre types primitifs et types objets.

Sous-typage

On parle de sous-typage en Java quand une variable de type B contient implicitement une référence à un objet d'une classe dérivée D de B. Ce sous-typage est implicite.

```
public class Test{
  public static void main(String[] args) {
    Pixel p1 = new Pixel(0,0);
    Pixel p2 = new Pixel(1,1);
    Segment s1 = new ColoredSegment(p1,p2,"red");
    Object t = s1;
    ColoredSegment s2 = s1; //erreur
    ColoredSegment s3 = (ColoredSegment)s1; //ok
  }
}
```

"Variables have type, objects have class"

instanceof

On peut tester le type d'un objet à l'aide de `instanceof` :

```
public class Test{
  public static void main(String[] args) {
    Pixel p1 = new Pixel(0,0);
    Pixel p2 = new Pixel(1,1);
    Segment s = new ColoredSegment(p1,p2,"red");
    if (s instanceof Object) {...} // vrai
    if (s instanceof Segment) {...} // vrai
    if (s instanceof ColoredSegment) {...} // vrai
    if (s instanceof Pixel) {...} // faux
    if (null instanceof Object) {...} // false
  }
}
```

L'usage de `instanceof` est restreint à des cas bien particuliers. Il ne doit pas se substituer au polymorphisme.

Héritage : interfaces

Une interface est une classe

- n'a que des méthodes abstraites et tacitement publiques ;
- et n'a que des données statiques et immuables (`static final`).

Une méthode abstraite est déclarée mais non définie.
Une interface sert à spécifier des méthodes qu'une classe doit avoir, sans indiquer comment les réaliser.

On crée une interface pour manipuler des formes géométriques 2D.

```
interface Shape {
  double getArea();
  String toStringArea();
}
```

Héritage : classes abstraites

Une classe abstraite est une classe qui

- peut avoir des méthodes concrètes ou abstraites.

Une classe abstraite sert en général à commencer les implémentations (parties communes aux classes qui en dériveront).

```
public abstract class AbstractShape implements Shape {
  private double width, height;
  protected AbstractShape(double width, double height) {
    this.width = width;
    this.height = height;
  }
  @Override public String toStringArea() {
    return "L'aire est " + getArea();
  }
```

Héritage : classes concrètes

```
public class Rectangle extends AbstractShape {
  public Rectangle(double width, double height) {
    super(width, height);
  }
  public double getArea() {
    return width * height;
}}
```

```
public class Ellipse extends AbstractShape {
  public Ellipse(double width, double height) {
    super(width, height);
  }
  public double getArea() {
    return width * height * Math.PI/4;
}}
```

Héritage

```
public class Test{
  public static void main(String[] args) {
   Shape r = new Rectangle(6,10);
   Shape e = new Ellipse(3,5);
   System.out.println(r.toStringArea());
   System.out.println(e.toStringArea());
   ArrayList<Shape> list = new ArrayList<Shape>();
   list.add(new Rectangle(6,10));
   list.add(new Ellipse(3,5));
   ...
   for (Shape s:list)
      System.out.println(s.getArea()); //polymorphisme
  }
}
```

Héritage

```
public class Test{
  public static void main(String[] args) {
   Shape r = new Rectangle(6,10);
   Shape e = new Ellipse(3,5);
   System.out.println(r.toStringArea());
   System.out.println(e.toStringArea());
   List<Shape> list = new ArrayList<Shape>();
   list.add(new Rectangle(6,10));
   list.add(new Ellipse(3,5));
   ...
   for (Shape s:list)
      System.out.println(s.getArea()); //polymorphisme
  }
}
```

La classe java.lang.Object

```
protected Object          clone()
                          throws CloneNotSupportedException
public boolean            equals(Object obj)
protected void            finalize()
public final Class<?>     getClass()
public int                hashCode()
public String             toString()
public final void         notify()
public final void         notifyAll()
public final void         wait()
public final void         wait(long timeout)
public final void         wait(long timeout, int nanos)
```

Affichage d'un objet et hashCode

La méthode `toString()` retourne la représentation d'un objet sous forme de chaîne de caractères (par défaut le nom de la classe suivi de son `hashCode`) :

```
System.out.println(new Integer(3).toString());
//affiche 3
System.out.println(new Object().toString());
//affiche java.lang.Object@1f6a7b9
```

La valeur du hashCode peut être obtenue par la méthode `hashCode()` de la classe `Object`.

Égalité entre objets et hashCode

La méthode `equals()` de la classe `Object` détermine si deux objets sont égaux. Par défaut deux objets sont égaux s'ils sont accessibles par la même référence.

Toute classe hérite des deux méthodes de `Object`

```
public boolean equals(Object o)
public int hashCode()
```

qui peuvent être redéfinies en respectant the "Object Contract".

Le "contrat objet"

Une classe peut redéfinir la méthode `equals()`. Il faut alors aussi redéfinir la méthode `hashCode()`.

- `equals` doit définir une relation d'équivalence ;
- `equals` doit être consistente (plusieurs appels donnent le même résultat) ;
- `x.equals(null)` doit être faux (si `x` est une référence) ;
- `hashCode` doit donner la même valeur sur des objets égaux par `equals`.

Le "contrat objet"

```
interface Shape {
  double getArea();
  String toStringArea();
}
public class Rectangle implements Shape {
  private final int width;
  private final int height;
  public Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
  }
   ...
}
```

Le "contrat objet"

```
public class Rectangle implements Shape {
  ...
  @Override public boolean equals(Object o) {
    if (!(o instanceof Rectangle))
      return false;
    Rectangle rarg = (Rectangle)o;
    return (width == rarg.width)
      && (height == rarg.height);
  }

  @Override public int hashCode() {
    return new Integer(width).hashCode()
         + new Integer(height).hashCode();
  }
}
```

Le "contrat objet"

ou bien

```
public class Rectangle implements Shape {
  ...
  @Override public boolean equals(Object o) {
    if (!(o instanceof Rectangle))
      return false;
    Rectangle r = (Rectangle)o;
    return (width == r.width)
      && (height == r.height);
  }
  @Override public int hashCode() {
    return Objects.hash(width,height);
  }
}
```

On utilise ici la méthode `hash` de la classe outils `java.util.Objects`.