Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

ATL – Ateliers Logiciels

Orienté objet

Héritage, polymorphisme

> **Consignes**
>
> Vous pouvez télécharger les ressources du TD via le lien `https://git.esi-bru.be/ATL/oo_heritage/repository/archive.zip`.
> N'oubliez pas de vous **connecter** au serveur `git.esi-bru.be` pour avoir accès au téléchargement.

1 Héritage

L'héritage est une technique qui permet à une classe, dite sous-classe ou classe enfant, de récupérer (hériter) le type, les attributs et les méthodes d'une classe, dite super-classe ou classe parent.

Reprenons notre classe `Point` légèrement modifiée (la méthode `move` retourne le point).

```java
package esi.atl.oo_inheritance;

public class Point {
   private double x;
   private double y;

   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }

   public double getX() { return x; }
   public double getY() { return y; }

   public Point move(double dx, double dy) {
      x += dx;
      y += dy;
      return this;
   }

   @Override
   public String toString() {
      return "("+x+", "+y+")";
   }
}
```


Version du document
19 septembre 2021

Nous avons ajouté une sous-classe `ColoredPoint`

```
1  package esi.atl.oo_inheritance;
2
3  public class ColoredPoint extends Point {
4     private int color;  // A 32-bit integer of the form: 0xRRGGBBAA
5                         // where AA represents the alpha value
6
7     public ColoredPoint(double x, double y, int color) {
8        super(x, y);
9        this.color = color;
10    }
11
12    public int getColor() { return color; }
13
14    @Override
15    public String toString() {
16       return super.toString() + " - "+ String.format("%08X", color);
17    }
18 }
```

et une méthode principale pour tester nos classes.

```
1  package esi.atl.oo_inheritance;
2
3  public class TestPoints {
4
5     public static void main(String[] args) {
6        ColoredPoint p = new ColoredPoint(2, 4, 0xFF0000FF);
7        p.move(1, 2);
8        System.out.println(p);
9        System.out.println("x: " + p.getX());
10       System.out.println("color : " + String.format("%08X", p.getColor()));
11    }
12 }
```

La classe `ColoredPoint` étend (`extends`) la classe `Point`. Elle en hérite son type (`Point`), les attributs (`x` et `y`), et ses méthodes (`move()`, `getX()`, `getY()` et `toString()`). C'est pourquoi nous pouvons écrire les lignes 7 à 9 dans la méthode `main`.

De plus la classe définit un nouvel attribut, `color`, et une nouvelle méthode publique, `getColor()`. Ce qui nous autorise à écrire la ligne 10 de la méthode `main`.

Question 1

1. Qu'affiche le programme `TestPoints` ?
2. Quelle(s) erreur(s) de compilation obtient-on si on modifie la première ligne comme suit :

   ```
   Point p = new ColoredPoint(2, 4, 0xFF0000FF);
   ```

   Quelle ligne pose problème et pourquoi ?

   A-t-on toujours une erreur si on supprime (ou commente) cette ligne problématique ?
3. Peut-on ajouter la ligne suivante au `main` ?

   ```
   ColoredPoint p2 = new Point(2, 4);
   ```

   Pourquoi ?

4. Peut-on remplacer le contenu de la méthode `toString()` de `ColoredPoint` via :

```
return this.x +" - "+ this.y +" - "+ this.color;
```

5. Quelle erreur obtient-on si l'on modifie la déclaration de la classe `Point` comme suit :

```
public class Point extends ColoredPoint {
    ...
}
```

6. Quelle erreur obtient-on si on déclare `final` la classe `Point` :

```
public final class Point {
    ...
}
```

7. Remettez le code dans son état d'origine.

Théorie

Une classe `C` héritant d'une classe `A` est appelée classe enfant, ou sous-classe, `A` sera appelée classe parent ou super-classe. Par extension, on parlera de descendant pour désigner une classe `C` qui est une sous-classe d'une sous-classe (etc.) de la classe `A`, appelée ancêtre.

Héritage

En java, l'héritage s'implémente via le mot-clef `extends` (étendre).
Soient une classe `A`, et une sous-classe `C` (classe enfant, en anglais *child*) directe ou indirecte (p.ex. sous-classe d'une sous-classe de `A`).

- Toute instance de la classe `C` est également de type `A`. La sous-classe `C` hérite du *type* de la super-classe. On peut donc référencer une instance de la classe `C` avec une variable de type `A` :

```
A a = new C(...);
```

- Les instances de la sous-classe `C` héritent les méthodes et les attributs de la classe `A`.
  Si `foo()` est une méthode accessible de la classe `A` on peut écrire :

```
C c = new C();
c.foo();
```

- La sous-classe peut définir de nouvelles méthodes et de nouveaux attributs.
- Le mot-clef `super` permet de faire un appel à un membre (méthode ou attribut) de la classe parent (super-classe directe), pourvu qu'il soit accessible.
- Il ne peut y avoir de cycle dans la relation an/cêtre-descendant : Si `C` est un descendant de `A`, `A` ne peut être un descendant de `C`.

Une classe déclarée `final` ne peut pas être sous-classée.

2 La classe `Object`

Considérons les classes `Point` et `ColoredPoint` et la méthode principale.

Question 2

1. Peut-on ajouter la ligne suivante au `main` ?

   ```
   Object p3 = new Point(2, 4);
   ```

   Pourquoi ?
2. Peut-on ajouter la ligne suivante au `main` ?

   ```
   Object p4 = new ColoredPoint(2, 4, 0xFF0000FF);
   ```

   Pourquoi ?
3. Peut-on ajouter la ligne suivante au `main` ?

   ```
   p.hashCode();
   ```

   Où est définie cette méthode ?

   Pourquoi peut-on l'appeler sur un objet de type `ColoredPoint` ?
4. Remettez le code dans son état d'origine.

La classe `Point` ne déclare pas de super-classe (pas de mot-clef `extends` dans sa déclaration), elle étend par défaut la classe `Object`. Sa déclaration est équivalente à :

```
public class Point extends Object {
   ...
}
```

La classe `Point` hérite donc du type, des méthodes et des attributs de la classe `Object`. La classe `ColoredPoint` hérite également, au travers de la classe `Point`, de tous les éléments de la classe `Object`.

La javadoc de la classe `Object` indique :

> Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Consultez la javadoc [1] et donnez la liste des méthodes d'`Object`.

La classe `Object`

- La classe `Object` est la racine de la hiérarchie de classe. Elle est l'ancêtre de toutes les classes Java.
- Une classe qui n'étend pas explicitement une autre classe, est (implicitement) une sous-classe directe de la classe `Object`.
- Toute classe est un descendant de la classe `Object`.
- Tout objet est de type `Object`. Pour toute classe `A`, on peut écrire :

  ```
  Object o = new A(...);
  ```

1. https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Object.html

3 Constructeur

Considérons les classes `Point` et `ColoredPoint`.

Question 3

1. Quelle erreur de compilation obtenez-vous si vous ajoutez

   ```
   System.out.println("test");
   ```

   comme première ligne du constructeur de la classe `ColoredPoint` ?
   (retirez cette ligne après avoir fait le test)
2. Quelle erreur de compilation obtenez-vous si vous supprimez la ligne

   ```
   super(x,y);
   ```

   dans le constructeur de la classe `ColoredPoint` ? A quoi sert cette ligne ?
3. Après avoir supprimé la ligne au point précédent, ajoutez dans la classe `Point` le constructeur suivant :

   ```
   public Point() {
      this(0,0);
   }
   ```

   A-t-on toujours la même erreur qu'au point précédent ?
4. Remettez le code dans son état d'origine.

Considérons maintenant le test suivant :

```
1  package esi.atl.oo_inheritance;
2
3  class A {
4     A () {
5        System.out.println("constructor of A");
6     }
7  }
8  class B extends A {
9     B () {
10       System.out.println("constructor of B");
11    }
12 }
13 class C extends B {
14    C () {
15       System.out.println("constructor of C");
16    }
17 }
18 public class TestConstructor {
19
20    public static void main(String[] args) {
21       new C();
22    }
23 }
```

Question 4

1. Qu'affiche ce programme ?
2. Supprimez le constructeur de la classe `C`, qu'affiche maintenant le programme ?
3. Après avoir remis le constructeur de la classe `C`, ajoutez dans chaque constructeur un appel explicite au constructeur de la super-classe (`super();`). Vérifiez que l'effet est bien identique.

4. A votre avis, quelles sont les constructeurs de la classe `Object` ? Vérifiez en consultant la javadoc.
5. Remettez le code dans son état d'origine.

Théorie

Constructeur et héritage

- Tout constructeur commence par un appel à un constructeur de la même classe ou de la classe parent `this(...)` ou `super(...)`.
- `this(...)` est un appel explicite à un constructeur de la même classe.
- `super(...)` est un appel explicite à un constructeur de la classe parent.
- Si aucun appel explicite (`this(...)` ou `super(...)`) n'est présent, un appel implicite au constructeur sans paramètre de la classe parent est ajouté. Ceci est équivalent à ajouter l'instruction suivante en tout début de constructeur.

```
super();
```

- Les appels `this(...)` ou `super(...)` ne peuvent figurer que comme *première* instruction du constructeur.

Exécution des constructeurs

Lorsqu'un constructeur est appelé, l'exécution se déroule comme suit :

- la machine virtuelle Java réserve de la mémoire pour stocker l'objet ;
- cette mémoire est initialisée : les champs sont mis à 0, `false` ou `null` suivant leur type ;
- le constructeur invoqué est appelé (mais pas encore exécuté jusqu'au bout) ;
- si ce constructeur appelle un autre constructeur de la même classe, alors ce dernier est appelé ;
- si ce constructeur appelle un constructeur de sa super-classe (de manière implicite ou explicite), alors ce dernier est appelé ;
- une fois que la chaîne d'appel des constructeurs a été épuisée,
  - alors les initialiseurs de champs sont appelés ;
  - (les blocs non statiques sont exécutés ;)
  - le(s) constructeur(s) est(sont) exécuté(s) ;

  'en descendant' dans la hiérarchie (ou plus exactement dans la chaîne d'appel).

4 Redéfinition de méthodes

Considérons la classe `PinnablePoint` qui définit un point que l'on peut fixer (*pin* en anglais). Une fois fixé, le point ne bouge plus.

```
1  package esi.atl.oo_inheritance;
2
3  public class PinnablePoint extends Point {
4
5     private boolean pinned; // once pinned cannot move.
6
7     public PinnablePoint(double x, double y) {
8        super(x, y);
9        this.pinned = false;
10    }
11
12    public void pin() { pinned = true; } // once pinned, no way to unpin.
13
14    @Override
15    public Point move(double dx, double dy) {
16       if(!pinned) {
17          super.move(dx, dy);
18       }
19       // do nothing if pinned.
20       return this;
21    }
22
23    @Override
24    public String toString() {
25       return super.toString() + " - "+ (pinned? "pinned": "not pinned");
26    }
27
28    public static void main(String[] args) {
29       Point p = new PinnablePoint(0, 0);
30       System.out.println(p);
31       p.move(1, 1);
32       PinnablePoint pp = (PinnablePoint)p; // we know it is a PinnablePoint
33       pp.pin();
34       p.move(1,1);
35       System.out.println(p);
36    }
37 }
```

Question 5

1. Qu'affiche ce programme ?
2. Selon vous, quelle méthode `move` est exécutée :
   - ▷ celle de `Point` car la variable est déclarée de type `Point` ?
   - ▷ celle de `PinnablePoint` car l'objet référencé par la variable à ce moment-là est de type `PinnablePoint` ?
3. Ajoutez une exception à la méthode `move`.

```
@Override
public Point move(double dx, double dy) throws Exception {
   if(!pinned)
   super.move(dx, dy);
   else throw new Exception("Point is pinned, cannot move anymore");
   return this;
}
```

Quelle erreur obtenez-vous ?

4. Remplacez `Exception` par une `IllegalStateException`. Avez-vous toujours une erreur ? Pourquoi ?
5. Retirez la clause `throws` (mais gardez le `throw`), avez–vous une erreur ? [2]
6. Remplacez le type de retour par `PinnablePoint`, avez-vous une erreur ?
7. Remplacez le type de retour par `Object`, obtenez-vous une erreur ? pourquoi ?
8. Remplacez le modificateur d'accès `public` par `protected`, quelle erreur obtient-on ?
9. Dans la méthode `move`, que fait l'appel `"super.move(...);"` ?
10. Remettez le code dans son état d'origine.

Théorie

Redéfinition de méthodes

- Une sous-classe peut redéfinir des méthodes de ses ancêtres et le signaler [a] via l'annotation `@Override`.
- La méthode doit avoir :
  - le même nom
  - les mêmes paramètres : mêmes types et dans le même ordre
  - le type de retour doit-être le même ou d'une sous-classe du type de retour original
  - ne pas déclarer le lancement de nouvelles exceptions (mais peut en déclarer moins)
  - la visibilité (l'accès) ne peut être restreinte (p.ex. une méthode `public` doit rester `public`), mais peut être élargie (p.ex. `protected` vers `public`).
- La redéfinition doit respecter le contrat établi par la méthode redéfinie. Ce contrat spécifie ce qui est attendu lors d'un appel : la visibilité, le type de retour et les éventuels types d'exceptions.

a. https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Override.html

2. Si vous ne comprenez pas ce point il est important d'aller revoir les exceptions (https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html) un jour (pas maintenant).

5 Polymorphisme

Le polymorphisme en Java désigne la possibilité qu'une variable d'un type déclaré référence (à des moments différents de l'exécution) des objets de types différents. De plus, un même appel de méthode sur une telle variable ne déclenche pas toujours l'exécution du même code. Cela dépendra du type de l'objet référencé.

Considérons le programme `TestPolymorphism`

```
1  package esi.atl.oo_inheritance;
2  import java.util.*;
3
4  public class TestPolymorphism {
5
6     public static void main(String[] args) {
7        List<Point> points = new ArrayList<>();
8        points.add(new Point(0,0));
9        points.add(new ColoredPoint(2, 4, 0xFF0000FF));
10       points.add(new PinnablePoint(1, 1));
11       moveAndPrintPoints(points);
12    }
13
14    static void moveAndPrintPoints(List<Point> list) {
15     if(list.isEmpty()) {
16       return; // base case
17     }
18
19     Point p = list.remove(0);
20     p.move(1,1);
21     System.out.println(p);
22
23     moveAndPrintPoints(list); // recursion
24    }
25 }
```

Question 6

▷ Qu'affiche ce programme ?

Dans cet exemple, la méthode `moveAndPrintPoints(..)` reçoit une liste de `Point`. Cette méthode parcourt récursivement la liste et assigne un à un les points de celle-ci à la variable `p` qui est de type `Point`.

Pour générer l'affichage, un appel (polymorphique) est fait aux méthodes `move` et `toString`. Le code des méthodes `move` et `toString` réellement exécuté dépendra du type de l'objet référencé. Il se trouvera dans la classe `Point`, ou dans la classe `ColoredPoint` ou encore dans la classe `PinnablePoint` selon le type de l'objet référencé à ce moment-là.

L'héritage (en particulier du type) et la redéfinition sont des éléments essentiels de l'implémentation du polymorphisme en Java.

6 Interface

Les interfaces en java permettent de définir des contrats que les classes doivent respecter. Ce contrat a la forme d'une liste de méthodes.

Soit l'interface `Movable` qui définit une unique méthode `move`.

```
1 package esi.atl.oo_inheritance;
2
3 public interface Movable {
4     Movable move(double dx, double dy);
5 }
```

Question 7

1. Ajoutez le mot clef `protected` devant la méthode `move`. Que constatez-vous ?
2. Remplacez `protected` par `public`. Que constatez-vous ?
3. Modifiez la déclaration de la classe `Point` afin qu'elle implémente l'interface `Movable` :

   ```
   public class Point implements Movable {
       ...
   }
   ```

4. Remplacez `Point` par `Movable` dans les déclarations de `TestPolymorphisme`. Qu'affiche le programme ?

L'intérêt des interfaces est de les utiliser comme type de variable/attribut/paramètre/-retour (p.ex. l'interface `List`). Alliées au polymorphisme, elles donnent plus de souplesse aux implémentations.

Interface

- ▷ Une interface est déclarée par le mot clef `interface`.
- ▷ Une classe déclare implémenter une ou plusieurs interfaces avec le mot clef `implements`.
- ▷ Une interface définit un contrat sous la forme d'une liste de signature de méthodes automatiquement publiques.
- ▷ Chaque méthode ne contient ni code ni bloc d'instruction et se termine par un point virgule.
- ▷ Depuis Java 8, des méthodes dites `default` peuvent être ajoutées, avec une implémentation par défaut [a] .
- ▷ Les attributs sont automatiquement `public` et `static`.

a. https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

7 Classe abstraite

Une classe doit être déclarée abstraite si toutes ses méthodes ne sont pas implémentées. Notez également que, même si toutes les méthodes d'une classe sont implémentées, une classe peut être déclarée abstraite. Afin de bien comprendre l'utilisation des classes abstraites, consultez la documentation [3] .

3. https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html