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

ATL – Ateliers Logiciels

Orienté objet

Les bases

Consignes

Dans ce document vous trouverez un *rappel* des notions de classe, d'objet et d'énumération en Java. Pour ce faire, vous pouvez télécharger les ressources du TD via le lien `https://git.esi-bru.be/ATL/oo_base/repository/archive.zip`.
N'oubliez pas de vous **connecter** au serveur `git.esi-bru.be` pour avoir accès au téléchargement.

1 Classes

Dans le code ci-dessous, on définit une classe `Point` qui représente un point dans le plan.

```java
package esi.atl.oo_base;

public class Point {

   private double x;
   private double y;

   public Point() {
      this(0,0);
   }
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
   public double getX() {return x;}
   public double getY() {return y;}
   public void move(double dx, double dy) {
      x += dx;
      y += dy;
   }
   @Override
   public String toString() {
      return "("+x+", "+y+")";
   }
}

class TestPoint {
   public static void main(String args[]) {
      Point p = new Point();
      System.out.println(p);
      p.move(2,2);
      System.out.println(p);
   }
}
```


Version du document
20 septembre 2021

Cette classe possède :

- deux attributs, `x` et `y`, de type double.
- un constructeur prenant deux paramètres qui initialise les attributs `x` et `y`.
- un constructeur sans paramètre.
- une méthode `move` qui permet de déplacer le point.
- une redéfinition de la méthode `toString` de la classe `Object`.

La classe `Point` est `public` et doit donc se trouver dans un fichier nommé `Point.java`. La classe `TestPoint` se trouve dans le même fichier, ce qui est possible car elle n'est pas `public`. La méthode principale se trouve dans la classe `TestPoint`.

Question 1

1. Qu'affiche ce programme (en réfléchissant, sur papier, sans l'exécuter) ?
2. Si vous changez la visibilité de la classe `TestPoint` en public, quelle erreur le compilateur rapporte-t-il ?
3. Si vous changez la visibilité de la classe `Point` en visibilité par défaut (retirer le modificateur `public`), quelle erreur obtenez-vous ?
4. Remettez le code dans son état d'origine (la classe `Point` en public et la class `TestPoint` en visibilité par défaut).

Théorie

Visibilité, fichiers

- Le mot clef `public` devant le nom de la classe définit la *visibilité* de la classe. Une classe peut être :
  - `public` : elle peut être utilisée partout, ou bien
  - sans modificateur : elle peut être utilisée uniquement à l'intérieur du package.
- Une classe java `public` doit se trouver dans un fichier portant le nom de la classe suivi de l'extension `.java`.
- Le fichier peut contenir d'autres classes si elles ne sont pas déclarées `public`.

Ceci est valable pour les classes non-imbriquées. La définition de classes imbriquées sera exposée lors d'une séance ultérieure.

2 Membres

On appelle *les membres* d'une classe ses attributs et ses méthodes.

Question 2

1. Ajoutez dans la méthode principale de la classe `TestPoint` la ligne

```
System.out.println(p.x);
```

   Quelle erreur obtient-on ? Remettez ensuite le code dans son état d'origine.

2. Ajoutez la méthode suivante à la classe `Point`

```
public void move(int dx, int dy) {
    System.out.println("méthode move(int, int)");
    x += dx;
    y += dy;
}
```

   Qu'affiche maintenant le programme ?

3. Que se passe-t-il si on tente d'ajouter la méthode (le type de retour est différent mais le nombre et le type des paramètres sont identiques)

```
public boolean move(double dx, double dy) {
    x += dx;
    y += dy;
    return true;
}
```

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

Théorie

Visibilité des membres

Les *modificateurs de visibilité* pour les membres sont :

- ▷ private : le membre est accessible uniquement à l'intérieur de la classe.
- ▷ sans modificateur (appelé *visibilité package*) : accessible dans toutes les classes du même package.
- ▷ protected : accessible dans le package et dans les sous-classes (qui ne sont pas nécessairement dans le même package).
- ▷ public : le membre est accessible partout.

Surcharge d'une méthode

La *surcharge*, aussi appelée *surdéfinition*, d'une méthode est le fait d'utiliser un même nom pour plusieurs méthodes. La surcharge est possible si :

- ▷ le nombre de paramètres est différent, ou bien
- ▷ le nombre de paramètres est le même mais le type d'au moins un des paramètres est différent.

Le type de retour et la visibilité de la méthode n'interviennent donc pas pour déterminer si une méthode est une surcharge d'une autre.

3 Constructeurs

La classe `Point` possède 2 constructeurs, il y a surcharge du constructeur.

Le constructeur `Point(double, double)` initialise les coordonnées `x` et `y` du point avec les valeurs passées en paramètres.

Le constructeur sans paramètre `Point()` fait appel au constructeur `Point(double, double)` en lui passant les valeurs 0 et 0. Cet appel se fait par l'instruction `this(0,0);`.

Question 3

1. Modifiez le constructeur sans paramètre comme suit (ajout de l'appel `println`) :

```
public Point() {
    System.out.println("test");
    this(0,0);
}
```

   Quelle erreur obtient-on ?

2. Supprimez le constructeur sans paramètre de la classe `Point`.

   Quelle erreur de compilation obtient-on ?

   Supprimez l'autre constructeur. La classe `Point` ne définit plus de constructeur.

   Obtient-on toujours une erreur ? Pourquoi ?

   Qu'affiche ce programme ?

   Lorsqu'une classe ne définit aucun constructeur, un constructeur par défaut sans paramètre est créé, c'est ce constructeur qui est utilisé à la première ligne : `Point p = new Point();`

3. Il est possible, mais déconseillé, d'initialiser les attributs avec des valeurs par défaut explicites. Par exemple à partir du code de la question précédente :

```
public class Point {
    private double x = 10;
    private double y = 10;
    ...
}
```

   Qu'affiche maintenant ce programme ?

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

Théorie

Le constructeur est appelé lors de la création d'un nouvel objet. Il permet d'initialiser l'objet et en particulier ses attributs.

Constructeurs

- Les constructeurs portent le même nom que la classe, n'ont pas de type de retour, et peuvent être déclarés (comme les membres) `public`, `private`,`protected` ou `package` (lorsqu'il n'y a pas de modificateur d'accès).
- Une classe peut contenir plusieurs constructeurs (surcharge).
- Un constructeur peut faire appel à un autre constructeur via l'appel `this(...)`. Cet appel doit être la première instruction du constructeur.
- Un constructeur *par défaut* est ajouté à une classe si celle-ci ne possède aucun constructeur. Le constructeur par défaut n'a aucun paramètre.
- Un nouvel objet est créé par un appel au constructeur via le mot-clef `new` :
  1. La mémoire est réservée.
  2. Les attributs sont initialisés avec les valeurs par défaut implicites : 0 pour les nombres, `false` pour les booléens et `null` pour les références (objets et tableaux).
  3. Le constructeur de la classe à instancier est appelé (mais pas encore exécuté jusqu'au bout).
  4. Le constructeur de la classe `Objet` est appelé.
  5. Les initialisations explicites des attributs de la classe `Objet` sont exécutées.
  6. Le constructeur de la classe `Objet` est exécuté.
  7. Les initialisations explicites des attributs de la classe à instancier sont exécutées.
  8. Le constructeur de la classe à instancier est exécuté.

4 Enumérations

Une énumération est une classe dont les seules instances sont pré-définies. Aucune autre instance ne peut être construite.

```
1  package esi.atl.oo_base;
2
3  public enum Direction {
4     NORTH('N', "North"), SOUTH('S', "South"), WEST('W', "West"), EAST('E',
       ↪ "East");
5     private char c;
6     private String str;
7
8     Direction(char c, String str) {
9        this.c = c;
10       this.str = str;
11    }
12    public char getChar() {
13       return c;
14    }
15    @Override
16    public String toString() {
17       return str;
18    }
19 }
```

L'énumération `Direction` contient 4 instances prédéfinies : `NORTH`, `SOUTH`, `WEST`, `EAST`.

Une énumération étant une classe, il est possible d'ajouter des attributs ou des méthodes, ou encore de redéfinir la méthode `toString`. Il est aussi possible d'ajouter un constructeur mais celui-ci ne peut être utilisé que pour initialiser les instances pré-définies car il n'est pas possible de créer de nouveaux objets d'une énumération.

On obtient un tableau contenant toutes les valeurs de l'énumération (dans l'ordre selon lequel elles sont déclarées) via la méthode static `values()`. On peut donc écrire par exemple :

```
for (Direction d : Direction.values()) {
   System.out.println(d);
}
```

5 Encapsulation

[Cette section est largement inspirée de wikipedia]

L'encapsulation en programmation consiste à cacher les données d'une classe aux autres classes, c'est-à-dire, empêcher l'accès aux données (attributs) par un autre moyen que des méthodes.

Reprenons la classe `Point` présentée précédemment.

```java
package esi.atl.oo_base;

public class Point {

   private double x;
   private double y;

   public Point() {
      this(0,0);
   }
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
   public double getX() {return x;}
   public double getY() {return y;}
   public void move(double dx, double dy) {
      x += dx;
      y += dy;
   }
   @Override
   public String toString() {
      return "("+x+", "+y+")";
   }
}

class TestPoint {
   public static void main(String args[]) {
      Point p = new Point();
      System.out.println(p);
      p.move(2,2);
      System.out.println(p);
   }
}
```

Les données (attributs) sont protégées par le mot clef `private`. La seule manière de modifier les attributs (à partir d'une autre classe) est d'y accéder par les constructeurs ou la méthode `move`.

Puisque les attributs `x` et `y` peuvent prendre n'importe quelle valeur, il ne faut faire aucune vérification sur les paramètres dans les constructeurs et la méthode `move`.

Considérons maintenant la classe `Circle` représentant un cercle :

```java
package esi.atl.oo_base;

public class Circle {

   private double radius;
   private Point center;

   public Circle(Point center, double radius) {
      if (radius <= 0) {
         throw new IllegalArgumentException("radius must be positive" +
               ", received: " + radius);
      }
      this.radius = radius;
      this.center = center;
   }

   public void move(double dx, double dy) {
      center.move(dx, dy);
   }

   public double area() {
      return Math.PI * radius * radius;
   }

   public Point getCenter() {
      return center;
   }

   public void scale(double factor) {
      if (factor <= 0) {
         throw new IllegalArgumentException("Scale factor must be positive" +
               ", received: " + factor);
      }
      radius *= factor;
   }

   @Override
   public String toString() {
      return "Circle : [" + center + ", " + radius + "]";
   }
}

class TestCircle {

   public static void main(String args[]) {
      Point p = new Point();
      Circle c = new Circle(p, 5);
      System.out.println(c);
      c.move(2, 5);
      System.out.println(c);
      c.scale(2);
      System.out.println(c);
   }
}
```

Question 4

1. Qu'affiche ce programme ?
2. Combien d'instances différentes de la classe `Point` sont créées dans ce programme ?

Ici le constructeur et les méthodes `move` et `scale` (mise à l'échelle) modifient la valeur des attributs :

- ▷ Le centre peut être quelconque, il n'y a donc aucune vérification à faire pour celui-ci.
- ▷ Le rayon doit être strictement positif, le constructeur et la méthode `scale` vérifient que c'est bien le cas.

Un cercle cohérent est un cercle dont le rayon est positif, notre implémentation assure que c'est toujours le cas. On appelle cela un invariant de la classe `Circle` : l'attribut `radius` est à tout moment strictement positif.

Encapsulation

- ▷ L'encapsulation est implémentée en Java à l'aide des limitateurs d'accès : `public`, `private` :
  - ▷ tous les attributs et certaines méthodes sont déclarées `private`,
  - ▷ les méthodes nécessaires pour interagir avec la classe sont déclarées `public`.
- ▷ Un *invariant de classe* est une propriété que les objets de la classe doivent respecter à tout moment (rayon positif pour un cercle, le mois compris entre 1 et 12 pour une date, etc).
- ▷ L'encapsulation permet à la classe d'avoir un contrôle total sur les modifications des données. Cela permet d'assurer à tout moment la cohérence de ces données, et donc d'assurer que les invariants de la classe soient toujours respectés.
- ▷ Les constructeurs et les méthodes modifiant l'état sont responsables de préserver cette cohérence.

6 Copie défensive

Prenons la classe `TestDefensiveCopy` :

```
class TestDefensiveCopy {

   public static void main(String args[]) {
      Point p = new Point();
      Circle c = new Circle(p, 5);
      System.out.println(c);
      p.move(2, 5); //on bouge le point et non pas le cercle.
      System.out.println(c);
      Point p2 = c.getCenter();
      p2.move(-2, -5);
      System.out.println(c);
   }
}
```

Question 5

1. Qu'affiche ce programme ?
2. Combien d'instances de la classe `Point` et `Circle` sont créées dans ce programme ? Quelles sont les instances référencées par la variable `p` et `p2` dans le main ? Quelle instance référence l'attribut `center` de l'instance `c` créée dans le main ?
3. Ajoutez une copie défensive à la ligne 14 de la classe `Circle` comme suit :

   ```
   this.center = new Point(center.getX(), center.getY()); //copie défensive.
   ```

   Qu'affiche maintenant le programme `TestDefensiveCopy` ?

4. Remplacez la ligne 26 de la classe `Circle` par la ligne suivante :

```
return new Point(center.getX(), center.getY()); //copie défensive
```

   Qu'affiche maintenant le programme `TestDefensiveCopy` ?

5. Après avoir effectué ces modifications au programme :
   Combien d'instances de la classe `Point` et `Circle` sont créées dans ce programme ?
   Quelles sont les instances référencées par la variable `p` et `p2` dans le main ?
   Quelle instance référence l'attribut `center` de l'instance `c` créée dans le main ?

Avec l'implémentation de la classe `Circle` avant modifications, il est possible de déplacer le cercle sans passer par la méthode `move` de la classe `Circle`. Il suffit de conserver une référence au centre du cercle (passée en paramètre au constructeur) ou d'obtenir cette référence via la méthode `getCenter()`. Avec cette référence, bouger le centre permet de déplacer le cercle, sans donc passer par la méthode `move` de `Circle`. [1]

En effectuant une *copie défensive* du centre lors de la construction du cercle et dans la méthode `getCenter()` on assure que l'unique manière de déplacer le cercle est d'utiliser la méthode `move` de `Circle`.

7 Constructeur par copie

Pour effectuer la copie défensive ci-dessus, nous avons récupéré la valeur de chaque attribut (ici `x` et `y`) et appelé le constructeur de `Point` avec ces valeurs. Une manière plus élégante est d'offrir un *constructeur par copie* dans la classe `Point`.

```
public Point(Point p) {
   this(p.x, p.y);
}
```

On peut maintenant remplacer la ligne 14 de la classe `Circle` par :

```
this.center = new Point(center); //copie défensive
```

et la ligne 26 par :

```
return new Point(center); //copie défensive
```

Il est théoriquement possible d'utiliser la méthode `clone` (définie dans la classe `Object`) pour obtenir le même effet. Cependant il est généralement déconseillé en Java d'utiliser la méthode `clone` [2].

8 Préserver les invariants de classe

Considérez la classe `Rectangle` ci-dessous. Un rectangle est représenté par le coin inférieur gauche (en anglais *bottom left*), et le coin supérieur droit (*upper right*). Lors de la construction d'une nouvelle instance, le constructeur vérifie que le point inférieur gauche est bien en dessous et à gauche du second point. Cette propriété des attributs est un invariant de la classe.

1. On constate donc que c'est bien la référence `center` qui est inaccessible (`private`) en dehors de la classe et non l'objet de type `Point` référencé par cette variable. Il est impossible en dehors de la classe de modifier la référence (c-à-d l'instance 'pointée' par la variable) mais il n'est pas impossible de modifier l'objet, il suffit d'en posséder une référence en dehors de la classe.
2. http://www.javapractices.com/topic/TopicAction.do?Id=71

```java
package esi.atl.oo_base;

public class Rectangle {
   private Point bl; //bottom left corner
   private Point ur; //upper right corner

   public Rectangle(Point bottomLeft, Point upperRight) {
      if(bottomLeft.getX() >= upperRight.getX() || bottomLeft.getY() >=
       ↪ upperRight.getY())
          throw new IllegalArgumentException(
                "bottomLeft must be below and on the left of upperRight"
                +", received (bottomLeft - upperRight): "
                +bottomLeft+"-"+upperRight);
      this.bl = bottomLeft;
      this.ur = upperRight;
   }
   public void move(double dx, double dy) {
      bl.move(dx, dy);
      ur.move(dx, dy);
   }
   public double getPerimeter() {
      return 2*((ur.getX()-bl.getX())+(ur.getY()-bl.getY()));
   }
   @Override
   public String toString() {
      return "Rectangle : ["+bl+", "+ur+"]";
   }
}

class TestRectangle {
   public static void main(String args[]) {
      Point bl = new Point(0, 0);
      Point ur = new Point(5, 3);
      Rectangle r = new Rectangle(bl, ur);
      System.out.println(r);
      System.out.println("perimeter: "+r.getPerimeter());
      r.move(2, 5);
      System.out.println(r);
      System.out.println("perimeter: "+r.getPerimeter());
   }
}
```

Question 6

1. Qu'affiche ce programme ?
2. Donnez le diagramme d'objets (instances) de ce programme en indiquant les instances que les variables `bl`, `ur` et `r` référencent.
3. L'implémentation n'effectue pas de copie défensive des points passés en paramètres au constructeur. Ajoutez la ligne suivante à la ligne 36 de la méthode main :

   ```java
   bl.move(10,10);
   ```

   L'invariant est-il toujours respecté ?
   Qu'affiche maintenant le programme ?
   En particulier, que vaut le périmètre du rectangle ?
   Pourquoi cette valeur s'affiche-t-elle ?
4. Dans cet exemple, il est crucial de faire des copies défensives des paramètres reçus par le constructeur afin de garantir, à tout moment, l'intégrité des données. Dans le cas du cercle, même si l'encapsulation n'est pas totalement respectée, les données restent intègres puisque le centre du cercle n'a pas de contrainte particulière.
   Modifiez la classe `Rectangle` en lui ajoutant les copies défensives là où elles sont nécessaires. Qu'affiche maintenant le programme (modifié au point précédent) ?

9 Objets Immuables

Un objet est immuable (ou immutable) s'il est impossible de modifier ses attributs après sa construction. Les objets de la classe `String` ou `Integer` sont des exemples d'objets immuables parmi de nombreux autres.

Voici la version immuable de la classe `Point`.

```
1  package esi.atl.oo_base;
2
3  public final class ImmutablePoint { //declared final !
4
5     private final double x;
6     private final double y;
7
8     public ImmutablePoint() {
9        this(0,0);
10    }
11    public ImmutablePoint(double x, double y) {
12       this.x = x;
13       this.y = y;
14    }
15    public ImmutablePoint move(double dx, double dy) {
16       return new ImmutablePoint(x+dx, y+dy); //defensive copy !
17    }
18    @Override
19    public String toString() {
20       return "("+x+", "+y+")";
21    }
22 }
23
24 class TestImmutablePoint {
25    public static void main(String args[]) {
26       ImmutablePoint p = new ImmutablePoint();
27       System.out.println(p);
28       ImmutablePoint newP = p.move(2,2);
29       System.out.println(p);
30       System.out.println(newP);
31    }
32 }
```

Pour la rendre immuable nous avons :

- ▷ Déclaré la classe `final` (ligne 3) afin d'empêcher la création de toute sous-classe.
- ▷ Déclaré les attributs `final` afin de s'assurer qu'ils ne soient jamais modifiés après initialisation.
- ▷ Remplacé la méthode `move` par une version immuable : la méthode ne modifie pas les attributs mais elle retourne un nouveau `ImmutablePoint` correspondant au point déplacé. Notez que le nom de la méthode est légèrement trompeur : le point n'est pas déplacé (puisqu'il est immuable).

Si vous le souhaitez et pour vous entrainer, vous pouvez réaliser l'exercice supplémentaire suivant. Écrivez une classe `ImmutableCircle` possédant un attribut `centre` de type `Point`.

Théorie

Objets immuables

Pour rendre une classe immuable il faut :

- ▷ Déclarer la classe `final` pour s'assurer qu'elle ne puisse pas être sous-classée.
- ▷ Rendre tous les attributs `private` et `final` [a] pour s'assurer que leur valeur ne change jamais une fois initialisée.
- ▷ Ne fournir aucune méthode modifiant l'état (ni `setXXX` ni tout autre méthode modifiant les attributs).
- ▷ Faire des copies défensives de tous les attributs mutables (ce n'est pas nécessaire si l'attribut est immuable).

Les avantages des objets immuables sont nombreux, entre autres :

- ▷ ils n'ont pas besoin de constructeur de copie ;
- ▷ ils ne doivent pas être copiés défensivement ;
- ▷ ils font d'excellentes clefs pour les `HashMap` et éléments pour les ensembles (`Set`) car ces objets (clefs et éléments d'ensemble) ne peuvent jamais être modifiés lorsqu'ils sont dans la collection ;
- ▷ ils peuvent être utilisés sans craintes dans un environnement multi-tâches.

a. Notez que déclarer les attributs `final` n'est pas strictement nécessaire pour que la classe soit immuable.