LU2IN002 - Introduction à la programmation orientée-objet

Responsable de l'UE : Christophe Marsala
(email: Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(email: Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)

SCIENCES SORBONNE UNIVERSITÉ

Cours 5 – 10 octobre 2022

Plan du cours

Programme du jour

POO $\neq$ static

POO

- Un objet protège ses attributs
- Un objet possède des méthodes pour gérer ses attributs

Usage

1. Création d'un objet
2. Appel de méthodes sur cet objet

Static

- Les attributs/méthodes static ne dépendent pas d'un objet
- Tous les objets d'une classe ont accès aux mêmes informations static

Usage

1. Appel de méthode/attribut indépendamment des objets

Certains problèmes sont par nature des problèmes plutôt orientés objets, tandis que d'autre non

- Par exemple, pour générer un nombre aléatoire, inutile de créer un objet

Exemples de cas d'usage de static

- **Attribut static** : partage d'information entre objets de la classe
  - Constantes : *TAILLE_MAX*, π ...
  - Compteurs

    Combien d'instances de `Point` ont-elles été créées ?
    Question non triviale avec les outils actuels !
  - Liste des objets créés

    Je voudrais accéder à n'importe quel point créé jusqu'ici...
- **Méthodes static** : méthodes non liées à un objet
  - outils

    Calculer le cosinus est un problème qui ne dépend pas d'un objet
  - accesseur à un attribut static
  - méthode `main`
  - l'exemple du Singleton

Syntaxe : variable static

Deux sortes d'attributs :

- variable d'instance
- variable de classe (variable static)

```java
public class MaClasse {
   private int varI;
   private static int varC = 0;

   public MaClasse(int varI) {
      this.varI=varI;
      varC++;
   }
```

Bonne pratique

En général :

- les variables d'instance s'initialisent dans le constructeur
- les variables de classe s'initialisent lors de la déclaration

Remarque : comme les variables static sont déjà initialisées avant la création du premier objet, on peut les utiliser dans le constructeur

Syntaxe : méthode static

Deux sortes de méthodes :

- méthode d'instance
- méthode de classe (méthode static)

```java
public class MaClasse {
   private int varI;
   private static int varC = 0;

   public MaClasse(int varI) {
      this.varI=varI;
      varC++;
   }
   public static int methodeStatic(int a) {
      // instructions qui ne dépendent
      // pas d'un objet
      return a+varC;
   }
   public static int getVarC() {
      return varC;
   }
}
```

Bonne pratique

En général, l'accesseur d'une variable static doit être static, car il ne dépend pas d'un objet

Syntaxe : les constantes

```java
public class MaClasse {
   public static final int MA_CONSTANTE=10;
   ...
```

- `public` : si le client est autorisé à connaître la valeur de la constante, `private` sinon
- `static` : une constante ne dépend pas d'un objet
- `final` : une constante ne doit pas être modifiée

△ Ne pas confondre les mots clés `static` et `final`

```java
public class MaClasse {
   private static final int [] tab={11,12,13};
   public static void maMethode() {
      tab[0] = 15; // OK modification de la valeur d'une case
      tab = new int [4]; // Faux, la variable tab est final
   }
}
```

Pour les tableaux (et les objets), c'est la valeur de la **variable** qui ne peut pas changer, par contre, les valeurs des cases du tableau (et les objets) peuvent changer $\Longrightarrow$ déclarer la variable `private`

Syntaxe : classe "Outil"

Pour certains problèmes, on n'a pas besoin d'objets, mais d'une classe pour "stocker" des valeurs et faire des "calculs"

- Par exemple, la classe `Math` contient la variable `PI` pour "stocker" π et des méthodes pour calculer cos, sin, ...
- De même, on peut aussi écrire une classe qui contient seulement :
  - des attributs `static`
  - des méthodes `static`
  - un constructeur privé pour empêcher la création d'objets

```
1 public class MaClasseOutil {
2     public static final int MA_CONSTANTE=10;
3     private static int autreAttributStatic=15;
4     private MaClasseOutil() { }
5     public static void maMethode() { }
6 }
```

△ Comme le constructeur est privé, on ne peut pas créer d'objets **dans une autre classe**

```
// dans le main
MaClasseOutil mco = new MaClasseOutil(); // Erreur compilation
```


Syntaxe : utilisation dans une autre classe

```
1 public class MaClasseOutil {
2     public static final int MA_CONSTANTE=10;
3     private MaClasseOutil() { }
4     public static void maMethode() { }
5 }
```

★ Comment utiliser une variable static ou une méthode static dans une autre classe ? Par exemple dans le main ?

Variable static : NomClasse.nomVariableStatic

Exemples :

```
21 System.out.println(MaClasseOutil.MA_CONSTANTE);
22 double x = Math.PI;
23 System.out.println(x); // out : variable static
```

Appel de méthode : NomClasse.nomMethodeStatic(...)

Exemples :

```
24 MaClasseOutil.maMethode();
25 double y = Math.random();
26 String s = String.format("%.2f",y);
```

Remarque : aucune instanciation d'objet n'a été nécessaire pour utiliser ces variables et méthodes static

Syntaxe/philosophie comparative

Programmation objet :

```
1 // Instantiation
2 Point p = new Point(1,2);
3
4 // Invocation de méthode
5 // SUR L'INSTANCE
6 p.move(3, 3);
7 ...
```

Philosophie :

Les méthodes *accèdent / modifient* l'instance

Programmation static

```
1 // Pas d'instantiation de la classe
2 // Appel directement sur la classe
3 double pi = Math.PI;
4
5 // Pareil pour les méthodes
6 double d = Math.cos(pi);
```

Philosophie :

- Pas d'instance, pas d'accès aux variables d'instance
- Constante indépendante
- Méthode indépendante

⇒ **Essayons maintenant de mélanger les 2 philosophies pour faire des choses nouvelles**

Plan du cours

Cas classique : comptage d'instances

Combien d'instances de `Point` ont-elles été créées ?
Question non triviale avec les outils actuels !

Identifiant unique/comptage des instances

```
1 Point p1 = new Point(); // constructeur random
2 Point p2 = p1;
3 Point p3 = new Point(3,5);
4 ...
```

- Peut-on avoir un **compteur** qui compte le nombre d'objets `Point` créés ?
- Peut-on attribuer à chaque Point un **identifiant** unique lié à son ordre de création ?

Comptage d'instances : syntaxe standard

```
1 public class Point {
2   private static int cpt = 0;     // compteur d'instances
3   private final int id;           // identifiant d'une instance
4   private double x,y;
5   public Point(double x, double y){
6         this.x = x; this.y = y;
7         cpt++;
8         id = cpt;
9   }
```

- **Chaque `Point` a :**
  - un x, un y, un id
- **Tous les `Point` partagent :**
  - un compteur `cpt` défini au niveau de la classe

Remarque : cette classe contient la forme standard pour déclarer un compteur d'instances et l'utiliser pour donner un identifiant

1. déclaration d'une variable static initialisée à 0
2. incrémentation de la variable static dans le constructeur
3. utilisation de la variable static pour initialiser l'identifiant `id`

Comptage d'instances : syntaxe standard (2)

```
1  public class Point {
2    private static int cpt = 0;
3    private final int id;
4    private double x,y;
5
6    public Point(double x, double y){
7          this.x = x; this.y = y;
8          cpt++;
9          id = cpt;
10   }
11   // garantie de bonne gestion des id
12   public Point(){
13     this(Math.random()*10, Math.random()*10);
14   }
```

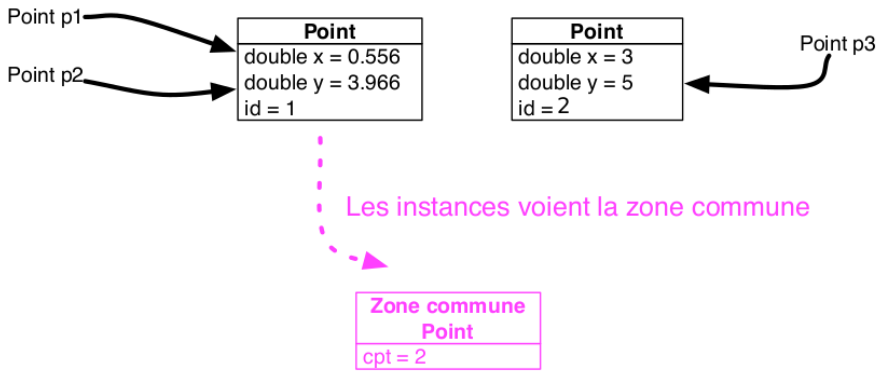
- Piège : attention aux constructeurs multiples
  - usage de `this(...)` très fortement conseillé pour passer toujours par le constructeur de référence et bien compter
  - après le `this(...)` de la ligne 13, ne pas remettre `cpt++`, car c'est déjà fait par le `this(...)`

Comptage & représentation mémoire

```
1  public class Point {
2    private static int cpt=0;
3    private final int id;
4    private double x,y;
5    ...
```

```
11 Point p1 = new Point();
12 Point p2 = p1;
13 Point p3 = new Point(3,5);
```

- Dans la représentation mémoire :
  - où se trouve l'`id` ? où se trouve le compteur `cpt` ?



△ Par contre, la zone commune ne connaît pas les instances

static / non static : asymétrie

- Les instances voient ce qui est static
- Les parties static ne voient pas les instances

```
1  public class Point{
2    private static int cpt = 0;
3    private final int id;
4    private double x,y;
5    ...
6    // Cas 1: OK méthode static, accès variable static
7    public static int getCpt(){return cpt;}
8    // Cas 2: OK méthode d'instance, accès variable static
9    public void methInst(){System.out.println("cpt="+cpt);}
10   // Cas 3 : KO méthode static, pas accès variable d'instance
11   public static void methStatic(){System.out.println("id="+id);}
```

△Les méthodes static ne peuvent pas utiliser de variables d'instance

Dans le main : `Point p1 = new Point();`

```
21 Point.getCpt(); // OK syntaxe naturelle appel méthode static
22 p1.getCpt(); // OK, mais à éviter

23 Point.methInst(); // Erreur compil : méthode d'instance nécessite un objet
24 p1.methInst(); //  OK syntaxe naturelle appel méthode d'instance
```


Liste des objets créés

★ Peut-on garder une liste des objets créés ?

```
1  import java.util.ArrayList;
2  public class Point {
3    private double x,y;
4    private static ArrayList<Point> alp=new ArrayList<Point>();
5
6    public Point(double x, double y){
7      this.x = x; this.y = y;
8      alp.add(this); // ajout du point créé dans la liste
9    }
10   public String toString() {
11     return "["+x+","+y+"]";
12   }
13   public static void afficherListePoints() {
14     for(Point p : alp) {
15       System.out.println(p);
16     }
17   }
18 }
```

```
31 // Main
32 new Point(3,4);
33 new Point(5,6);
34 Point.afficherListePoints();
```

Affiche les deux points :

```
[3.0,4.0]
[5.0,6.0]
```


L'exemple du singleton

★ Comment garantir qu'une classe ne puisse n'avoir qu'une seule instance ?

Approche du patron de conception Singleton :

```
1  public class Singleton {
2      private static final Singleton INSTANCE = new Singleton();
3
4      private Singleton() {}
5
6      public static Singleton getInstance() {
7          return INSTANCE;
8      }
9  }
```

1. une variable static pour stocker la seule instance
2. un seul constructeur private pour empêcher la création d'autres instances
3. une méthode static pour obtenir la référence vers cette unique instance

Remarque : il existe plusieurs variantes de ce patron

Un exemple d'utilisation du Singleton

- Une classe pour représenter l'origine du repère orthonormé : l'origine est un Point unique

```
1  public class Origine {
2      private static final Origine INSTANCE = new Origine(0,0);
3      private double x, y;
4
5      private Origine(double x, double y) {
6        this.x = x ; this.y = y ;
7      }
8
9      public static Origine getInstance() {return INSTANCE;}
10
11     public String toString() {
12         return "origine␣("+this.x+",␣"+this.y+")";
13     }
14
15     public double distanceAOrigine(Point p) {
16       return Math.sqrt( p.getX()*p.getX() + p.getY()*p.getY());
17     }
18 }
```

Un exemple d'utilisation du Singleton

- Une classe pour représenter l'origine d'un repère orthonormé

```
1 Point p1 = new Point(3, 2);
2 System.out.println(p1);
3
4 Origine orig = Origine.getInstance();
5 System.out.println(orig);
6
7 System.out.println( "Distance entre " + orig + " et "
8             + p1 + ":" + orig.distanceAOrigine(p1) );
```

- Résultat :

```
(3.0, 2.0)
origine (0.0, 0.0)
Distance entre origine (0.0, 0.0) et (3.0, 2.0): 3.605551275
```


Bilan...

★ Comment savoir si une variable est static ou pas ?

- Est-ce que la **variable dépend d'un objet** ?
  - ♦ Si oui, variable d'instance (VI)
  - ♦ Si non, variable de classe (static) (VC)

Exemple : On veut écrire une classe `Personne`

- age ? VI
  l'age d'une personne dépend de la personne
- `AGE_MAJORITE` ? VC
  l'age de la majorité ne dépend pas d'une personne en particulier
- `cptPersonnes` ? VC
  le nombre de personnes créées ne dépend pas d'une personne en particulier
- `cptEnfants` ? VI △ Tous les compteurs ne sont pas static
  le nombre d'enfants d'une personne dépend de la personne

Quand on vous parle de *static*, n'oubliez pas :

- Ce sont des cas très particuliers et assez rare
- N'oubliez pas les bonnes pratiques de la POO !!!!!

Plan du cours

Principes Orientés Objets

Principe 1 : Encapsulation

- Rapprochement données (attributs) et traitements (méthodes)
- Protection de l'information (private/public)

Principe 2 : Composition/Agrégation

- Un objet de la classe A est composé d'objets de la classe B
- Classe A AVOIR des Classe B

Principe 3 : Héritage

- Un objet de la classe B **est un** objet de la classe A aussi
- Classe B **ETRE** une Classe A

$\Longrightarrow$ La classe B **hérite** de la classe A

Idée de l'héritage

Spécialiser une classe, ajouter des fonctionnalités dans une classe
Hériter du comportement d'une classe existante

- **Une classe $\Longrightarrow$ plusieurs spécialisations possibles**
  - Animal $\rightarrow$ Vache, Chien, Panda...
  - hiérarchisation possible : Animal $\rightarrow$ Insecte $\rightarrow$ Papillon

Objectifs :

- **Ne pas avoir à modifier le code existant**
  - ne pas modifier la classe de base
  - Point $\rightarrow$ PointNomme : un point avec un nom
- **Ne pas avoir à faire de copier-coller !**
  - faire hériter le comportement d'une classe

Pour les cas suivants : dire si les relations sont des relations de type **Composition** ou **Héritage** :

- Salle de bains et baignoire
- Piano et pianiste
- Personne, enseignant et étudiant
- Animal, chien et labrador
- Cercle et ellipse
  - Un cercle est une ellipse particulière
  - OU Une ellipse est un cercle avec un attribut en plus
- Entier et réel
  - Un entier est un réel particulier
  - OU Un réel est un entier avec une partie décimale en plus

△ Pour un même problème plusieurs modélisations sont souvent possibles