LU2IN002 - Introduction à la programmation orientée-objet

Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ

Cours 5 – 13 octobre 2022

Plan du cours

POO ≠ static

POO

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

Usage

1. Création d'une instance
2. Appel de méthode sur cette instance

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 instances

- une instance est un individu
- les instances d'une classe partagent toutes ce qui est static

Usages de static

- **Attribut static** (aussi appelé : **variable de classe**) : usage : partager des informations entre les classes
  - Compteurs

Combien d'instances de Point ont-elles été créées ?
Question non triviale avec ce que l'on vu jusqu'ici !

  - Liste des objets créés

Comment accéder à n'importe quel Point créé jusqu'ici...

  - Constantes

PI, MAX_VALUE, POSITIVE_INFINITY,...

- **Méthodes static** : méthodes non liées à une instance
  - outils (opérations entre instances, opérations annexes)

Ex : cos, une méthode n'utilisant aucun attribut, **utilisable directement, sans instantiation d'un objet de la classe** `Math`

  - accesseur à un attribut static
  - méthode main

Syntaxe/philosophie comparative

Programmation objet

```
// Instantiation
Point p = new Point(1,2);

// Invocation de méthode
// SUR L'INSTANCE
p.move(3, 3);
p.toString();
...
```

Philosophie :

Les méthodes accèdent / modifient l'instance

Programmation static

```
// Pas d'instantiation de la classe
// Appel directement sur la classe
double pi = Math.PI;

// Pareil pour les méthodes
double d = Math.cos(pi);
```

Philosophie :

- Pas d'instance, pas d'accès aux attributs
- Constante indépendante
- Méthode indépendante de tout objet

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

Création de constantes

```
public class MaClasse{
  public static final int MA_CONSTANTE  = 42;
  ...
}
```

- public : accès autorisé pour le client
- static : une constante ne dépend pas d'un objet
- final : une constante ne doit pas être modifiable
- Utilisation (par exemple, dans un main) :

```
public class TestMaClasse{
  public static void main(String[] args){
    System.out.println("Valeur : "+ MaClasse.MA_CONSTANTE);
  }
}
```

- pas de création d'objet nécessaire !

Usage classique : comptage d'instances

Combien d'instances de Point ont-elles été créées ?
Question non triviale avec ce que l'on vu jusqu'ici !

Identifiant unique/comptage des instances

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

- Combien d'instances ?
- Peut-on attribuer à chaque Point un identifiant unique lié à son ordre de création ?

Comptage d'instances : syntaxe standard

Forme standard

```
1 public class Point{
2   private static int cpt = 0; // initialisation obligatoire ici
3   private final int id; // initialisation dans le constructeur
4                         // la variable ne doit pas être modifiable
5   private double x,y;
6
7   public Point(double x, double y){
8         this.x = x; this.y = y;
9         id = cpt++; // ou: id = cpt; cpt++;
10  }
```

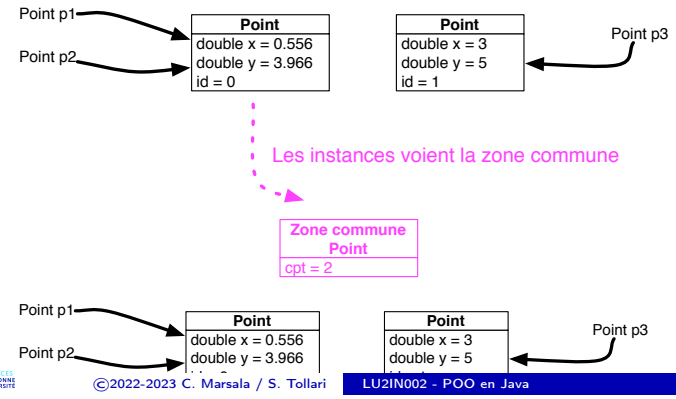
- **Chaque** `Point` **a :**
  - un x, un y, un id
- **Tous les** `Point` **partagent :**
- un compteur cpt défini au niveau de la classe

Le partage permet de raisonner sur des concepts qui dépassent UNE SEULE instance

Comptage & représentation mémoire

```
1 Point p1 = new Point();
2 Point p2 = p1;
3 Point p3 = new Point(3,5);
```

- Où se trouve l'`id` ?, Où se trouve le compteur ? (dans une représentation mémoire)



Comptage d'instances : syntaxe standard (2)

Forme standard

```
1 public class Point{
2   private static int cpt = 0; // initialisation obligatoire ici
3   private final int id; // initialisation dans le constructeur
4                         // la variable ne doit pas être modifiable
5   private double x,y;
6
7   public Point(double x, double y){
8         this.x = x; this.y = y;
9         id = cpt++; // ou: id = cpt; cpt++;
10  }
11
12  // garantie de bonne gestion des id
13  public Point(){
14    this(Math.random()*10, Math.random()*10);
15  }
```

- Piège : attention aux constructeurs multiples
  ⇒ utiliser this() : passer toujours par le constructeur de référence et bien compter.

Variables partagées

- Variable de classe/static = partage d'information

Exemples :

- Combien d'instances ont été créées jusqu'ici ? (compteur)
- Modélisation bancaire : une banque, plusieurs agences, toutes les agences voient/modifient l'ensemble des comptes...
- Maillage CAO : chaque triangle connaît tout le maillage (ensemble des instances créées) ⇒ interaction avec les voisins (alternative à une liste chainée)

Toujours vérifier qu'une variable static ne décrit pas une instance
⇒ sinon, on a fait une faute de conception

- le compteur d'instances est commun pour toutes les instances
  ⇒ static
- l'identifiant est spécifique à chaque instance
  ⇒ non static

Fonction static

- Boîte à outils (quelques exemples) :
  - génération de nom aléatoire (lettre aléatoire ou alternance voyelles/consonnes)
  - distance entre Points (formulation alternative à celle intra-classe),
    - possibilité de définitions multiples pour prendre en compte des contraintes
    - optimisation ultérieure
- L'exemple de la classe Math

L'exemple du singleton

- Idée : comment garantir qu'une classe ne puisse n'avoir qu'une unique instance ?
- Approche du Singleton :
  - bloquer l'accès au constructeur pour contrôler la création d'instance
  - méthode pour obtenir LA SEULE instance existante

```
public class Singleton {
    // variable de classe finale pour stocker une référence
    // et une seule
    private static final Singleton INSTANCE = new Singleton();

    // constructeur privé: interdiction de créer des objets
    // en dehors de la classe
    private Singleton() {}

    // méthode public pour récupérer la référence de l'unique
    // objet créé
    public static Singleton getInstance() {return INSTANCE;}
}
```


Un exemple d'utilisation

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

```
public class Origine {
    private static final Origine INSTANCE = new Origine(0,0);
    private double x, y;

    private Origine(double x, double y) {
      this.x = x ; this.y = y ;
    }

    public static Origine getInstance() {return INSTANCE;}

    public String toString() {
        return "origine (" +this.x+", "+this.y+")";
    }

    public double distanceAOrigine(Point p) {
      return Math.sqrt( p.getX()*p.getX() + p.getY()*p.getY());
    }
}
```


Un exemple d'utilisation

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

```
public class TestOrigine {
    public static void main(String[] args) {
        Point p1 = new Point(3, 2);
        System.out.println(p1);

        Origine orig = Origine.getInstance();
        System.out.println(orig);

        System.out.println( "Distance entre " + orig + " et "
                    + 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
```


static / non static : asymétrie

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

```
public class Point{
  private static int cpt = 0;
  private int id;
  private double x,y;
  ...
  // Cas 1: OK méthode static, accès variable static
  public static int getCpt(){return cpt;}
  // Cas 2: OK méthode d'instance, accès variable static
  public  int getCptInst(){return cpt;}
  // Cas 3 : KO méthode static, accès variable d'instance
  public static int getID(){return id;} // non sens!!
```

Depuis le main :

```
Point p1 = new Point();
// syntaxe  naturelle :
Point.getCpt();
// syntaxe possible (mais pas recommandée)
p1.getCpt();
// syntaxe impossible (évidemment) :
Point.getCptInst();
```


Bilan...

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

- Ce sont des cas très particuliers
- 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/Association

- Classe A est composé d'un objet de la Classe B
- Classe A utilise la Classe B

Principe 3 : Héritage

- Un objet de la classe B **est un** objet de la classe A aussi
- La classe B **hérite** de la classe A

Héritage

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
- **Objectif :** 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

Exemples & contre exemples

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

- Salle de bains et baignoire
- Piano et pianiste
- Personne, enseignant et étudiant
- Animal, chien et labrador
- Cercle et ellipse
- Entier et réel