SCIENCES SORBONNE UNIVERSITÉ

# LU2IN002
# STATIC

Vincent Guigue & Christophe Marsala

LIP6

MLIA Machine Learning & Deep Learning for Information Access

# Static ≠ POO

## 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

# Usage de static

- **Attributs static (=variable de classe):**

  partager des informations entre les instances d'une classe

  - 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...

  - Constantes (cas particulier, cf plus loin)

    $\pi$...

- **Méthodes static**:

  outils non reliés à une instance

  - Outils (opérations entre instances, opérations annexes)
  - Accesseur à un attribut static
  - Singleton

# Usage de static

- **Attributs static (=variable de classe):**
  partager des informations entre les instances d'une classe
  - Compteurs
  - Liste des objets créés
  - Constantes (cas particulier, cf plus loin)
- **Méthodes static**:
  outils non reliés 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
  - Singleton

## Programmation objet:

```
1 // Instantiation
2 Point p = new Point(1,2);
3
4 // Invocation de methode
5 // SUR L'INSTANCE
6 p.move(3, 3);
7 p.toString();
8 ...
```

### Philosophie:

Les méthodes *accèdent / modifient* l'instance

## Programmation objet:

```
1 // Instantiation
2 Point p = new Point(1,2);
3
4 // Invocation de methode
5 // SUR L'INSTANCE
6 p.move(3, 3);
7 p.toString();
8 ...
```

### 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 methodes
6 double d = Math.cos(pi);
```

### Philosophie:

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

## Programmation objet:

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

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

### Philosophie:

Les méthodes *accèdent / modifient* l'instance

## Programmation static:

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

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

### Philosophie:

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

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

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 ...
```

- Combien d'instance?
- Peut-on attribuer à chaque Point a un identifiant unique (lié à son ordre de création)?

## Forme standard

```
1 public class Point{
2   private static int cpt = 0; // initialisation obligatoire ici
3   private int id;  // initialisation interdite ici (-> constr)
4   private double x,y;
5
6   public Point(double x, double y){
7         this.x = x; this.y = y;
8         id = cpt++; // ou: id = cpt; cpt++;
9   }
```

- **Chaque `Point` a :**
  - un `x`, un `y`, un `id`
- **Tous les `Point` partagent:**
- un compteur `cpt`

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

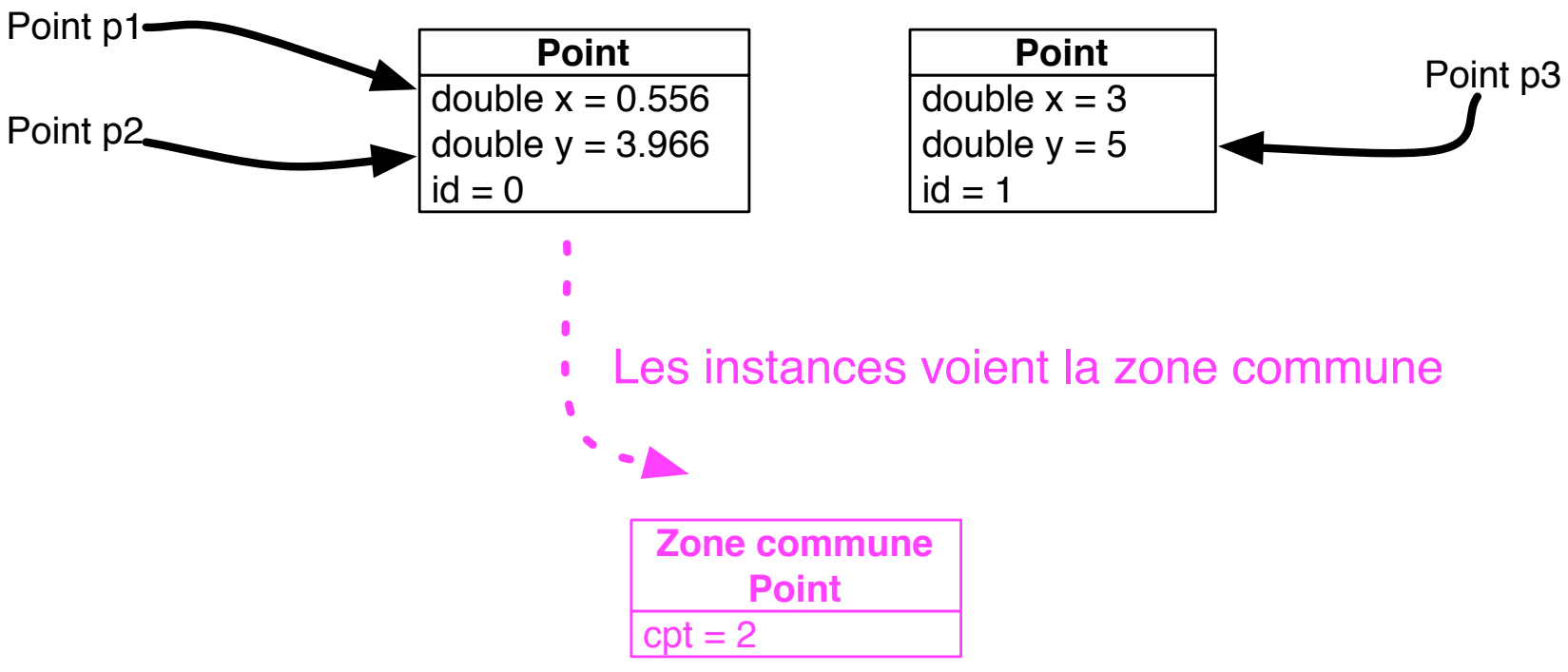
# Comptage & représentation mémoire

```
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)

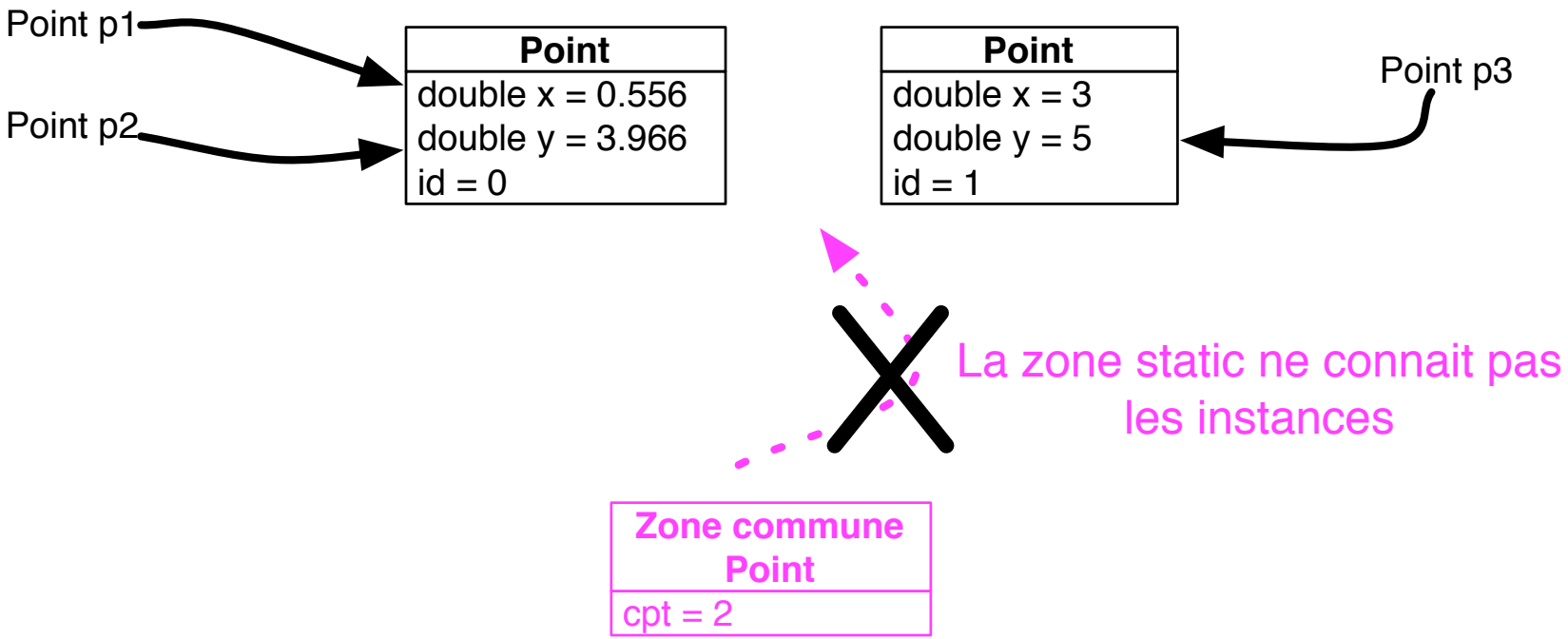
```
1 Point p1 = new Point();
2 Point p2 = p1;
3 Point p3 = new Point(3,5);
```

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



```
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)



## Forme standard

```
1  public class Point{
2    private static int cpt = 0; // initialisation obligatoire ici
3    private int id;  // initialisation interdite ici (-> constr)
4    private double x,y;
5
6    public Point(double x, double y){
7          this.x = x; this.y = y;
8          id = cpt++; // ou: id = cpt; cpt++;
9    }
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
  $\Rightarrow$ usage de `this()` très fortement conseillé pour passer toujours par le constructeur de référence et bien compter.

# Static ou non static : toujours vérifier

**Toujours vérifier** qu'une variable static **ne décrit pas** une instance
$\Rightarrow$ sinon, on a fait une faute de conception

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

Exercice du td: la classe `Chien`

| | |
|---|---|
| `nom` | `chercherLivreSurChiens()` |
| `aboyer()` | `vidéothèqueSurChiens` |
| `siteWebSurChiens` | `metsPrefere` |
| `siteWebDuChien` | `bibliographieSurChiens` |
| `siteWebSPA` | `dateNaissance` |
| `couleurDuPoil` | `manger()` |
| `courir()` | `regarderDVD()` |

# Autre exemple: conserver tous les points créés jusqu'ici

1. Je suis dans la classe `Point`

```
1 public class Point{
```

# Autre exemple: conserver tous les points créés jusqu'ici

1. Je suis dans la classe `Point`
2. Déclaration d'un tableau –taille variable– de `Point`

**Partagé entre toutes les instances de `Point`**

```
1 public class Point{
```

# Autre exemple: conserver tous les points créés jusqu'ici

1. Je suis dans la classe `Point`
2. Déclaration d'un tableau –taille variable– de `Point`
   **Partagé entre toutes les instances de `Point`**
3. static $\Rightarrow$ instantiation immédiate

```
1 public class Point{
2   private static ArrayList<Point> tab = new ArrayList<Point>();
```

# Autre exemple: conserver tous les points créés jusqu'ici

1. Je suis dans la classe `Point`
2. Déclaration d'un tableau –taille variable– de `Point`
   **Partagé entre toutes les instances de `Point`**
3. static $\Rightarrow$ instantiation immédiate
4. Dès qu'un `Point` est créé, on l'ajoute dans le tableau

```
1 public class Point{
2   private static ArrayList<Point> tab = new ArrayList<Point>();
3   public Point(double x, double y){
4         [...] // instructions diveres
5         tab.add(this); // ajout de l'instance courante dans le tab
6   }
```

$\Rightarrow$ Possibilités de recherche dans l'historiques etc...

# Fonction static

- Boite à outils:
  - 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

- Boite à outils:
  - 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

Génération aléatoire de lettre:

```
1 public class Alea {
2         public static char lettre() {
3                 return (char)(((char)(Math.random()*('z'-'a'+1)))+'a');
4         }
5 }
```

$\Rightarrow$ `lettre()` ne dépend d'aucun attribut

- Boite à outils:
  - 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

Génération aléatoire de lettre:

```
1 public class Alea {
2         public static char lettre() {
3                 return (char)(((char)(Math.random()*('z'-'a'+1)))+'a');
4         }
5 }
```

$\Rightarrow$ `lettre()` ne dépend d'aucun attribut

Usage:

```
1 // dans un main
2 char c = Alea.lettre(); // tres simple !
```

# Digression sur les classes outils

- Si une classe n'a pas vocation à être instanciée...
- Il faut interdire la possibilité de le faire !

Par défaut, on interdit tout... Et particulièrement ce que l'on n'est pas censé faire

$\Rightarrow$ mais comment interdire l'instanciation?

*Rappel:*

pas de constructeur = constructeur sans argument, public, qui ne fait rien.
= possibilité de créer une instance

- Si une classe n'a pas vocation à être instanciée...
- Il faut interdire la possibilité de le faire !

Par défaut, on interdit tout... Et particulièrement ce que l'on n'est pas censé faire

⇒ mais comment interdire l'instanciation?

*Rappel:*

pas de constructeur = constructeur sans argument, public, qui ne fait rien.
= possibilité de créer une instance

```
1 public class Alea {
2         private Alea(){
3                 // code vide, mais accolades obligatoires
4         }
5         public static char lettre() {
6                 [...]
7         }
8 }
```

# Un fonctionnement asymétrique

- 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 int id;
4   private double x,y;
5   ...
6   // Cas 1: OK methode static, acces variable static
7   public static int getCpt(){return cpt;}
8   // Cas 2: OK methode d'instance, acces variable static
9   public  int getCptInst(){return cpt;}
```

# Un fonctionnement asymétrique

- 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 int id;
4    private double x,y;
5    ...
6    // Cas 1: OK methode static, acces variable static
7    public static int getCpt(){return cpt;}
8    // Cas 2: OK methode d'instance, acces variable static
9    public  int getCptInst(){return cpt;}
10   // Cas 3 : KO methode static, acces variable d'instance
11   public static int getID(){return id;} // non sens !!
```

# Un fonctionnement asymétrique

- 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 int id;
4    private double x,y;
5    ...
6    // Cas 1: OK methode static, acces variable static
7    public static int getCpt(){return cpt;}
8    // Cas 2: OK methode d'instance, acces variable static
9    public  int getCptInst(){return cpt;}
10   // Cas 3 : KO methode static, acces variable d'instance
11   public static int getID(){return id;} // non sens !!
```

Depuis le main:

```
1  Point p1 = new Point();
2  // syntaxe  naturelle :
3  Point.getCpt();
4  // syntaxe possible (mais pas recommandée)
5  p1.getCpt();
6  // syntaxe impossible (évidemment) :
7  Point.getCptInst();
```

- Pas d'accès au constructeur
- Méthode pour récupérer LA SEULE instance existante

```java
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {return INSTANCE;}
}
```

## Cas d'usage : définition d'un nouveau type

- Classe `MonBooleen`, constructeur privé, deux attributs static
- Accesseur static `MonBooleen MonBooleen.getTrue()`, `MonBooleen MonBooleen.getFalse()`
- Dans le main, possibilité d'utiliser ==

**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!!!!!