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 2 – lundi 19 septembre 2022

Plan du cours

Programme du jour

Rappels et vocabulaire

```java
public class Point{
 private double x,y;
 public Point(double x2,double y2){
    x = x2;
    y = y2;
 }
 public double getX() {
   return x;
 }
 public void setX(double x2) {
   x=x2;
 }
 public String toString() {
   return "["+ x +","+ y +"]";
 }
}
```

classe $\simeq$ modèle pour créer des objets

Une classe est composée de :

- attributs
- constructeurs
- méthodes

Attributs

Il existe différents types d'attributs.
x et y sont **des attributs** qui sont aussi appellés variables d'instance

Rappels et vocabulaire

méthode $\simeq$ fonction définie dans une classe

Certaines méthodes sont un peu particulières :

- les accesseurs (*"getter"*) dont le seul but est d'accéder à la valeur d'un attribut.
  - Exemple : `double getX() { return x; }`
  - Par convention, la signature d'un accesseur est : `typeAttribut getNomAttribut()`
- les mutateurs (*"setter"*) dont le but est de modifier la valeur d'un attribut.
  - Exemple : `void setX(double x2) { x=x2; }`
  - Par convention, la signature d'un mutateur est : `void setNomAttribut(typeAttribut nomVar)`
- les **méthodes standards** qui sont déjà définies dans chaque objet, même si elles ne sont pas écrites dans la classe
  - Exemples : `toString`, `equals`...
- la méthode **main** : point d'entrée du programme

Rappels et vocabulaire

objet = instance d'une classe

```
21 Point p1 = new Point(1,2);
22 Point p2 = new Point(3,4);
```

```
1 public class Point{
2  private double x,y;
3  public Point(double x2, double y2){
4    x = x2; y = y2;
5  }
6  public Point add(Point p){
7    return new Point(x+p.x, y+p.y);
8  }
9 }
```

La variable `p1` référence un objet/instance de la classe `Point`.
La variable `p2` référence un **autre** objet/instance de la classe `Point`.

$\Longrightarrow$ Il y a 2 objets/instances de la classe `Point` créés

objet courant = l'objet avec lequel on a appelé la méthode

```
23 Point p3 = p1.add(p2);
```

A la ligne 7, l'objet courant est l'objet référencé par `p1`
(le paramètre `p` correspond à `p2`)

```
24 Point p4 = p2.add(p1);
```

A la ligne 7, l'objet courant est l'objet référencé par `p2`
(le paramètre `p` correspond à `p1`)

Variable d'instance, locale, paramètre

```
1  public class Point {
2    private double x,y;
3    public Point(double x2,double y2) {
4      x=x2;
5      y=y2;
6    }
7   public Point add(Point p2){
8     Point p3=new Point(x+p2.x, y+p2.y);
9     return p3;
10  }
11 }
```

Quelles sont les variables qui sont :

- des variables d'instance ?
  $\Longrightarrow$ x, y
- des paramètres ?
  $\Longrightarrow$ x2, y2, p2
- des variables locales ?
  $\Longrightarrow$ p3

Rappels : conventions d'écritures

- Le nom des **classes** et des **constructeurs** commence par une majuscule
  - Exemples : `MaClasse`, `MaClasse()`
- Le nom des **méthodes** et des **variables** (dont les attributs) commence par une minuscule
  - Exemples : `maMethode()`, `maVariable`
- Les **mots réservés** sont obligatoirement écrits tout en minuscules
  - Exemples : `public`, `class`, `true`, `false`...
- Les **constantes** sont généralement écrits tout en majuscules
  - Exemples : `Math.PI`, `MA_CONSTANTE`

$\Longrightarrow$ Rien qu'à la façon dont c'est écrit, vous pouvez savoir ce que c'est (une variable, une méthode, un constructeur...)

$\triangle$ Bien respecter ces conventions d'écritures quand vous écrivez un programme Java

Plan du cours

Surcharge du constructeur

Comment construire un `Point` ? ... de plusieurs manières !

Ex :
- 2 valeurs à fournir : le plus classique
- 1 valeur : affectation de la même valeur pour `x` et `y`
- 0 valeur : génération aléatoire de `x` et `y`

$\Longrightarrow$ il suffit de définir plusieurs constructeurs (= surcharge)

△ signatures toutes différentes

```
1  public class Point{
2    private double x,y;
3
4    public Point(double x2, double y2){
5      x = x2;
6      y = y2;
7    }
8    public Point(double d){ // surcharge
9      x = d;
10     y = d;
11   }
12   public Point(){ // autre surcharge
13     // aléatoire entre 0 et 10
14     x = Math.random()*10;
15     y = Math.random()*10;
16   }
17 }
```

Création de points en appelant le constructeur en fonction des besoins

```
31 Point p1 = new Point(2., 3.1);
```
⇒ appel constructeur à 2 paramètres

```
32 Point p2 = new Point(4.);
```
⇒ appel constructeur à 1 paramètre

⇒ x et y auront la même valeur

```
33 Point p3 = new Point();
```
⇒ appel constructeur sans paramètre

⇒ x et y aléatoires

Surcharge de constructeur

En général, que doit faire un constructeur ?
- ★ Initialiser les variables d'instance
- ★ Si besoin, faire d'autres initialisations

Un point a des coordonnées (x,y) et aussi un nom.

```
1  public class Point {
2    private double x,y;
3    private String nom;
4    public Point(double x2,double y2){
5      x=x2; y=y2;
6    }
7    public Point(double x2,double y2,
8                 String n){
9      x=x2; y=y2;
10     nom=n;
11   }
12   public String toString() {
13     return "Point␣"+nom
14            +"␣("+x+","+y+")";
15   }
16 }
```

Dans le main :

```
21 Point p=new Point(2.,3.);
22 System.out.println(p.toString());
```

Quel est l'affichage ?

`"Point null (2.0,3.0)"`

Quel est le problème ?

⊖ Dans le constructeur à 2 paramètres, la variable `nom` n'a pas été initialisée

Bonne pratique

En général, chaque constructeur doit initaliser chaque variable d'instance

```
4  public Point(double x2,double y2){
5      x=x2; y=y2;
6      // Nom aléatoire
7      nom="P"+(int)(Math.random()*1000+1);
8  }
```

Affiche : `"Point P198 (2.0,3.0)"`

Surcharge de méthode

Surcharge de méthode
- Plusieurs méthodes avec le même nom, mais paramètres différents
- Le type de retour ne compte pas

```
1  public class Point {
2    private double x,y;
3    ...
4    public void move(double dx,double dy){
5      x+=dx; y+=dy;
6    }
7    public void move(int dx,int dy){
8      x+=dx; y+=dy;
9    }
10   public void move(double dx,double dy,
11                    double scale){
12     x+=dx*scale; y+=dy*scale;
13   }
14   public void move(Point p){
15     x+=p.x; y+=p.y;
16   }
17 }
```

Pourquoi faire de la surcharge de méthode ?

★ Même nom de méthode pour faire la même chose

```
Point p=new Point(1.,2.);
p.move(a,b);
```

Que les variables a et b (supposées définies avant) soient de type `int` ou de type `double`, l'instruction `p.move(a,b)` est OK

Rappel : dans la classe, accès total aux attributs privés des autres instances de la **même classe** (=> ligne 14 : `p.x` et `p.y` OK)

Le mot clé this (1/4)

`this` : référence de l'objet courant dans une classe

- `this.maVariable` : accès à la variable d'instance de l'objet courant appelée `maVariable`
  - Exemples de syntaxe : `this.x, this.y`
- `this.nomMethode(...)` : appel de la méthode `nomMethode` (de **même signature**) de l'objet courant
  - Exemples de syntaxe : `this.toString(), this.add(p);`
- `this(...)` : appel au constructeur (de **même signature**) de l'objet courant
  - Exemples de syntaxe : `this(1,5); this(3); this();`

Quelques exemples d'utilisation dans les slides suivants.

Le mot clé this (2/4)

Pour rendre le code plus lisible, on voudrait que le paramètre qui sert à initialiser une variable d'instance porte le même nom que la variable d'instance.

```
1 public class Point {
2   private double x,y;
3   public Point(double x,double y) {
4     x = x;  // FAUX
5     y = y;  // FAUX
6   }
7 }
```

△ Faux, car le paramètre `x` cache la variable d'instance `x`

★ Solution : utiliser `this.x` pour préciser que c'est la variable d'instance `x`

```
1 public class Point {
2   private double x,y;
3   public Point(double x,double y) {
4     this.x = x;
5     this.y = y;
6   }
7 }
```


Le mot clé this (3/4)

- Comment éviter de répéter inutilement des instructions ?

```
1  public class Point {
2      private double x,y;
3      public Point(double x, double y){
4          this.x = x;
5          this.y = y;
6          -- initialisations --
7      }
8      public Point(double d){ // surcharge
9          x = d;
10         y = d;
11         -- initialisations --
12     }
13     public Point(){ // autre surcharge
14         x = Math.random()*10;
15         y = Math.random()*10;
16         -- initialisations --
17     }
```

Ici, on suppose que `"- initialisations -"` correspond à un long bloc d'instructions nécéssaires à l'initialisation de l'objet (indépendamment des paramètres du constructeur). Comment éviter de répéter ces **mêmes initialisations** dans chaque constructeur ?

Le mot clé this (4/4)

- Solution : utiliser un appel à un autre constructeur `this(...)`

```
1  public class Point {
2      private double x,y;
3
4      public Point(double x, double y){
5          this.x = x;
6          this.y = y;
7          -- initialisations --
8      }
9      public Point(double d){
10         this(d,d); // appel du constructeur Point(double, double)
11     }
12     public Point(){
13         this(Math.random()*10,Math.random()*10);
14     }
```

★ Le résultat est le même, mais les instructions d'initialisations ne sont plus répétées (code plus lisible, évite les bugs...)

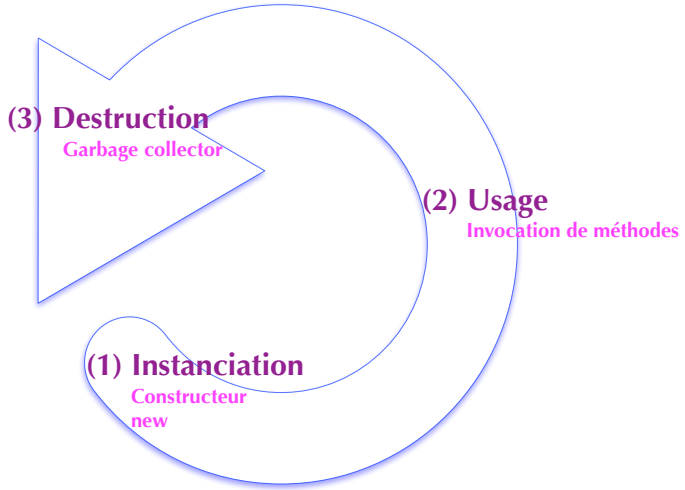
△ `this(...)` doit être la première instruction du constructeur

△ `this(...)` doit toujours être utilisé dans un constructeur, et jamais dans une méthode

Plan du cours

Cycle de vie : définition

Se placer du point de vue de l'objet :



(1) création d'une instance

(2) évolution / utilisation de l'instance

(3) condition de destruction

(1) Instanciation

Coté fournisseur :

mise en route de l'objet

Instanciation = constructeur = contrat d'initialisation des attributs

```
1 public class Point{
2   private double x,y;
3
4   public Point(double x2,double y2){
5     x = x2;
6     y = y2;
7   }
8 }
```

Coté client :

création d'une instance

Instanciation = création d'une zone mémoire réservée à l'objet

```
Point p1 = new Point(1., 2.);
```

Diagramme mémoire
(représentation des objets dans la mémoire)

(2) Usage

- le **fournisseur** développe et garantit le bon fonctionnement des méthodes pour *utiliser* l'objet correctement,
- le **client** invoque les méthodes sur des objets pour les manipuler.

```
1  public class Point{
2    private double x,y;
3    public Point(double x2,double y2){
4      x = x2;    y = y2;
5    }
6
7    public void move(double dx,
8                     double dy){
9      x += dx; y += dy;
10   }
11   ...
12 }
```

```
1 Point p1 = new Point(1., 2.);
2 p1.move(2., 3.); // p1 => [x=3, y=5]
```

Référence null

- Que se passe-t-il quand on déclare une variable (sans l'instancier) ?

```
1 Point p;
```

$\Longrightarrow$ p vaut `null`

- On peut écrire de manière équivalente

```
2 Point p = null;
```

- On ne peut pas invoquer de méthode

```
3 p.move(1., 2.); // => CRASH de l'exécution :
4                 // NullPointerException
```

- N'importe quel objet peut être `null` et réciproquement, on peut donner `null` à n'importe quel endroit où un objet est attendu... Même si ça provoque parfois des crashs.

```
1 // classe UnObjet,
2 // (classe sans importance)
3 ...
4 public void maFonction(Point p){
5   ...
6   p.move(1., 1.); // si p non null
7   ...
8 }
```

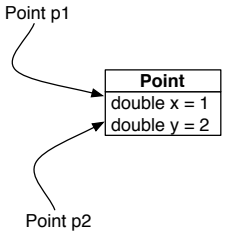
```
1 // dans le main
2 UnObjet obj = new UnObjet();
3
4 obj.maFonction(null);
5 // La méthode doit gérer !
```


(3) Destruction

1. Un objet est détruit lorsqu'il n'est plus référencé
2. La destruction est implicite (contrairement au C++) et traitée en tâche de fond (garbage collector)

- Un objet peut être référencé plusieurs fois...

```
1 Point p1 = new Point(1., 2.);
2 Point p2 = p1;
```



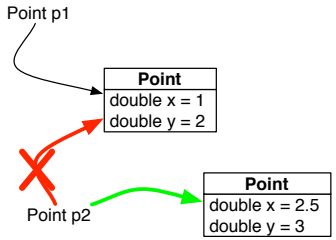
△ Il y a **un seul objet créé**, mais deux variables *p1* et *p2* qui référencent cet objet

- mais quand est-il dé-référencé ?

Dé-référencement d'un objet

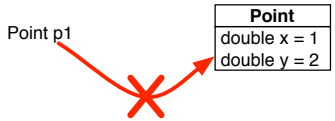
1. Dé-référencement explicite (usage de =)

```
1     Point p1 = new Point(1., 2.);
2     Point p2 = p1;
3     p2 = new Point(2.5, 3.);
```



2. Dé-référencement implicite (logique de bloc, destruction de variables $\Rightarrow$ destruction de références)

```
1 for (int i; i<10;i++) {
2       Point p1 = new Point(1., 2.);
3       System.out.println(p1);
4 }
5 System.out.println(p1);
6 // ERREUR DE COMPILATION
7 // p1 n'existe plus ici !
```



Retour sur la logique de bloc...

1. le dé-référencement dépend de l'endroit où la variable est **déclarée** (pas de l'endroit où la variable est **initialisée**)
2. ne pas confondre la destruction d'une **variable** et la destruction d'une **instance**

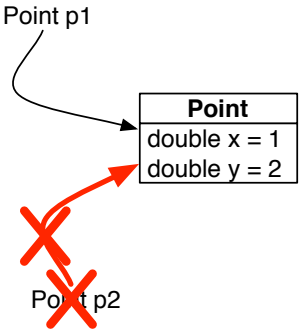
```
1 {
2   Point p1 = new Point(1.,2.);
3   System.out.println(p1);
4 } // destruction de
5   // la variable p1
6
7 System.out.println(p1);
8 // ERREUR DE COMPILATION
9 // p1 n'existe plus ici !
```

```
50 Point p1; // déclaration
51           // avant le bloc
52 {
53   // initialisation de p1
54   p1 = new Point(1.,2.);
55   System.out.println(p1);
56 } // pas de destruction de p1
57
58 System.out.println(p1);
59 // OK, pas de problème
```

Retour sur la logique de bloc (2)

1. le dé-référencement dépend de l'endroit où la variable est **déclarée** (pas de l'endroit où la variable est **initialisée**)
2. ne pas confondre la destruction d'une **variable** et la destruction d'une **instance**

```
1  Point p1; // déclaration
2            // avant le bloc
3  {
4    Point p2 = new Point(1.,2.);
5    // initialisation de p1
6    p1 = p2;
7    System.out.println(p1);
8  } // destruction de p2
9
10 System.out.println(p1);
11 // OK, pas de problème
```

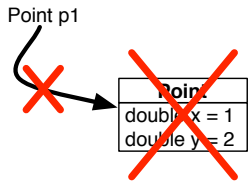


- Fin de bloc = destruction des **variables** déclarées dans le bloc
- Destruction d'instance ⇔ l'instance n'est plus référencée

Destruction des instances

Destruction d'instance ⇔ l'instance n'est plus référencée

```
1 Point p1 = new Point(1.,2.);
2 p1 = null;  // référence vers ''rien''
```



- Pas besoin d'expliquer comment détruire un objet (≠ C++)
- Le Garbage Collector planifie la destruction

```
1 for(int i=0; i<10; i++) {
2   // optimisation possible:
3   //      réutilisation de la mémoire allouée
4   Point p1 = new Point( Math.random()*10, Math.random()*10);
5 }
```

- Appel explicite au garbage collector (pour libérer la mémoire) :

```
1   System.gc();
```


Destruction des instances

... sur un exemple parlant :

```
1 Point p1 = new Point(1,2);
2 Point p2 = new Point(3,4);
3 // Point p3 = p1; // différence avec et sans cette ligne
4 p1 = p2;
```

- Cas 1 : ligne 3 commentée.
  - l'instance `Point(1,2)` **est détruite** à l'issue du re-référencement de l'objet référencé par `p1`...
  - ... de toutes façons, cette instance était devenue inaccessible.

```
1 Point p1 = new Point(1,2);
2 Point p2 = new Point(3,4);
3 Point p3 = p1; // différence avec et sans cette ligne
4 p1 = p2;
```

- Cas 2 : ligne 3 dé-commentée
  - l'instance `Point(1,2)` est **conservée**...
  - on y accède grâce à la variable `p3`

Plan du cours

Méthodes standards à redéfinir

- Des méthodes standards sont disponibles pour tous les objets (cf cours héritage)...
  - ♦ mais avec un comportement pas toujours satisfaisant
- Ex : conversion d'un objet en chaîne de caractères `public String toString()`

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3  ...
4  public String toString(){
5     // redéfinition
6     return "["+ x +","+ y +"]";
7  }
8 }
```

Client

```
20 Point p = new Point(2., 3.1);
21
22 String str = p.toString();
23
24 System.out.println("p:" + str);
```

```
» p: Point@8764152
```

```
» p: [2, 3.1]
```


Classes enveloppes

Les types de base en JAVA sont doublés de wrappers (ou classes enveloppes) pour :

- utiliser les classes génériques (cf cours `ArrayList`)
- fournir quelques outils fort utiles

`int, double, boolean, char, byte, short, long, float` ⇒ `Integer, Double, Boolean, Character`...

Outils : constantes et fonctions utiles

```
1 Double d1 = Double.MAX_VALUE; // valeur maximum possible
2 Double d2 = Double.POSITIVE_INFINITY; // valeur spécifique
3                                       // gérée dans les opérations
4 Double d3  = Double.valueOf("3.5"); // String => double
5 // Double.isNaN(double d), Double.isInfinite(double d)...
```

Documentation : `http://docs.oracle.com/javase/8/docs/api/java/lang/Double.html`

```
6 // conversions implicites = (un)boxing (depuis JAVA 5)
7 double d4 = d1;
8 Double d5 = d4;
```