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 3 – 26 septembre 2022

Méthode toString()

```
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 String toString() {
8      return "["+x+","+y+"]";
9    }
10 }
```

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

Quel est l'affichage ?
[2.0,3.0]

La méthode `toString()` est une méthode standard de Java, elle est appelée automatiquement dans certains cas.
Par exemple :

- quand on demande d'afficher un objet

```
23 System.out.println(p);          // Affiche : [2.0,3.0]
```

- quand on concatène une chaîne et un objet

```
24 System.out.println("p="+p);     // Affiche : p=[2.0,3.0]
```

⟹ Bien souvent, il est inutile d'écrire `p.toString()`, il suffit d'écrire `p`

Programme du jour

- Méthode `toString()`

1. Notion de composition
   - UML et composition/agrégation
2. Type de base *vs* Objet
3. Copie d'objet
   - Constructeur de copie
4. Égalité entre objets
   - Égalité référentielle
   - Égalité structurelle
5. Javadoc, débuggage : devenir autonome...
   - débugger son programme
   - se documenter et documenter soi-même

Plan du cours

Notion de composition

En POO, la composition ...

- est un type de **relation entre les classes**
- indique une relation de dépendance de type "AVOIR" entre les classes

Exemple :

- un segment **est composé de** deux points
  - ♦ reformulation : un segment a (**AVOIR**) deux points
  - $\Longrightarrow$ La classe `Segment` aura deux variables d'instance de type `Point`
- par contre, un point n'a pas de segments

Remarque : un type de relation similaire est l'agrégation. Dans l'UE LU2IN002, pour simplifier, on appellera "relation de composition", les relations de composition, mais aussi les relations d'agrégation.

Composition : syntaxe

```
1  public class Segment{
2    private Point a, b; // déclaration, pas d'objets créés
3
4    public Segment(Point a, Point b) {
5      this.a = a;
6      this.b = b;
7    }
8    public Segment() {
9      a=new Point();  // this(new Point(), new Point());
10     b=new Point();
11   }
12   public void move(double dx, double dy) {
13      a.move(dx, dy); // vision public du Point
14      b.move(dx, dy);
15   }
16   public String toString() {
17     // return "Segment [a="+a.toString()+", b="+b.toString()+"]";
18     return "Segment [a=" + a + ", b=" + b + "]";
19   }
20 }
21
```

★ En général, quand il y a composition, la méthode `toString()` de la classe qui agrège, appelle la méthode `toString()` des variables d'instance qui sont des objets

Composition : syntaxe

Un objet complexe = un objet qui utilise des objets

- Chaque classe reste petite, lisible et facile à débugguer
- Mais on peut construire des concepts complexes

```
31 Point p=new Point(1,2);
32 Segment s1=new Segment(p,new Point());
33 Segment s2=new Segment();
```

- Combien d'objets de la classe `Segment` ?
  - $\Longrightarrow$ 2 objets de la classe `Segment`
- Combien d'objets de la classe `Point` ?
  - $\Longrightarrow$ 4 objets de la classe `Point` (2 par objet `Segment`)
    - ♦ 1 objet à la ligne 31 (`new Point(1,2)`)
    - ♦ 1 objet à la ligne 32 (`new Point()`)
    - ♦ 2 objets dans le constructeur sans paramètre aux ligne 9 et 10 (slide précédent)

Rappels : UML (Unified Modeling Language)

UML est un **langage de modélisation** standard

Dans l'UE LU2IN002, le but de ce cours n'est pas d'apprendre UML, mais d'en utiliser une **version simplifiée** comme un outils pour modéliser :

- les relations entre les classes par un diagramme de classes
  - ♦ les **classes** sont représentées par un **rectangle composé de 3 parties** : 1/nom de la classe, 2/attributs, 3/méthodes
  - ♦ les relations entre les classes par des **lignes spéciales** entre les rectangles
- les objets dans la mémoire par un diagramme mémoire
  - ♦ les **objets** sont représentées par un **rectangle composé de 2 parties** : 1/type de l'objet, 2/attributs
  - ♦ les liens entre variables et objets par des **lignes fléchées**

△ Ne pas confondre diagramme de classe et diagramme mémoire

Remarque : parfois au lieu de **diagramme**, on dit **schéma** ou **représentation**. Exemples : dessiner le schéma des classes, représenter les objets dans la mémoire

UML : composition/agrégation

- La relation de composition/agrégation est représentée par une ligne avec un losange du côté de la classe qui agrège

```
1 public class Segment{
2   private Point a,b;
3   ...
```

Diagramme de classe détaillé

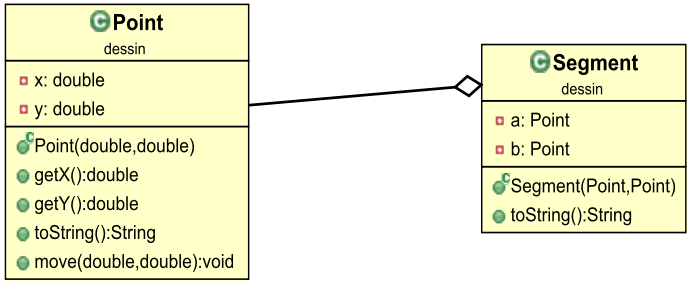
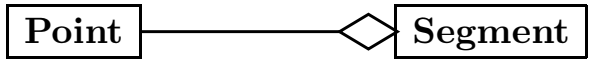


Diagramme de classe simplifié



Plan du cours

Problématique

- Le signe = se comporte de manière spécifique avec les objets...
- Le signe == également spécifique avec les objets...

Vocabulaire (uniquement pour les opérations sur objets)

new : instanciation / création d'instance
= : affectation de la valeur de la référence
== : égalité référentielle


```
1   Point p = new Point(1,2);
2   Point q = p;
3   Point r = new Point(1,2);
4   System.out.println( p == q ); // true
5   System.out.println( p == r ); // false
```


Types de base *vs* Objet : signification de =

Les types de base et les objets ne se comportent pas de la même façon avec l'affectation =

- Liste des types de base (cf. annexe du poly de TD) :
  `int, double, boolean, char, byte, short, long, float`

```
1 double a, b;
2 a = 1;
3 b = a; // affectation de la valeur 1
```

⇒ Si b est modifié, pas d'incidence sur a

- et pour un objet :

```
4 Point p = new Point(1,2);
5 Point q = p; // affectation de la valeur de la référence...
6              // mais toujours 1 seule instance !
```

2 variables mais 1 seule instance

Références et arguments de fonctions

- Passer un argument à une fonction revient à utiliser un signe =
- ... objets et types de base se comportent différemment !

```java
public class UnObjet{
  ...
  public void maFonction1(double d){
    ...
    d = 3.; // syntaxe correcte
            // mais très moche !
    ...
  }
  public void maFonction2(Point p){
    ...
    // L'objet est modifié
    p.move(1., 1.);
    ...
  }
}
```

```java
double c = 2.;
// c vaut 2
obj.maFonction1(c);
// c vaut toujours 2
```

$\Longrightarrow$ Le paramètre `d` de `maFonction1` est une variable locale

```java
UnObjet obj = new UnObjet();

Point q = new Point(1.,2.);
// q a pour attributs (x=1.,y=2.)
obj.maFonction2(q);
// q a pour attributs (x=2.,y=3.)
```

$\Longrightarrow$ p et q référencent le même objet

- Quand un type de base est passé en argument :
  - il y a **copie de la valeur** de la variable
- Quand un objet est passé en argument :
  - **il n'y a pas copie de l'objet**
  - mais il y a **copie de la valeur de la référence** vers l'objet

Plan du cours

Copie d'objets

★ Comment créer une copie d'un objet ?

Idée (assez raisonnable somme toute)

Créer un nouvel objet dont les valeurs des attributs sont identiques

- si l'attribut est de type de base, alors affectation de valeur
- si l'attribut est de type objet, alors il faut copier l'objet

Solutions possibles :

1. constructeur de copie
2. méthode standard `clone()` (étudié plus tard)

Copie d'objets : constructeur de copie

Solution 1 : constructeur de copie

Constructeur qui prend en **paramètre un objet de même type** et qui pour chaque attribut du paramètre duplique l'attribut et l'affecte à l'attribut correspondant de l'objet courant

- Si l'attribut est de **type de base**, il suffit d'une **affectation**
  - Exemple : constructeur de copie de la classe `Point`

```java
public Point(Point p) {
  x = p.x; // affectation
  y = p.y; // affectation
}
```

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

$\Longrightarrow$ il y a 2 objets `Point` avec les mêmes valeurs d'attributs

- Si l'attribut est un **objet**, il faut **copier l'objet**
  - Exemple : constructeur de copie de la classe `Segment`

```java
public Segment(Segment s) {
  a=new Point(s.a); //copie
  b=new Point(s.b); //copie
}
```

```java
Segment s1 = new Segment(p1,p2);
Segment s2 = new Segment(s1);
```

$\Longrightarrow$ il y a 2 `Segment` et 4 `Point`

Copie d'objets : méthode clone()

Solution 2 : méthode standard `clone()`

Méthode qui retourne un nouvel objet qui est une copie du point courant

△ Cette solution sera étudiée plus tard

- Exemple de code dans la classe `Point`

```
1 public class Point{
2 ...
3   public Point clone(){
4     return new Point(x, y);
5   }
6 }
```

- Usage :

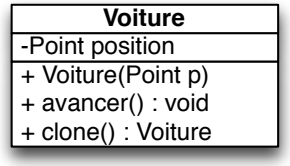
```
1 // main
2 Point p1 = new Point(1,2);
3 Point p2 = p1.clone();
```

- Comparaison constructeur de copie et méthode `clone()`
  - Résultat ABSOLUMENT identique
  - Cas d'utilisation un peu différent

Copie d'objets : LE PIEGE

★ Pourquoi il faut copier les attributs qui sont des objets ?

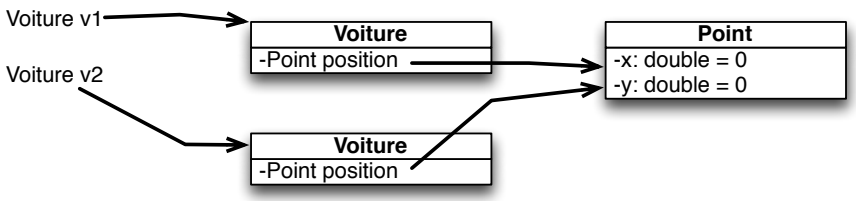
Exemple : besoin de copier une `Voiture` dont la position est définie par un attribut `Point`



Implémentation INCORRECTE :

```
1 // Dans voiture
2 public Voiture (Voiture v) {
3   position=v.position; // Faux
4 }
```

```
11 Point p=new Point(0,0);
12 Voiture v1 = new Voiture(p);
13 Voiture v2 = new Voiture(v1);
```

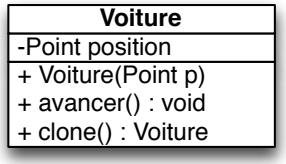


△ **PROBLEME :** il y a 2 objets `Voiture`, mais une seule position (un seul objet `Point`)... Si l'une bouge, l'autre aussi ! !

Copie d'objets : LE PIEGE

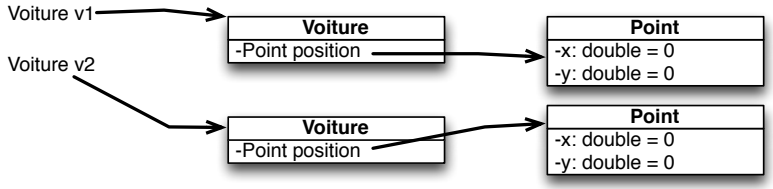
★ Pourquoi il faut copier les attributs qui sont des objets ?

Exemple : besoin de copier une `Voiture` dont la position est définie par un attribut `Point`



Implémentation correcte :

```
1 // Dans voiture
2 public Voiture (Voiture v) {
3   position=new Point(v.position);  // copie du point
4 }
```

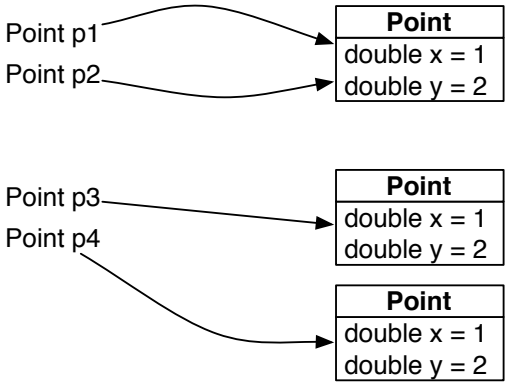


Plan du cours

Création de points *vs* Affectation

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

Point p3 = new Point(1, 2);
Point p4 = new Point(1, 2);
```



- Les variables p1 et p2 référencent la même instance
- p3 et p4 référencent des instances différentes

Types de base *vs* Objet : signification de ==

L'opérateur == : prend 2 opérandes de **même type** et retourne un `boolean`

- Type de base : égalité **des valeurs**
- Objet : égalité **des références** (égalité référentielle)
- ⚠ ATTENTION aux classes enveloppes (qui sont des objets)

```java
double d1 = 1.;
double d2 = 1.;
System.out.println(d1==d2); // affichage de true
                            //dans la console
Point p1 = new Point(1, 2);
Point p2 = p1;
System.out.println(p1==p2); // affichage de true

Point p3 = new Point(1, 2);
Point p4 = new Point(1, 2);
System.out.println(p3==p4); // affichage de false

Double d3 = 1.; // classe enveloppe Double = objet
Double d4 = 1.;
System.out.println(d3==d4); // affichage de false
System.out.println(d3.equals(d4)); // affichage de true
```


Comment tester l'égalité structurelle ?

Idée (toujours assez raisonnable)

Créer une méthode qui teste l'égalité des attributs

- **Solution 1** (simple mais pas utilisée)

```java
// Dans la classe Point
public boolean egalite(Point p){
  return p.x == x && p.y == y;
}
```

```java
// Dans le main

Point p1 = new Point(1.,2.);
Point p2 = p1;
Point p3 = new Point(1.,2.);
Point p4 = new Point(1.,3.);

p1.egalite(p2); // true
p1.egalite(p3); // true
p1.egalite(p4); // false
```

- `public boolean egalite(Point p)` produit le résultat attendu
- ⚠ ATTENTION à la signature :
  - la méthode retourne un booléen
  - la méthode ne prend qu'un argument (on teste l'égalité entre l'instance qui invoque la méthode et l'argument)

Comment tester l'égalité structurelle ?

Quand il y a de la composition, il faut appeler la méthode qui compare les objets.

```java
// Dans la classe Segment
public boolean egalite(Segment s){
  return a.egalite(s.a) && b.egalite(s.b);
}
```

Méthode standard : boolean equals(Object o)

Solution 2 : méthode standard `equals` (un peu plus complexe)

△ Cette solution sera étudiée plus tard

- `equals` existe dans tous les objets (comme `toString`)
  - par défaut : test de l'égalité référentielle...
    ⟶ pas intéressant (comme `toString` en version de base)
- ⇒ Redéfinition : faire en sorte de tester les attributs
  Un processus en plusieurs étapes :
  1. Vérifier s'il y a égalité référentielle et référence null
  2. Vérifier le type de l'`Object` o (cf cours polymorphisme)
  3. Convertir l'`Object` o dans le type de la classe (idem)
  4. Vérifier l'égalité entre attributs

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null) return false;
4      if (getClass() != obj.getClass())
5          return false;
6      Point other = (Point) obj;
7      if (x != other.x) || (y != other.y)
8          return false;
9      return true;
10 }
```


Egalité structurelle : attention au equals

△ Cette solution sera étudiée plus tard

- Structure standard classique...
- jusqu'au moment du test sur les attributs :
  ⟶ penser au `equals` (au lieu de ==)

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null) return false;
4      if (getClass() != obj.getClass())
5          return false;
6      Voiture other = (Voiture) obj; // pour accéder aux attributs
7      if (!position.equals(other.position))
8          return false;
9      return true;
10 }
```


Plan du cours

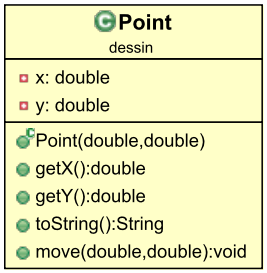
Les bons reflexes...

1. Lire les messages d'erreur dans la console
2. Savoir corriger les erreurs les plus courantes
3. Savoir chercher dans la documentation officielle JAVA...
4. ... Et éventuellement documenter votre propre code

Compilateur, JVM et garbage collector

- **Compilateur**
  - **syntaxe** (;, parenthèses, ...)
  - vérifie le **type des variables**,
  - l'existence des méthodes/attributs et les niveaux d'accès :
    - les méthodes/attributs existent-elles dans l'objet,
    - les accès sont-ils permis (**public**/**private**)
- **JVM**
  - gestion dynamique des liens (cf redéfinition avec l'héritage)
  - gestion des erreurs d'utilisation des objets
    - problème d'instanciation,
    - dépassement dans les tableaux,
    - gestion des fichiers...
  - garbage collector (cf cycle de vie des objets)

Erreurs usuelles à corriger soit même



Compilation (les plus faciles !) :

Toujours bien regarder la ligne de l'erreur (elle est donnée). Trouver le raccourci de votre éditeur permettant d'aller à la ligne fautive

Syntaxe

```
1 Point p = new Point(1,2)
2 p.move(1, 0);
3 // Syntax error, insert ";" to complete BlockStatements
```

Niveau d'accès

```
1 Point p = new Point(1,2);
2 p.x = 3;
3 // The field Point.x is not visible
```

Existence des méthodes

```
1 Point p = new Point(1,2);
2 p.mover(1,3);
3 // The method mover(int, int) is undefined for the type Point
```


Erreurs usuelles à corriger soit même

Execution (JVM) : Toujours vérifier la ligne également

- NullPointerException

```
1 Point p = null;
2 p.move(1, 0);
3 // Exception in thread "main" java.lang.NullPointerException
4 //     at cours1.TestPoint.main(TestPoint.java:2)
```

  - Cette erreur arrive souvent dans des cas plus complexe de composition d'objet

- IndexOutOfBoundException

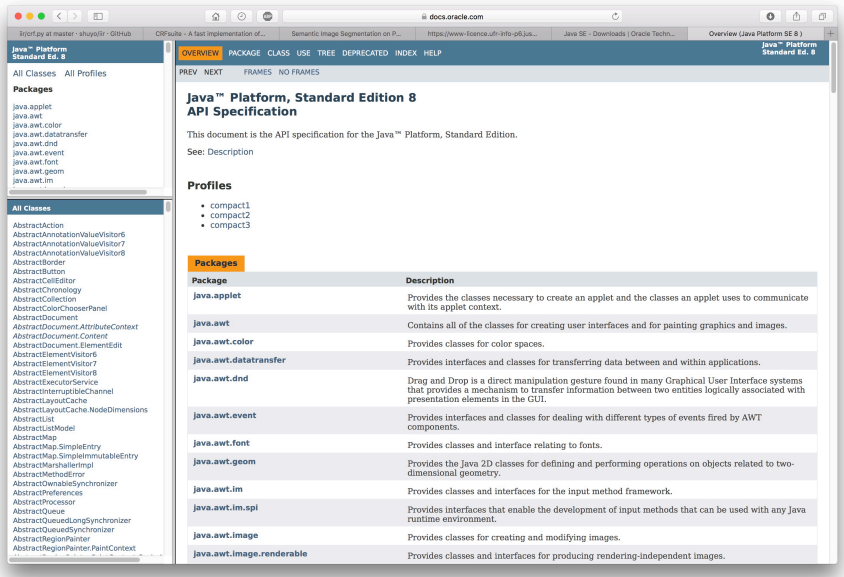
```
1 int[] tab = new int[3];
2 tab[3] = 2;
3 // Exception in thread "main"
4 //       java.lang.ArrayIndexOutOfBoundsException: 3
5 //     at cours1.TestPoint.main(TestPoint.java:2)
```

  - Vérifier la ligne et l'index !
  - Souvent dans les boucles for

Documentation

Java est un langage très bien documenté et plein d'outils :

`https://docs.oracle.com/javase/8/docs/api/index.html`



Documenter soi-même

De manière générale, on programme pour les autres...
⇒ documenter son code pour le rendre utilisable

1. premier niveau : choisir des noms de classes, méthodes et variables **explicites**.
2. deuxième niveau : faire des classes et des méthodes courtes, utiliser des méthodes privées...
3. troisième niveau : ajouter des commentaires pour créer une documentation.
   - ♦ outil intégré dans JAVA : commentaires spéciaux + création automatique d'une page web

Création d'une documentation

```java
/**
 * @author Vincent Guigue
 * Cette classe permet de gérer des points en 2D
 */
public class Point} {
  /**
   * Attributs correspondant aux coordonnées du point
   */
 private double x, y;
 /**
   * Constructeur standard à partir de 2 réels
   * @param x : abscisse du point
   * @param y : coordonnée du point
   */
    public Point(double x, double y) {
          this.x = x;
        this.y = y;
    }
    /**
     * @return l'abscisse du point
     */
    public double getX() {
        return x;
    }
}

```

```
$ javadoc Point.java
```


Javadoc : quelques options utiles

- De manière générale : vérifier la documentation

```
$ javadoc -h
```

- Pour gérer les accents :

```
$ javadoc -encoding utf8 -docencoding utf8 -charset utf8 [fichier.java]
```

- Pour sélectionner le répertoire de stockage du html :

```
$ javadoc -d <directory> [fichier.java]
```

- Représentation `public/private`
  (par défaut, représentation de la partie `public` seulement)

```
$ javadoc -public/-private [fichier.java]
```


Javadoc : résultats obtenus

- Classe Point, présentation conforme à la javadoc standard (présence des liens hypertextes...)