LU2IN002 - Introduction à la programmation orientée-objet

Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ

Cours 3 – 30 septembre 2022

Programme du jour

1. Rappel
2. Destruction d'objets
3. Classes enveloppes
4. Objets composés, composition d'objets
5. Egalité entre objets, Clonage d'objet
6. Javadoc, débuggage : devenir autonome...

Plan du cours

1. Rappel
2. Destruction d'objets
3. Classes enveloppes
4. Objets composés, composition d'objets
5. Egalité entre objets, Clonage d'objet
6. Javadoc, débuggage : devenir autonome...

Rappel sur la Programmation Orientée Objet (POO)

Un programme OO est constitué d'objets qui communiquent par envois de messages

- **objet** = instance d'une classe
- **communiquer avec un objet** par l'envoi d'un message = appel d'une méthode que connaît l'objet
- **méthode** = fonction définie dans une classe
- **objet courant** = l'objet qui est en train de répondre à un appel en exécutant le code d'une de ses méthodes

Rappel sur la Programmation Orientée Objet (POO)

Toute action doit être pensée selon un objet

Par exemple : additionner 2 nombres complexes c1 et c2

- en maths : c1+c2 rend un résultat qui est un complexe
  - l'opérateur + définit l'opération à réaliser entre 2 complexes
- en POO : **déséquilibre entre l'objet qui exécute l'addition et l'objet qui est un argument de cette addition**
  - on demande à c1 de rendre le complexe qui est le résultat de son **addition avec** c2
  - Instruction : `c1.addition(c2)` qui rend un nouveau complexe

```
1 public class Complexe { // voir exercice TME 2
2   ...
3   public Complexe addition(Complexe c2){
4     return new Complexe(this.reelle+c2.reelle, this.imag+c2.imag);
5   }
6
7 }
```


Plan du cours

1. Rappel
2. Destruction d'objets
3. Classes enveloppes
4. Objets composés, composition d'objets
5. Egalité entre objets, Clonage d'objet
6. Javadoc, débuggage : devenir autonome...

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**

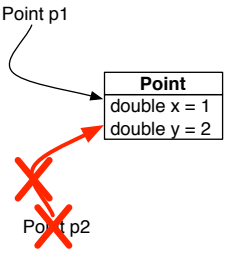
```
public static
    void main(String[] args) {

  {
    Point p1 = new Point(1.,2.);
    System.out.println(p1);
  }
  // sortie du bloc:
  // destruction de
  // la variable p1

  System.out.println(p1);
  // ERREUR DE COMPILATION
  // p1 n'existe plus ici !
}
```

```
public static
  void main(String[] args) {

  Point p1; // déclaration
            // avant le bloc
  {
    // initialisation de p1
    p1 = new Point(1.,2.);
    System.out.println(p1);
  } // pas de destruction de p1


  System.out.println(p1);
  // 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

```
public static
    void main(String[] args) {
  Point p1; // déclaration
            // avant le bloc
  {
    Point p2 = new Point(1.,2.);
    // initialisation de p1
    p1 = p2;
    System.out.println(p1);
  } // destruction de p2

  System.out.println(p1);
  // OK, pas de problème
}
```

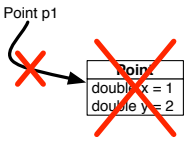


- Fin de bloc = destruction des **variables** déclarées dans le bloc
- Destruction d'instance ⇔ l'instance n'est plus référencée

Destruction des instances

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

```
public static void main(String[] args) {
  Point p1 = new Point(1.,2.);
  p1 = null;  // référence vers ''rien''
}
```



- Pas besoin de dire comment détruire un objet
- Mécanisme interne à la JVM : le Garbage Collector

```
public static void main(String[] args) {
...
  for(int i=0; i<10; i++){
    // optimisation possible:
    //      réutilisation de la mémoire allouée
    Point p1 = new Point( Math.random()*10, Math.random()*10);
    ...
  }
}
```

- Possibilité de faire un appel explicite au garbage collector :

```
System.gc();  // fait le ménage dans la mémoire
```

  - mais c'est très rarement (jamais) utilisé dans un programme

Le mot de la fin...

... sur un exemple parlant :

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

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

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

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 très utiles

types de base : `int, double, boolean, char, byte, short, long, float`

→ classes enveloppes : `Integer, Double...`

Outils : constantes et fonctions utiles

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

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

```
// conversions implicites = boxing / unboxing
double d4 = d1;       // unboxing: d1 converti en double
Double d5 = d4;       // boxing: double stocké dans un objet
```

Plan du cours

Philosophie & syntaxe

Un objet composé = un objet qui utilise d'autres objets

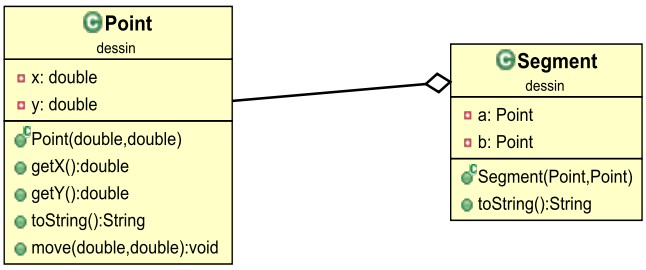
- Chaque classe reste petite, lisible et facile à débugguer
- Par agrégation, on construit des concepts complexes

```java
public class Segment{
  private Point a, b; // déclaration des variables d'instance
  public Segment(Point a, Point b) {
    this.a = a;
    this.b = b;
  }
  public Segment() {
    this.a = new Point(); // cf. la classe Point (cours 2)
    this.b = new Point();
  }
  public String toString() {
    return "Segment [a=" + a + ", b=" + b + "]";
  }
  public void move(double dx, double dy) {
     a.move(dx, dy); // méthode publique de Point
     b.move(dx, dy);
  }
}
```


Représentation des liens UML

```java
public class Segment{
  private Point a,b;
  ...
```

Représentation UML :
Lien d'agrégation : Un segment est composé de Point(s)



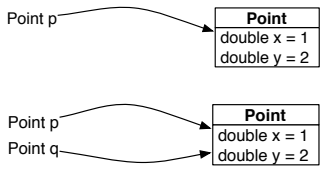
Plan du cours

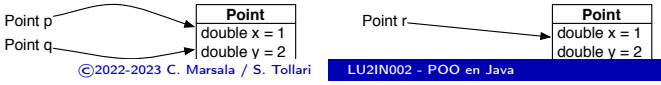
Problématique

- 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
= : duplication de référence
== : égalité référentielle





Types de base *vs* Objet : signification de =

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

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

```java
double a, b;
a = 1;
b = a; // duplication de la valeur 1
```

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

- et pour un Objet :

```java
Point p = new Point(1,2);
Point q = p; // duplication de la référence...
             // 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(Point p){
  ...
  p.move(1., 1.); // ajout à x et y
  ...
}

public void maFonction2(double d){
  ...
  d = 3.; // syntaxe correcte
          // mais très moche !
  ...
}
```

```java
// dans le main
UnObjet obj = new UnObjet();

Point q = new Point(1.,2.);
double d = 2.;

obj.maFonction1(q);
obj.maFonction2(d);

// q a pour attributs (x=2.,y=3.)
// d vaut 2
```

- Quand un type de base est passé en argument : duplication.
- Quand un objet est passé en argument :
  il n'y a pas duplication de l'instance (simplement une copie de la référence vers l'objet)

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

- Opérateur == : prend 2 opérandes de **même type** et retourne un `boolean`
- Types de base : vérification de l'égalité **des valeurs**
- Objet : vérification de l'égalité **des références**
- ⚠ 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
```


Comment tester l'égalité structurelle ?

Idée (toujours assez raisonnable)

Créer une méthode qui teste l'égalité des attributs

- **Solution 1** (simple)

```java
// Dans la classe Point
public boolean egalite(Point p){
  if (p == null)
      return false;
  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)

- **Solution 2 (vue plus tard)** : equals()
  (standard... mais un peu plus complexe)

Copie d'objets : constructeur de copie

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

- Exemple de constructeur par copie dans la classe `Point`

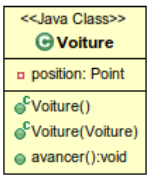
```java
// Constructeur de Point a partir d'un autre Point
public Point(Point p){
  this.x = p.x; // utiliser this ici est facultatif
  this.y = p.y; // utiliser this ici est facultatif
}
```

- Usage :

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


Copie d'objet composé : le piege

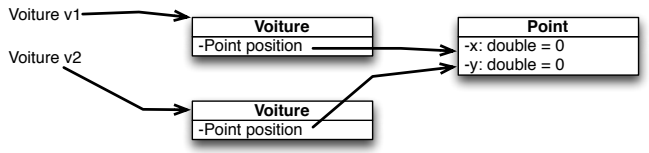
Cas classique : besoin de dupliquer une `Voiture` dont la position est définie par un attribut `Point`



Implémentation INCORRECTE :

```java
// Dans voiture
public Voiture(Voiture v){
  position= v.position
}
// dans le main
Voiture v1 = new Voiture(new Point(0,0));
Voiture v2 = new Voiture(v1);
```

⚠ GROS PROBLEME ! !



Il y a 2 instances de Voiture, mais 1 seule position...
Si une voiture bouge, l'autre aussi ! !
Implémentation correcte :

Copie d'objets : le clonage

méthode standard clone()

Méthode qui retourne un nouvel objet qui est une copie de l'objet courant.

On reviendra sur cette approche plus tard

- Exemple de code dans la classe `Point`

```java
public class Point{
...
  public Point clone(){
    return new Point(x, y);
  }
}
```

- Usage :

```java
// main
Point p = new Point(1,2);
Point p2 = p.clone();
```

NB : construction de nouvelle instance sans `new` écrit dans le `main`

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

Compilation et exécution

Exemple d'instructions de compilation/exécution :

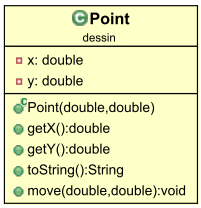
```
1 javac Point.java
2 javac MainPoint.java
3 java MainPoint
```

- Les deux premières instructions concernent la compilation... Même s'il n'y a pas de message d'erreur, il faut encore vérifier le bon fonctionnement du programme !
- La troisième ligne exécute le code compilé

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

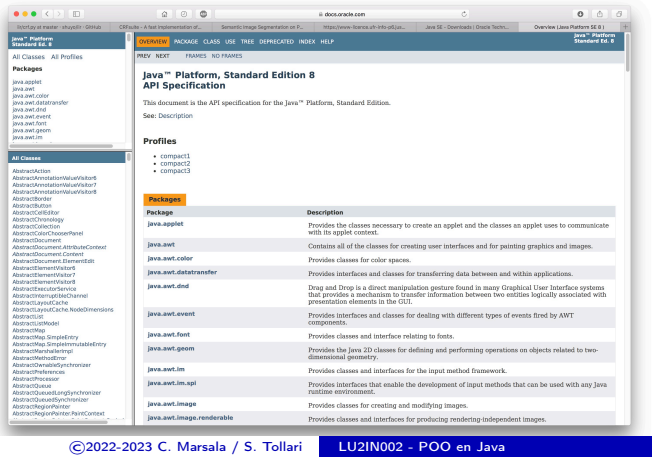
```
1 int[] tab = new int[3];
2 tab[3] = 2;
3 // Exception in thread "main"
4 //        java.lang.ArrayIndexOutOfBoundsException: 3
5 //      at cours1.TestPoint.main(TestPoint.java:2)
```

  - 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...)