

LU2IN002 - Introduction à la programmation objet

Christophe Marsala



Cours 3 – 21 septembre 2024

PLAN DU COURS

- 1 Rappels
- 2 Destruction d'objets (suite)
- 3 Représentation UML
- 4 Objets composés, composition d'objets
- 5 Classes enveloppes
- 6 Egalité entre objets, clonage
- 7 Javadoc, débogage : devenir autonome...

RAPPELS SUR LA PROGRAMMATION OBJET

Toute action doit être **pensée selon un objet**

Par exemple : additionner 2 nombres complexes c1 et c2

- o en maths : $c1+c2$ rend un résultat qui est un complexe
 - l'opérateur + définit l'opération à réaliser entre 2 complexes
- o 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 }
```

PROGRAMME DU JOUR

- 1 Rappels
- 2 Destruction d'objets (suite)
- 3 Représentation UML
- 4 Objets composés, composition d'objets
- 5 Classes enveloppes
- 6 Egalité entre objets, clonage
- 7 Javadoc, débogage : devenir autonome...

RAPPELS SUR LA PROGRAMMATION OBJET

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

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



©2024-2025 C. Marsala / S. Tollari

LU2IN002 - Programmation objet en Java

4/41

PLAN DU COURS

- 1 Rappels
- 2 Destruction d'objets (suite)
- 3 Représentation UML
- 4 Objets composés, composition d'objets
- 5 Classes enveloppes
- 6 Egalité entre objets, clonage
- 7 Javadoc, débogage : devenir autonome...



©2024-2025 C. Marsala / S. Tollari

LU2IN002 - Programmation objet en Java

5/41

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**

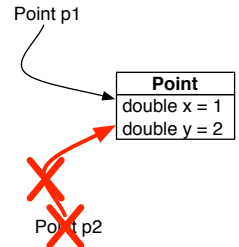
```
1 public static
2 void main(String[] args) {
3
4 {
5     Point p1 = new Point(1.,2.);
6     System.out.println(p1);
7 }
8 // sortie du bloc:
9 // destruction de
10 // la variable p1
11
12 System.out.println(p1);
13 // ERREUR DE COMPILATION
14 // p1 n'existe plus ici !
15 }
```

```
1 public static
2 void main(String[] args) {
3
4     Point p1; // déclaration
5             // avant le bloc
6 {
7     // initialisation de p1
8     p1 = new Point(1.,2.);
9     System.out.println(p1);
10 } // pas de destruction de p1
11
12 System.out.println(p1);
13 // OK, pas de problème
14
15 }
```

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 public static
2 void main(String[] args) {
3     Point p1; // déclaration
4             // avant le bloc
5 {
6     Point p2 = new Point(1.,2.);
7     // initialisation de p1
8     p1 = p2;
9     System.out.println(p1);
10 } // destruction de p2
11
12 System.out.println(p1);
13 // OK, pas de problème
14 }
```

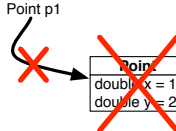


- o Fin de bloc = destruction des **variables** déclarées dans le bloc
- o Destruction d'instance \Leftrightarrow l'instance n'est plus référencée

DESTRUCTION DES OBJETS

Destruction d'objet \Leftrightarrow l'objet n'est plus référencé

```
1 public static void main(String[] args) {
2     Point p1 = new Point(1.,2.);
3     p1 = null; // référence vers 'rien'
4 }
```



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

```
1 public static void main(String[] args) {
2     ...
3     for(int i=0; i<10; i++){
4         // optimisation possible:
5         // réutilisation de la mémoire allouée
6         Point p1 = new Point(Math.random()*10, Math.random()*10);
7         ...
8     }
9 }
```

- o Possibilité de faire un appel explicite au garbage collector :
1 `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 :

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

- o Cas 1 : ligne 3 commentée.
 - l'objet Point(1,2) **est détruit** à l'issue du changement de référencement de p...
 - ... \rightarrow cet objet était devenu inaccessible.

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

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

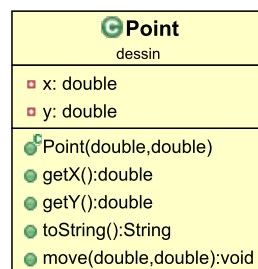
PLAN DU COURS

- 1 Rappels
- 2 Destruction d'objets (suite)
- 3 Représentation UML
- 4 Objets composés, composition d'objets
- 5 Classes enveloppes
- 6 Egalité entre objets, clonage
- 7 Javadoc, débogage : devenir autonome...

REPRÉSENTATION UML

On ne programme pas pour soi-même... mais pour les autres :

- o Respecter les **codes syntaxiques** : majuscules, minuscules...
- o Donner des **noms explicites** (classes, méthodes, attributs)
- o Développer une **documentation** du code (cf cours javadoc)
- o ... Et proposer une **vision synthétique** d'un ensemble de classes : \Rightarrow **UML : Unified Modeling Language**

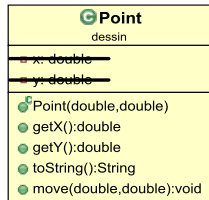


- o nom de la classe
 - o attributs
 - o méthodes (et constructeurs)
- + code pour visualiser **public/private**
- + liens entre classes pour les dépendances (cf cours sur la composition)

UML CLIENT vs FOURNISSEUR

Plusieurs types de diagrammes pour plusieurs usages :

- o Vue **fournisseur** : représentation complète
- o Vue **client** : représentation public uniquement



Idée :

Le code doit être pensé pour les autres :

- o tous les noms doivent être aussi clairs que possible
- o un diagramme plus limité est plus facile à lire

REPRÉSENTATION UML (SUITE)

Deux manières de voir l'UML :

- 1 Outil pour une **visualisation** globale d'un code complexe
- 2 Outil de **conception** / développement indépendant du langage

Dans le cadre de LU2IN002 : seulement l'approche 1

Limites de l'UML :

- o Vision architecte...
- o Mais pas d'analyse de l'exécution du code

PLAN DU COURS

- 1 Rappels
- 2 Destruction d'objets (suite)
- 3 Représentation UML
- 4 Objets composés, composition d'objets
- 5 Classes enveloppes
- 6 Égalité entre objets, clonage
- 7 Javadoc, débogage : devenir autonome...

COMPOSITION : REPRÉSENTATION MÉMOIRE

- o Classe pour créer des segments
 - la classe **Point** est donnée par ailleurs (cf. cours précédent)

```
1 public class Segment{
2     private Point a,b;
3     public Segment(Point a, Point b) {
4         this.a = a;
5         this.b = b;
6     }
7     ...
8 }
```

- o Utilisation de la classe **Segment**
 - **création de plusieurs objets** : les points, et le segment

```
1 public class TestSegment{
2     public static void main(String[] args) {
3         Point p1 = new Point(1,2);
4         Point p2 = new Point(3,4);
5         Segment seg1 = new Segment(p1,p2);
6         // ou création des points dans le constructeur:
7         Segment seg2 = new Segment();
8         ...
9     }
```

PHILOSOPHIE & SYNTAXE

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

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

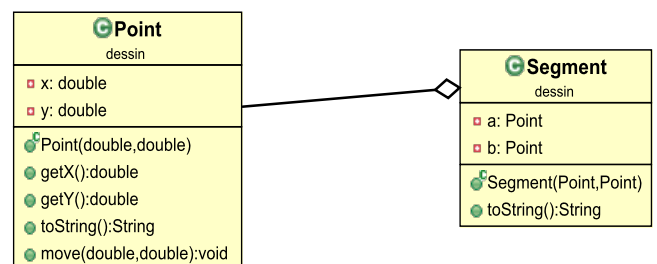
```
1 public class Segment{
2     private Point a, b; // déclaration des variables d'instance
3                         // cf. la classe Point (cours 2)
4     public Segment(Point a, Point b) {
5         this.a = a;
6         this.b = b;
7     }
8     public Segment() {
9         this(new Point(), new Point());
10    }
11    public String toString() {
12        return "Segment[a=" + a + ",b=" + b + "]";
13    }
14    public void move(double dx, double dy) {
15        a.move(dx, dy); // méthode publique de Point
16        b.move(dx, dy);
17    }
18 }
```

REPRÉSENTATION DES LIENS UML

```
1 public class Segment{
2     private Point a,b;
3     ...
4 }
```

Représentation UML :

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



PLAN DU COURS

- 1 Rappels
- 2 Destruction d'objets (suite)
- 3 Représentation UML
- 4 Objets composés, composition d'objets
- 5 Classes enveloppes
- 6 Egalité entre objets, clonage
- 7 Javadoc, débogage : devenir autonome...

PLAN DU COURS

- 1 Rappels
- 2 Destruction d'objets (suite)
- 3 Représentation UML
- 4 Objets composés, composition d'objets
- 5 Classes enveloppes
- 6 Egalité entre objets, clonage
- 7 Javadoc, débogage : devenir autonome...

TYPES DE BASE vs OBJET : SIGNIFICATION DE =

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

- o 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; // duplication de la valeur 1  
⇒ Si b est modifié, pas d'incidence sur a  
o et pour un Objet :  
4 Point p = new Point(1,2);  
5 Point q = p; // duplication de la référence ...  
6 // 1 seule instance !
```



2 variables mais 1 seule instance

CLASSES ENVELOPPES

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

- o fournir quelques outils très utiles
- o utiliser les classes génériques (cf cours ArrayList)

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

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

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

PROBLÉMATIQUE

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

Vocabulaire (uniquement pour les opérations sur objets)

new : instantiation / création d'instance

= : duplication de référence

== : égalité **référentielle**



RÉFÉRENCES ET ARGUMENTS DE FONCTIONS

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

```
1 public class UnObjet{  
2 ...  
3 public void maFonction1(Point p){  
4 ...  
5 p.move(1., 1.); // ajout à x et y  
6 ...  
7 }  
8  
9 public void maFonction2(double d){  
10 ...  
11 d = 3.; // syntaxe correcte  
12 // mais très moche !  
13 ...  
14 }  
  
1 // dans le main  
2 UnObjet obj = new UnObjet();  
3  
4 Point q = new Point(1., 2.);  
5 double d = 2.;  
6  
7 obj.maFonction1(q);  
8 obj.maFonction2(d);  
9  
10 // q a pour attributs (x=2.,y=3.)  
11 // d vaut 2
```

- o Quand un type de base est passé en argument : duplication.
- o 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)

```

1 double d1 = 1.;
2 double d2 = 1.;
3 System.out.println(d1==d2); // affichage de true
4                               //dans la console
5
6 Point p1 = new Point(1, 2);
7 Point p2 = p1;
8 System.out.println(p1==p2); // affichage de true
9
10 Point p3 = new Point(1, 2);
11 Point p4 = new Point(1, 2);
12 System.out.println(p3==p4); // affichage de false
13
14 Double d3 = 1.; // classe enveloppe Double = objet
15 Double d4 = 1.;
16 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)**

```

1 // Dans la classe Point
2 public boolean egalite(Point p){
3     if (p == null)
4         return false;
5     return p.x == x && p.y == y;
6 }
7
8 // Dans le main
9 Point p1 = new Point(1., 2.);
10 Point p2 = p1;
11 Point p3 = new Point(1., 2.);
12 Point p4 = new Point(1., 3.);
13
14 p1.egalite(p2); // true
15 p1.egalite(p3); // true
16 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 ? (2)

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

```

1 // Exemple de la classe Point
2 public boolean equals(Object obj) {
3     if (this == obj)
4         return true;
5     if (obj == null)
6         return false;
7     if (getClass() != obj.getClass())
8         return false;
9     Point other = (Point) obj;
10    if (x != other.x)
11        return false;
12    if (y != other.y)
13        return false;
14    return true;
15 }
    
```

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

```

1 // Constructeur de Point a partir d'un autre Point
2 public Point(Point p){
3     this.x = p.x; // utiliser this ici est facultatif
4     this.y = p.y; // utiliser this ici est facultatif
5 }
    
```

- Usage :

```

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

COPIE D'OBJET COMPOSÉ : LE PIEGE

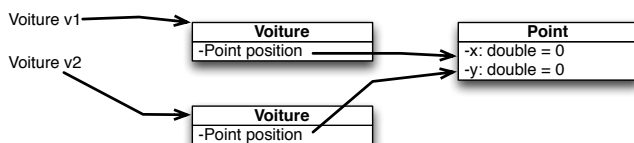
Cas classique : besoin de dupliquer une **Voiture** dont la position est définie par un attribut **Point**

Implémentation **INCORRECTE** :

```

1 // Dans voiture
2 public Voiture(Voiture v){
3     position = v.position
4 }
5 // dans le main
6 Voiture v1 = new Voiture(new Point(0,0));
7 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

```

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

- Usage :

```

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

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

PLAN DU COURS

- 1 Rappels
- 2 Destruction d'objets (suite)
- 3 Représentation UML
- 4 Objets composés, composition d'objets
- 5 Classes enveloppes
- 6 Egalité entre objets, clonage
- 7 Javadoc, débogage : devenir autonome...
 - déboguer son programme
 - se documenter et documenter soi-même

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

LES BONNS 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

o Compileur

- **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**)

o 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

Point
dessin
<ul style="list-style-type: none">x: doubley: double
<ul style="list-style-type: none">Point(double,double)getX():doublegetY():doubletoString():Stringmove(double,double):void

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
```

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

ERREURS USUELLES À CORRIGER SOIT MÊME

Execution (JVM) : Toujours vérifier la ligne également

o 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

o IndexOutOfBoundsException

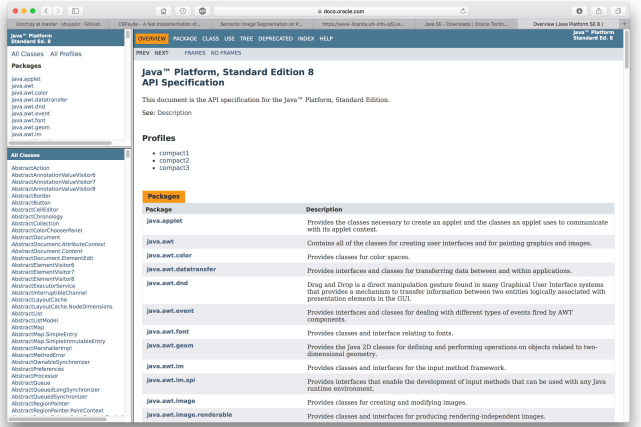
```
1 int[] tab = new int[3];
2 tab[3] = 2;
3 // Exception in thread "main"
4 // java.lang.ArrayIndexOutOfBoundsException: 3
5 // at cours1.TestPoint.main(TestPoint.java:2)
```

- 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

```
1 // **
2  * @author Vincent Guigues
3  * Cette classe permet de gérer des points en 2D
4  */
5  public class Point {
6      /**
7       * Attributs correspondant aux coordonnées du point
8       */
9      private double x, y;
10     /**
11      * Constructeur standard à partir de 2 réels
12      * @param x : abscisse du point
13      * @param y : coordonnée du point
14      */
15     public Point(double x, double y) {
16         this.x = x;
17         this.y = y;
18     }
19     /**
20      * @return l'abscisse du point
21      */
22     public double getX() {
23         return x;
24     }
25 }
26
```

\$ javadoc Point.java

JAVADOC : QUELQUES OPTIONS UTILES

- o De manière générale : vérifier la documentation

\$ javadoc -h

- o Pour gérer les accents :

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

- o Pour sélectionner le répertoire de stockage du html :

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

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

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

JAVADOC : RÉSULTATS OBTENUS

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

