LU2IN002 - Introduction à la programmation orientée-objet

Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ

Cours 2 – 23 septembre 2022

Programme du jour

1. Création et utilisation d'objets
2. Surcharge
3. Cycle de vie des objets
4. Classes enveloppes

Plan du cours

1. Création et utilisation d'objets
2. Surcharge
3. Cycle de vie des objets
4. Classes enveloppes

Créer un objet : instanciation

Coté fournisseur :
mise en route de l'objet

Constructeur = contrat d'initialisation des attributs

)

Coté client :
création d'une instance de classe

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

```java
//Fichier Point.java
public class Point{
  private double x,y;

  public Point(double x2,double y2){
    x = x2;
    y = y2;
  }
}
```

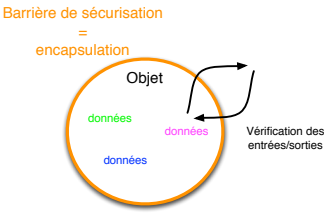
```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 2.);
  }
}
```

La variable p1, de type Point, référence une instance de la classe Point dont les attributs ont pour valeur 1 et 2.

Syntaxe : méthodes

Comment manipuler un Point ?

1. **définir des méthodes**
   eg : accéder aux attributs (en lecture)
2. les **invoquer** depuis l'extérieur



Fournisseur

```java
//Fichier Point.java
public class Point{
 private double x,y;
 public Point(double x2, double y2){
    x = x2;
    y = y2;
 }

 public double getX(){
    return x;
 }

 }
```

Client

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){

    // construction d'un point:
    Point p = new Point(2., 3.1);

    double px = p.getX();

    System.out.println(
          "coord x de p:"+px);
  }
}
```


Autorisations d'accès

- `public` : accessible / visible depuis l'extérieur de l'objet (eg : un main, un autre objet...)
- `private` : protégé / invisible depuis l'extérieur de l'objet

- Les constructeurs sont en général `public`
  - ils ont vocation à être appelés depuis l'extérieur
- Les attributs sont en général `private`
  - ils sont protégés et non accessibles depuis l'extérieur

Fournisseur

```java
//Fichier Point.java
public class Point{

  private double x,y;

  public Point(double x2, double y2){
    x = x2;
    y = y2;
  }
}
```

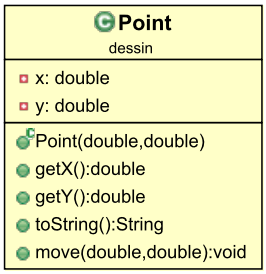
Client

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){

    // opération autorisée
    Point p = new Point(2., 3.1);

    // opération impossible :
    // ERREUR DE COMPILATION
    double d = p.x;
  }
}
```

Représentation UML

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

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



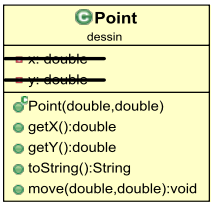
- nom de la classe
- attributs
- méthodes (et constructeurs)

\+ 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 :

- Vue fournisseur : représentation complète
- Vue client : représentation `public` uniquement



Idée :

Le code doit être pensé pour les autres :

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

Reflexion sur la syntaxe objet

Exemple type : addition de deux instances de `Point`
⟹ méthode `add` à écrire
elle permet d'obtenir une nouvelle instance dont les coordonnées sont les sommes respectives des `x` et `y` des instances à additionner

Client

```
1  //Fichier TestPoint.java
2  public class TestPoint{
3    public static void main
4                       (String[] args){
5
6      // construction d'un point:
7      Point p  = new Point(2., 3.1);
8      Point p2 = new Point(0.5, 1);
9
10     Point p3 = p.add(p2);
11   }
12 }
```

Fournisseur

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

Syntaxe objet = il faut penser objet... Pas évident au début !

Syntaxe : redéfinition de méthodes standards

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

Fournisseur

```
1  //Fichier Point.java
2  public class Point{
3   private double x,y;
4   ...
5   public String toString(){
6      // méthode redéfinie
7      return "["+ x +","+ y +"]";
8   }
9  }
```

Client

```
1  //Fichier TestPoint.java
2  public class TestPoint{
3    public static void main
4                       (String[] args){
5
6      // construction d'un point:
7      Point p = new Point(2., 3.1);
8
9      String str = p.toString();
10
11     System.out.println("p:" + str);
12   }
13 }
```

```
» p: Point@8764152
```

```
» p: [2, 3.1]
```


Classe String

Gestion des chaînes de caractères

`String` n'est pas un type de base, c'est un objet qui se comporte différemment des types de base... Mais c'est une classe complètement intégrée à Java et son caractère immutable la rapproche très nettement d'un type de base.

```
1 String s = "Luke"; // création d'une chaîne de caractères
2 s = s + "␣est␣le␣frère␣de␣Leia";
3 System.out.println(s); // affichage de s dans la console
```

⚠ Ne pas confondre l'**objet** `String` et l'affichage dans la console.

Les **possibilités sont nombreuses** : extraction de sous-chaînes (substring), division en plusieurs chaînes (split), recherche de caractères, construction de nouvelles chaînes à partir d'expressions régulières (replace)... Toute la documentation sur : `http://docs.oracle.com/javase/7/docs/api/java/lang/String.html`

String (suite)

2 choses à retenir sur les `String`

1. **Les chaînes sont immutables** : modifier une chaîne existante est impossible, il faut créer une nouvelle chaîne qui est une modification de l'ancienne.
2. Égalité de chaînes : **ne pas utiliser `==` avec les `String`** mais la méthode `.equals`.

```
1 String s1 = "Leia";
2 String s2 = "Luke";
3 if( s1.equals(s2) )
4     System.out.println("les␣chaînes␣sont␣identiques");
5 else
6     System.out.println("les␣chaînes␣sont␣différentes");
```

Formatages de chaînes

```
double[] tab = new double[10];
for(int i=0; i<10; i++)
    tab[i] = Math.random()*1000;
for(int i=0; i<10; i++)
    System.out.println(tab[i]);
```

```
510.4306229034564
775.6503067597263
15.528224029893511
...
```

```
for(int i=0; i<10; i++)
    System.out.println(
      String.format("%12f", tab[i]));
```

```
  510,430623
  775,650307
  15,528224
...
```

```
for(int i=0; i<10; i++)
    System.out.println(
      String.format("%10.3f", tab[i]));
```

```
  510,431
  775,650
   15,528
...
```

```
for(int i=0; i<10; i++)
    System.out.println(
      String.format("%010.3f", tab[i]));
```

```
000510,431
000775,650
000015,528
...
```

Ca marche aussi avec les entiers (%d) et les String (%s)

Plan du cours

1. Création et utilisation d'objets
2. Surcharge
3. Cycle de vie des objets
4. Classes enveloppes

Syntaxe : surcharge du constructeur

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

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

⇒ Syntaxe triviale : il suffit de définir plusieurs constructeurs !

CONTRAINTE : les signatures doivent être différentes

Fournisseur

```
//Fichier Point.java
public class Point{
  private double x,y;

  public Point(double x2, double y2){
    x = x2;
    y = y2;
  }
  public Point(double d){ // surcharge
    x = d;
    y = d;
  }
  public Point(){ // autre surcharge
    // aléatoire entre 0 et 10
    x = Math.random()*10;
    y = Math.random()*10;
  }
```

Client

```
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){

    // construction d'un point:
    Point p = new Point(2., 3.1);
    // construction d'un autre point:
    Point p2 = new Point();
    // construction d'un 3e point:
    Point p3 = new Point(4.2);

  }
}
```


Surcharge de méthode

Définition
- Même nom de fonction, arguments différents
- Le type de retour ne compte pas

```
public class Point {
  private double x,y;
    ...
    public void move(double dx, double dy){
        x+=dx; y+=dy;
    }
    public void move(double dx, double dy, double scale){
        x+=dx*scale; y+=dy*scale;
    }
    public void move(int dx, int dy){
        x+=dx; y+=dy;
    }
    public void move(Point p){
        x+=p.x; y+=p.y;
    }
```

Rappel : dans la classe, accès total aux attributs privés des autres instances de la classe

Le mot clé this (1/5)

- Variables d'instances, arguments et variables locales

```
public class Point {
    private double x,y;
    ...
    public void move(double dx, double dy){
        x += dx;
        y += dy;
    }
    public void move(double dx, double dy, double scale){
        x += dx*scale;
        y += dy*scale;
    }
    public void move(int dx, int dy){
        x += dx;
        y += dy;
    }
    public void move(Point p){
        x += p.x;
        y += p.y;
    }
```


Le mot clé this (2/5)

- `this` : référence de l'objet courant
- ⟶ moyen pour un objet d'accéder à ses attributs et méthodes

```
public class Point {
    private double x,y;
    ...
    public void move(double dx, double dy){
        this.x += dx;
        this.y += dy;
    }
    public void move(double dx, double dy, double scale){
        this.x += dx*scale;
        this.y += dy*scale;
    }
    public void move(int dx, int dy){
        this.x += dx;
        this.y += dy;
    }
    public void move(Point p){
        this.x += p.x;
        this.y += p.y;
    }
```

Le mot clé this (3/5)

- `this` : référence de l'objet courant
- ⟶ moyen pour un objet d'accéder à ses attributs et méthodes

```java
public class Point {
    private double x,y;
    ...
    ...
    public void changeEn(double x, double y){
        this.x = x;
        this.y = y;
    }
```


Le mot clé this (4/5)

- `this` : référence de l'objet courant
- Comment éviter de dupliquer des traitements ?

```java
public class Point {
    private double x,y;

    public Point(double x2, double y2){
        x = x2;   // ou bien this.x = x2
        y = y2;   //  ou bien this.y = y2
    }
    public Point(double d){ // surcharge
        x = d;
        y = d;
    }
    public Point(){ // autre surcharge
        // aléatoire entre 0 et 10
        x = Math.random()*10;
        y = Math.random()*10;
     }
```


Le mot clé this (5/5)

- `this` : référence de l'objet courant
- `this()` : appel d'un autre constructeur pour l'objet courant
- IMPORTANT : `this()` doit être la première instruction du constructeur

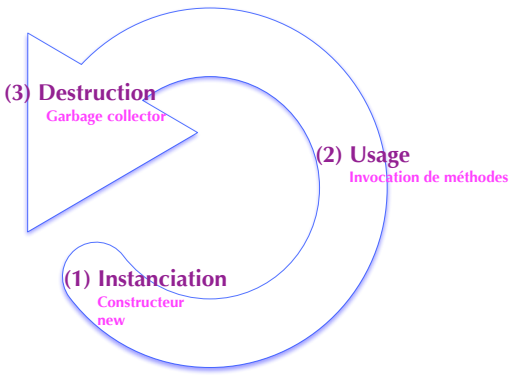
```java
public class Point {
    private double x,y;

    public Point(double x2, double y2){
        x = x2;   // ou bien this.x = x2
        y = y2;   // ou bien this.y = y2
    }
    public Point(double d){ // surcharge
        this(d,d); // appel du constructeur Point(double, double)
    }
    public Point(){ // autre surcharge
        // aléatoire entre 0 et 10
        this(Math.random()*10,Math.random()*10);
     }
```


Plan du cours

1. Création et utilisation d'objets
2. Surcharge
3. Cycle de vie des objets
4. Classes enveloppes

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

```java
//Fichier Point.java
public class Point{
  private double x,y;

  public Point(double x2,double y2){
    x = x2;
    y = y2;
  }
}
```

Coté client :
création d'une instance

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

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 2.);
  }
}
```

La variable `p1`, de type `Point`, référence un instance de `Point` dont les attributs ont pour valeur 1 et 2.

(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.

```java
//Fichier Point.java
public class Point{
  private double x,y;
  public Point(double x2,double y2){
    x = x2;   y = y2;
  }

  public void move(double dx,
                   double dy){
    x += dx; y += dy;
  }
  ...
}
```

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 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) ?

```java
Point p;
```

- p vaut null.
- on peut écrire de manière équivalente :

```java
Point p = null;
```

- on ne peut pas invoquer de méthode :

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

```java
// classe UnObjet,
// (classe sans importance)
...
public void maFonction(Point p){
  ...
  p.move(1., 1.); // si p non null
  ...
}
```

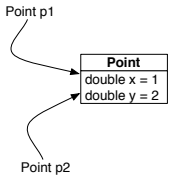
```java
// dans le main
UnObjet obj = new UnObjet();

obj.maFonction(null);
// La méthode doit gérer !
```


(3) Destruction

1. Un objet est détruit lorsqu'il n'est plus référencé
   - c'est-à-dire quand aucune variable ne contient sa référence
2. La destruction est implicite (contrairement au C++) et traitée en tâche de fond (garbage collector)

- Un objet peut être référencé plusieurs fois...

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 2.);
    Point p2 = p1;
  }
}
```

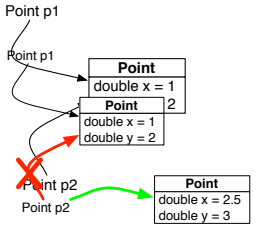


- mais quand est-il dé-référencé ?

Dé-référencement d'un objet

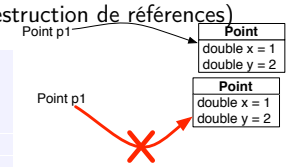
1. Dé-référencement explicite (usage de =)

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 2.);
    Point p2 = p1;
    p2 = new Point(2.5, 3.);
  }
}
```



2. Dé-référencement implicite (logique de bloc, destruction de variables ⇒ destruction de références)

```java
for (int i; i<10;i++) {
      Point p1 = new Point(1., 2.);
      System.out.println(p1);
  }
  System.out.println(p1);
  // ERREUR DE COMPILATION
  // 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**

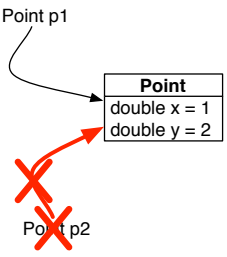
```java
public static
    void main(String[] args) {

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

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

```java
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

```java
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
}
```

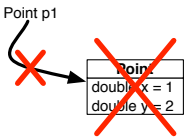


- Fin de bloc = destruction des **variables** déclarées dans le bloc
- Destruction d'instance ⇔ l'instance n'est plus référencée

Destruction des instances

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

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



- Pas besoin de dire comment détruire un objet
- 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 }
```

- 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;
```

- 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.

```
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;
```

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

Plan du cours

1 Création et utilisation d'objets
2 Surcharge
3 Cycle de vie des objets
4 Classes enveloppes

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

```
1 Double d1 = 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
8 Double d5 = d4;       // boxing
```