LU2IN002 - Introduction à la programmation objet

Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ

Cours 6 – 11 octobre 2024

Programme du jour

1. Retour sur static
2. Héritage
3. Héritage : principe de subsomption

Plan du cours

Fonction static

- Boîte à outils (quelques exemples) :
  - génération de nom aléatoire (lettre aléatoire ou alternance voyelles/consonnes)
  - distance entre Point (formulation alternative à celle intra-classe),
    - possibilité de définitions multiples pour prendre en compte des contraintes
    - optimisation ultérieure
- L'exemple de la classe Math

Utilisation de static : l'exemple du singleton

- Idée : comment garantir qu'une classe ne puisse n'avoir qu'une unique instance ?
- Approche du Singleton :
  - blocage de l'accès au constructeur pour contrôler la création d'instance
  - méthode pour obtenir LA SEULE instance existante

```
1  public class Singleton {
2      // variable de classe finale pour stocker une référence
3      // et une seule
4      private static final Singleton INSTANCE = new Singleton();
5
6      // constructeur privé: interdiction de créer des objets
7      // en dehors de la classe
8      private Singleton() {}
9
10     // méthode public pour récupérer la référence de l'unique
11     // objet créé
12     public static Singleton getInstance() {return INSTANCE;}
13 }
```


Un exemple d'utilisation du singleton

- Une classe pour représenter l'origine du repère orthonormé : l'origine est un Point unique

```
1  public class Origine {
2      private static final Origine INSTANCE = new Origine(0,0);
3      private double x, y;
4
5      private Origine(double x, double y) {
6        this.x = x ; this.y = y ;
7      }
8
9      public static Origine getInstance() {
10       return INSTANCE;
11     }
12
13     public static double distanceAOrigine(Point p) {
14       return Math.sqrt( p.getX()*p.getX() + p.getY()*p.getY());
15     }
16
17     public String toString() { // méthode non statique !
18           return "origine␣("+this.x+",␣"+this.y+")";
19     }
20 }
```

Un exemple d'utilisation

- Une classe pour représenter l'origine d'un repère orthonormé

```
1  public class TestOrigine {
2      public static void main(String[] args) {
3          Point p1 = new Point(3, 2);
4          System.out.println(p1);
5
6          Origine orig = Origine.getInstance();
7          System.out.println(orig);
8
9          System.out.println( "Distance entre " + orig + " et "
10                             + p1 + ":" + Origine.distanceAOrigine(p1) );
11     }
12 }
```

- Résultat :

```
1 (3.0, 2.0)
2 origine (0.0, 0.0)
3 Distance entre origine (0.0, 0.0) et (3.0, 2.0): 3.605551275
```


static / non static : asymétrie

Rappel

- Les instances voient ce qui est static
- Les éléments static ne voient pas les instances

```
1  public class Point{
2    private static int cpt = 0;
3    private int id;
4    private double x,y;
5    ...
6    // Cas 1: OK méthode static, accès variable static
7    public static int getCpt(){return cpt;}
8    // Cas 2: OK méthode d'instance, accès variable static
9    public  int getCptInst(){return cpt;}
10   // Cas 3 : KO méthode static, accès variable d'instance
11   public static int getID(){return id;} // non sens!!
```

Depuis le main :

```
1 Point p1 = new Point();
2 Point.getCpt();    // OK syntaxe  naturelle
3 p1.getCpt();       // OK syntaxe possible (mais non recommandée)
4 Point.getCptInst(); // KO! syntaxe impossible (évidemment) :
```


Bilan...

Quand on vous parle de static, n'oubliez pas :

- ce sont des cas très particuliers
- assez rare
- n'oubliez pas les bonnes pratiques de la POO!!!!!

Plan du cours

Principes Orientés Objets

Principe 1 : Encapsulation

- Rapprochement données (attributs) et traitements (méthodes)
- Protection de l'information (private/public)

Principe 2 : Composition/Association

- Classe A est composé d'un objet de la Classe B
- Classe A utilise la Classe B

Principe 3 : Héritage

- Un objet de la classe B **est un** objet de la classe A aussi
- La classe B **hérite** de la classe A

Héritage

Idée de l'héritage

Spécialiser une classe, ajouter des fonctionnalités dans une classe
$\Longrightarrow$ Hériter du comportement d'une classe existante

- **Une classe $\Longrightarrow$ plusieurs spécialisations possibles**
  - Animal $\rightarrow$ Vache, Chien, Panda...
  - hiérarchisation possible : Animal $\rightarrow$ Insecte $\rightarrow$ Papillon
- **Objectif :** Ne pas avoir à modifier le code existant
  - ne pas modifier la classe de base
  - Point $\rightarrow$ PointNomme : un point avec un nom
- **Ne pas avoir à faire de copier-coller !**
  - faire hériter le comportement d'une classe

Exemples & contre exemples

Pour les cas suivants : dire si les relations sont des relations de type **Composition/Association** ou **Héritage** :

- Salle de bains et baignoire
- Piano et pianiste
- Personne, enseignant et étudiant
- Animal, chien et labrador
- Cercle et ellipse
- Entier et réel

Exemple : Point et PointNomme

Problème

Soit deux deux classes à implémenter pour représenter :

1. un point en 2 dimensions
2. un point en 2 dimensions qui possède un nom

```java
public class Point { // (1) Type / nom de classe
  private double x;    // (2)  Attributs
  private double y;
  public Point(double x, double y) { // (3) Constructeurs
    this.x = x;
    this.y = y;
  }
  public double getX() {   // (4) Accesseurs
    return x;
  }
  public double getY() {
    return y;
  }
  // à suivre...
```


Exemple : Point et PointNomme

```java
  // ... suite de la classe Point
  // (4) méthodes (traitements)
   public double calculeDistance(Point p2) {
     double dx = Math.abs(p2.getX() - getX());
     double dy = Math.abs(p2.getY() - getY());
     return Math.sqrt(dx*dx+dy*dy);
  }

  public void move(double tx, double ty) {
    deplace(x+tx,y+ty);
  }
  // (4) méthodes privées
  private void deplace(double x, double y) {
    this.x=x; this.y=y;
  }
  // (5) méthodes standards
  public String toString() {
    return "("+x+","+y+")";
  }
}
```


Héritage, syntaxe : le mot-clé extends

```java
public class PointNomme extends Point {
  // PointNomme hérite de Point
  private String name;

  public PointNomme(double x, double y, String name) {
      super(x, y);
      this.name = name;
  }

  public String toString() {
    return "PointNomme [name=" + name + "] " +
          super.toString() +  "]";
  }
}
```

- Deux nouveaux mots-clés :
  - `extends` dans la signature de la classe
  - `super` (à voir plus tard)

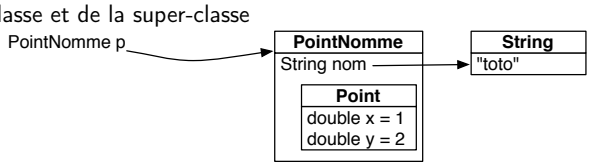
Erreur courante

Attention à ne pas dupliquer les attributs : la classe fille étend sa classe mère, la classe fille contient la super-classe

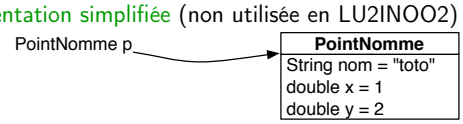
Représentation mémoire

```java
PointNomme p = new PointNomme(1, 2, "toto");
```

1. Représentation complète : tous les objets sont séparés, on représente explicitement la séparation entre attributs de la classe et de la super-classe



2. Représentation simplifiée (non utilisée en LU2INOO2) :



Attention : cette dernière représentation donne une vision trompeuse...

Remarques / vocabulaire

- Point est la classe mère de la classe PointNomme
  - Point est la super-classe de la classe PointNomme
- PointNomme est une classe fille de Point
  - PointNomme hérite de la classe Point
  - PointNomme étend la classe Point
  - PointNomme dérive de Point
- Une instance de PointNomme contient une instance de Point

Construction d'une instance de la classe fille

Principe : 2 étapes

1. appeler le constructeur de la classe mère
2. initialiser les attributs propres à la classe fille

```
public class PointNomme extends Point {
    private String name;
    public PointNomme(double x, double y, String name) {
        super(x, y); // obligatoirement en 1ère instruction
        this.name = name; // init. attributs de la classe fille
    }
```

- **Règle générale :** choisir un constructeur dans la super-classe et l'appeler avec `super(...)`
- **Cas particulier :** si la super-classe a un constructeur sans argument, l'appel à `super()` est implicite

Cas particulier : super()

S'il existe un constructeur accessible et sans argument dans la super-classe :

```
public class Point {
    private double x,y;
    public Point(){
        x=0; y=0;
    }
    ...
```

Alors, les deux écritures suivantes sont équivalentes :

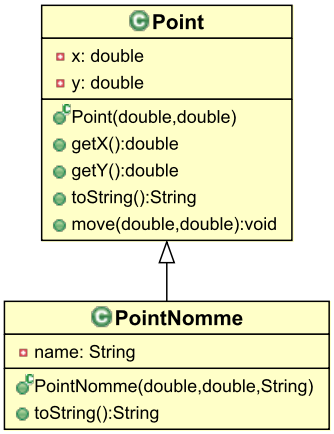
```
public class PointNomme
                extends Point {
private String name;

public PointNomme(String name) {
        super();
        this.name = name;
    }
    ...
```

```
public class PointNomme
                extends Point {
private String name;

public PointNomme(String name) {
        this.name = name;
    }
    ...
```


Représentation des liens UML



- Extension des capacités/propriétés d'un objet
- `PointNomme p = new PointNomme(1,2,"p1");`
- `p` est un PointNomme
- `p` est aussi un Point :
  - accès à toutes les méthodes
  - `p.getX(), p.getY()`...

Vision client : utilisateur de la classe

PointNomme
PointNomme(x, y, nom)
getX() : double getY() : double move(tx :double, ty :double) : void toString() : String **getNom() : String**

```
PointNomme p =
    new PointNomme(1,2,"p1");

System.out.println(p.getX());
// méthode définie dans Point
// héritée dans PointNomme
// utilisée de manière transparente
```

- Méthodes publiques de Point (mais pas les constructeurs)
- Pas de vision sur les données private

Nouveau niveau d'accès : protected

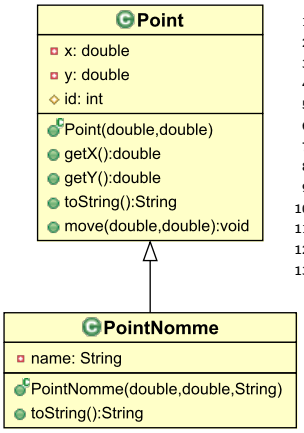
Niveau de visibilité intermédiaire

- `protected` : visibilité intermédiaire entre private et public

Récapitulatif :

- `public` : visible partout, dans la classe et chez le client
- `protected` : visible dans la classe et dans ses descendantes
- `private` : visible dans la classe uniquement

Exemple d'accès protected




```
public class Point {
  private double x,y;
  // les classes dérivées y ont accès
  protected int id;
  ...
}
public class PointNomme extends Point {
  ...
  void methode(double d){
    int toto = id; // ou super.id;
    ...
  }
}
```

Variable/Méthode protected

- Accès depuis la classe
- Accès depuis les classes filles
- Pas d'accès depuis une classe quelconque

Une classe ⇒ 3 visions possibles : développeur, héritier, client

Héritage : propriétés

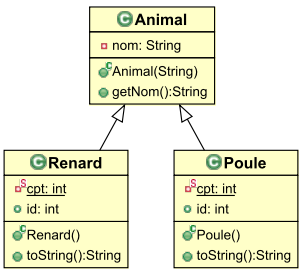
Si B hérite de A, implicitement, B hérite :

- des **méthodes publiques** de A
- des **méthodes protégées** de A
- d'un attribut **super** du type de A (super-classe)
  - **super** référence la *partie de B* qui correspond à A

En revanche, B n'hérite pas :

- des **attributs privés** de A
- des **méthodes privées** de A
- des contructeurs **publics, privés ou protégés** de A

Cas particulier : arguments par défaut



- Le constructeur de la super-classe prend des arguments
- Les constructeurs des classes filles non...
  - arguments par défaut

```
1  public class Animal {
2    private String nom;
3    public Animal( String nom) {
4      this.nom = nom;
5    }
6    public String getNom(){
7      return nom;
8    }
9  }
10
11 public class Poule extends Animal{
12   private static int cpt = 0;
13   public int id;
14
15   public Poule() {
16     super("poule");
17     id = cpt++;
18   }
19   public String toString(){
20     return getNom()+"␣␣"+id;
21   }
22 }
```


Conclusion

- Un nouveau paradigme de réflexion, au coeur de la POO
- Des éléments de syntaxe à maitriser
- ... à suivre (**subsomption, surcharge & redéfinition**)

Plan du cours

Subsomption et polymorphisme

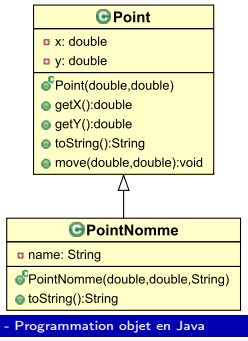
Si la classe B hérite de la classe A :

- le type B EST-UN type A
- les méthodes de A peuvent être invoquées sur une instance de la classe B (+ transitivité : A hérite de sa super-classe qui hérite de etc.)
- **Subsomption :** dans toute expression qui attend un A, on peut utiliser un B à la place

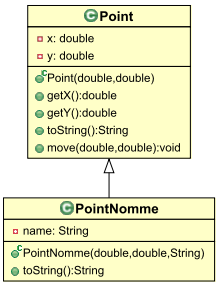
Polymorphisme = exploitation de la subsomption

```
1 Point p = new PointNomme(1,2,"toto");
```

Un `PointNomme` EST UN `Point` ⇒
on peut le traiter comme tel



Polymorphisme : représentation mémoire



- Syntaxe :

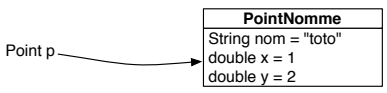
```
1 Point p = new PointNomme(1,2,"toto");
2 // PointNomme EST UN Point
```

- Attention : un Point ≠ un PointNomme

```
1 p.getNom(); // -> ERREUR de compilation
```

Rôle du compilateur :
il vérifie le type des variables est les possibilités offertes par celles-ci.

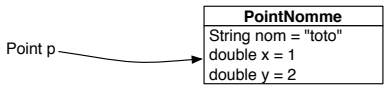
Représentation mémoire :



⇒ bien distinguer le type des instances et des variables

Polymorphisme : visions compilateurs *vs* JVM

```
1 Point p = new PointNomme(1,2,"toto");
```



Compilateur

La variable p est de type Point : seule les méthodes de Point sont accessibles :

- \+ `p.getX(); p.getY(); //...`
- − `p.getNom();`
  ⇒ Erreur de compilation (méthode inconnue dans la classe Point)

JVM

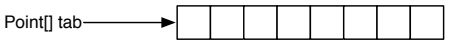
L'instance référencée par p est de type PointNomme

- \+ En cas d'appel à `toString()`, c'est bien la méthode de PointNomme qui est invoquée.

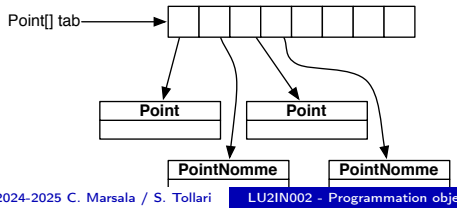
Polymorphisme : passage aux tableaux

Application classique sur les tableaux de concepts abstraits :

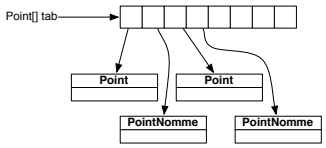
```
1 Point[] tab = new Point[10]; // OK,
2                   // il s'agit de 10 variables de type Point
```




```
3 for(int i=0; i<tab.length; i++){
4   if(i%2==0)
5     tab[i] = new Point(Math.random()*10, Math.random()*10);
6   else
7     tab[i] = new PointNomme(Math.random()*10,
8                             Math.random()*10, "toto"+i);
9 }
```

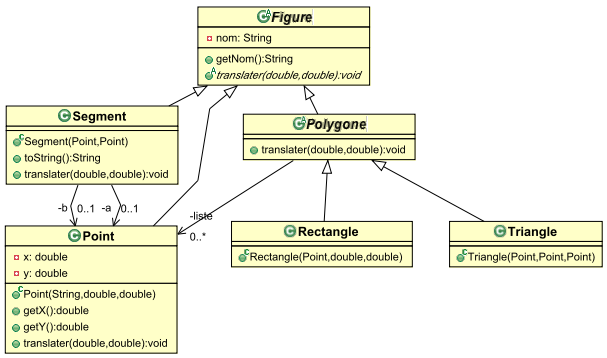


Polymorphisme : application sur un tableau




```
1  // par exemple, procédure de Figure (méthode de classe)
2  public static void translaterTout(Point[] pts,
3                                    double tx, double ty) {
4    for(int i=0;i<pts.length;i++)
5      pts[i].translater(tx,ty);
6  }
7
8  // variante
9
10 public static void translaterTout(Point[] pts,
11                                   double tx, double ty) {
12   for(Point p : pts)
13     p.translater(tx,ty);
14 }
```


Exemple plus complexe



⇒ Bien réfléchir aux opérations à effectuer sur les tableaux

- recherche/affichage d'un nom
- translation

Limites du polymorphisme

- On ne peut invoquer que les méthodes du super-type
  - ex. : si le type est Figure, on ne peut invoquer que les méthodes de Figure, même si l'instance est un Point

```
1 Figure fig1 = new Carre(2,1,4,5);
2 fig1.translater(2,2);  // OK type Figure
3 double x = fig1.calculerSurface(); // KO type Figure !
4
5 Figure fig2 = new Point(2,1);
6 fig2.translater(2,2);  // OK type Figure
7 fig2.getX(); // KO type Figure !
```

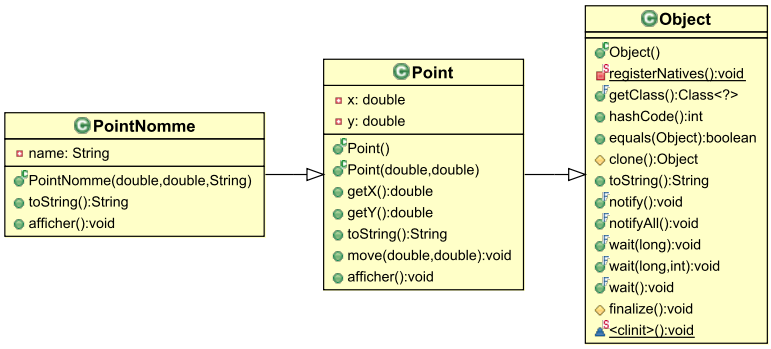
Role du compilateur :

il vérifie le type des variables est les possibilités offertes par celles-ci.

Classe Object

Classe standard

Toutes les classes dérivent de la classe Object de JAVA



Cet héritage est implicite, pas de déclaration dans la signature