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 7 – lundi 24 octobre 2022

Plan du cours

Programme du jour

Nouveaux concepts

- **Classe abstraite**
  - Classe qui ne sera pas instanciable
  - Les classes filles pourront être instanciables
  - Exemple :
    - Animal (abstraite) : définit un comportement général
    - Mouton, Tigre : animaux avec comportements spécifiques
- **Méthode abstraite**
  - Seulement dans les classes abstraites
  - Elle contient une signature mais pas de code
  - Exemple :
    - Animal (abstraite) : `String regimeAlimentaire();`
    - Mouton, Tigre : `"herbivore"`, `"carnivore"`

Classe abstraite

Définition

- Représente une classe qui ne peut pas être instanciée
- Un concept unificateur qui permet de factoriser du code pour toutes les classes qui hériteront
  - Tous les animaux (moutons, tigres...) ont un nom, un age...
    $\Longrightarrow$ On peut factoriser nom, age... dans la classe `Animal`
- Introduction de la notion de contrat : toutes les classes filles devront gérer ce qui est décidé par la classe mère (signature de méthode abstraite)
  - Tous les animaux ont un régime alimentaire ...
    $\Longrightarrow$ La signature de la méthode `regimeAlimentaire()` se trouve dans la classe mère `Animal`
  - ... mais la nature du régime dépend si c'est mouton, tigre...
    $\Longrightarrow$ Le code de la méthode `regimeAlimentaire()` se trouve dans chaque classe fille

Classe et méthode abstraite : syntaxe

```
1 public abstract class Figure {
2     public Figure() { }
3     public abstract String getTypeFigure(); // signature seulement
4 }                                           // pas d'accolades
```

- On ne peut pas créer d'instance de la classe `Figure`

```
new Figure(); // ERREUR compilation : la classe est abstract
```

- Des classes peuvent hériter de `Figure`, elles devront :
  - soit implémenter `getTypeFigure()`

```
public class Point extends Figure {
    ...
    public String getTypeFigure() { // code de la méthode
        return "Point";   // dépend de chaque classe fille
    }
}
```

  - soit être elles-même abstraites

```
public abstract class Polygone extends Figure {
    ...
}
```


Propriétés des classes abstraites

Les classes abstraites sont des classes comme les autres.
Elles peuvent avoir :

- des attributs
- des constructeurs
- des méthodes

mais en plus elles peuvent avoir des méthodes abstraites.

```
1  public abstract class Figure { // abstract
2      private double x, y ;
3      public Figure(double x, double y) {
4          this.x=x; this.y=y;
5      }
6      public void move(double x, double y) {
7          this.x=x; this.y=y;
8      }
9      public abstract String getTypeFigure(); // abstract
10 }
```

Idée

Les classes abstraites sont **pensées pour leurs descendantes**, les classes filles qui en seront dérivées

(Retour) sur les bonnes pratiques

Développement à long terme

modification d'un projet existant = ajout d'une classe

- ne pas modifier les classes existantes
- ajouter des classes filles

Idée

Structurer un projet avec des classes abstraites =

- les classes filles possèdent des fonctionnalités dès leur création
  - factorisation du code
- ajout de **contraintes** sur les classes filles
  - **plus facile** à développer (classe fille = canevas à remplir)
  - **contrat** sur les fonctionnalités (garanties)
  - garanties sur des **classes qui n'existent pas encore** : facilités d'évolution du code
- usage du polymorphisme
  - ex. : tableau hétérogène $\Rightarrow$ + de possibilités

Résumé : classe et méthode abstraite

Résumé de quelques règles à connaître :

- une classe abstraite ne peut être instanciée
- si une méthode est abstraite, alors sa classe doit être abstraite
- si une classe hérite d'une méthode abstraite, elle doit :
  - soit définir le corps de la méthode
  - soit être déclarée abstraite

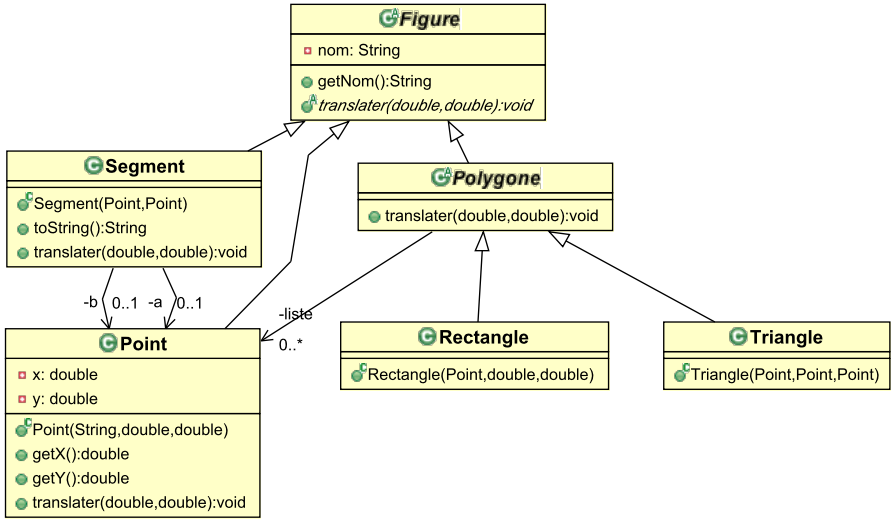
△ Une classe abstraite peut ne pas contenir de méthode abstraite

★ Pourquoi déclarer une classe abstraite si elle ne contient pas de méthode abstract ?

- si on veut empêcher la création d'instance
- si la classe représente une notion abstraite ($\neq$ concrète) pour notre problème
  - Exemple : `Animal` est une notion abstraite par rapport à `Tigre` ou `Mouton` qui sont des notions concrètes pour notre problème

$\Longrightarrow$ pas de sens de créer un objet `Animal` seulement

Exemple du logiciel de dessin



Plan du cours

Récupérer le type d'une instance dynamiquement

Cas amusant :

le type des instances est parfois (souvent) inconnu du développeur

Exemple :

```
1 Figure f;
2 if(Math.random()>0.5)
3   f = new Point(2,3);
4 else
5   f = new Segment(new Point(1,2), new Point(5,3));
```

- Pour **le compilateur**, la syntaxe est correcte :
  - dans tous les cas, `f` référence un objet qui est une `Figure`
- Pour **la JVM**, quel est le type de l'objet référencé par `f` ?
  - Cela dépend de l'exécution...

Récupérer le type d'une instance dynamiquement

- Le compilateur vérifie (statiquement) le type des variables
- Comment connaître le type d'un objet à l'exécution (dynamiquement) ? 2 moyens :
  1. Opérateur `instanceof`
  2. Méthode `getClass()`
- Comment revenir au type initial de l'objet pour accéder aux méthodes spécifiques ?
  - Utiliser un cast

```
1 Figure f = new Point(2,3);
2 f.methodeDePoint(); // Erreur compilation
3 ((Point)f).methodeDePoint(); // OK
4 // OU en 2 instructions :
5 Point p = (Point) f;
6 p.methodeDePoint(); // OK
```


Opérateur instanceof

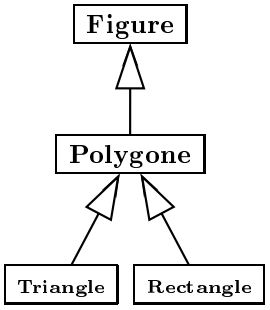
var **instanceof** *NomClasse*

⇒ retourne un boolean

- Retourne `true` si l'objet référencée par la variable `var` est une instance de la classe *NomClasse*
- Retourne `false` sinon (en particulier si `var` est `null`)

```
1 Figure f;
2 if(Math.random()>0.5)
3   f = new Point(2,3);
4 else
5   f = new Segment();
6
7 if(f instanceof Point)
8    System.out.println("C'est un Point (ou descendante de Point)");
9 else
10   System.out.println("Ce n'est pas un Point");
```


Opérateur instanceof




```
Figure f1 = new Rectangle(); // subsomption
```

Quelle est la valeur des expressions suivantes ?

- f1 **instanceof** Rectangle // true
- f1 **instanceof** Triangle // false
- f1 **instanceof** Polygone // true
- f1 **instanceof** Figure // true

⟹ L'objet est un rectangle ... qui est un polygone... qui est une figure, mais n'est pas un triangle

Il faut comprendre `instanceof` comme «EST UN ?»

```
Figure f2 = null;
f2 instanceof Figure // false : pas d'objet
```

```
Figure f3 = new Triangle();
f3 instanceof Rectangle // false : l'objet est un triangle
                        // et un triangle n'est pas un rectangle
```


Opérateur instanceof : discussion (1/2)

△ Attention à **ne pas mal utiliser** `instanceof`

Ex : il ne faut pas utiliser `instanceof` dans une méthode...

- ... de la classe mère pour connaître ses classes filles
- ... quelconque qui distingue toutes les filles connues

★ Pourquoi ?

```
1 public static void afficheType(Figure f) {
2     if(f instanceof Point)
3         System.out.println("C'est un Point");
4     else if(f instanceof Segment)
5         System.out.println("C'est un Segment");
6     else if(f instanceof Rectangle)
7         System.out.println("C'est un Rectangle");
8     // etc ... un if par classe fille de Figure connue
9 }
```

Que se passe-t-il si on écrit une nouvelle classe fille de `Figure` ?

⇒ La méthode devient fausse.

Opérateur instanceof : discussion (2/2)

Bonne façon de résoudre ce genre de problème

Utiliser une méthode abstract dans la classe mère pour imposer aux filles d'écrire le code de la méthode

- dans Figure :

```
1 public abstract String getTypeFigure();
```

- dans Point :

```
2 public String getTypeFigure() { return "Point"; }
```

- dans Rectangle :

```
3 public String getTypeFigure() { return "Rectangle"; }
```

- Code générique (éventuellement en dehors des classes) :

```
4 public static void afficheType(Figure f) {
5     System.out.println("C'est un "+f.getTypeFigure());
6 }
```


Méthode getClass()

Méthode getClass()

- Méthode de la classe Object
- S'utilise sur une instance (syntaxe différente de instanceof)
- Retourne la classe de l'instance

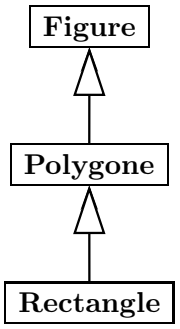
```
1 Figure f = new Segment(p1, p2);
2 System.out.println("f est de type : "+f.getClass());
3 // Affiche :  f est de type : class Segment
```

Usage classique pour comparer le type de deux instances :

```
1 // soit deux Objets obj1 et obj2
2 if (obj1.getClass() != obj2.getClass())
3   ...
```


Comparaison getClass() et instanceof

△ Ne pas confondre getClass() et instanceof



- instanceof est un **opérateur** qui indique si l'objet est de type **classe MaClasse ou ses descendantes**

```
Figure f1 = new Rectangle(); // subsomption
```

  - f1 instanceof Rectangle // true
  - f1 instanceof Polygone // true
  - f1 instanceof Figure // true

- getClass() est une **méthode** qui retourne la **classe exacte** de l'objet

```
Figure figR1 = new Rectangle();
Figure figR2 = new Rectangle();
Figure figP = new Point();

figR1.getClass() == figR2.getClass() // true
figR1.getClass() == figP.getClass() // false
```


Héritage et cast

Cast = 2 modes de fonctionnement

- Conversion sur les types basiques : le codage des données change. Souvent implicite dans votre codage...

```
1 double d = 1.4;
2 int i = (int) d;  // i=1
```

- Conversion dans les hiérarchies de classes :

```
1 Figure fig = new Point(); // subsomption
2 Point p1 = fig;  // Erreur compilation
3 Point p2 = (Point) fig;  // OK
```

△ le cast ne modifie pas l'objet

Le cast convertit la référence à un objet de type classe A en une référence à un objet de type classe descendante de A.

Caster c'est un peut comme-ci le programmeur disait au compilateur : *"je sais que normalement je n'ai pas le droit de faire cette affectation, mais fait moi confiance"* ... cependant le programmeur peut se tromper...

Cast : limite

- Cast inutile : subsomption

```
1 Point p1 = new Point();
2 Figure fig1 = (Figure)p1; // OK mais cast inutile
3 Figure fig2 = p1; // OK subsomption
```

- Cast obligatoire : comment **revenir** à un `Point` ?

```
4 Point p2 = fig2; // Erreur compilation
5 Point p3 = (Point)fig2; //OK compilation cast obligatoire
6                         //OK exécution l'objet est un point
```

- Mauvais cast

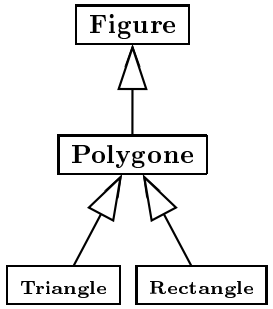
```
7 Figure fig3 = new Segment();
8 Point p4 = (Point)fig3; // compilation OK (!)
```

Exécution : Crash du programme avec le message suivant

```
Exception in thread "main" java.lang.ClassCastException:
Segment cannot be cast to Point
```

A l'exécution, l'objet qui est un `Segment`, ne peut pas être affecter à une variable de type `Point`, car un segment n'est pas un point (comparer avec le cas à la ligne 5 où l'objet est un point).

Cast : avec plusieurs niveaux de hiérarchie



Exemples de cas possibles quand on a 3 niveaux d'héritage

```
1  Figure fig1 = new Rectangle();
2  Polygone poly1 = new Rectangle();
3
4  Rectangle rect1 = (Rectangle) fig1; // OK
5  Rectangle rect2 = (Rectangle) poly1; // OK
6
7  Polygone poly2 = (Polygone) fig1; // OK
8
9  Figure fig2 = poly2 ;  // OK subsomption
10
11 Rectangle rect3=(Rectangle) fig2;  // OK
```

Exemples de cas impossibles :

```
1 Figure fig1 = new Rectangle();
2 Triangle tri1=(Triangle) fig1; // OK compilation
3 // ERREUR execution ClassCastException : Rectangle cannot be cast to Triangle
4
5 Triangle tri2=new Triangle();
6 Rectangle rect4=(Rectangle)tri2; // ERREUR compilation !!!
7 // incompatible types: Triangle cannot be converted to Rectangle
```


Cast : sécurisation

Idée

Vérifier le type de l'instance avant la conversion

En général, pour éviter les erreurs de cast à l'exécution, il est préférable de tester le type de l'instance avant de caster.

```
1         Figure f = new Segment();
2         f.methodeDeSegment(); // ERREUR compilation
3         Segment s;
4         if(f instanceof Segment) { // bon usage de instanceof
5             s = (Segment) f;
6             s.methodeDeSegment(); // OK
7         }
```

⇒ vous utiliserez systématiquement cette sécurisation

Méthode equals : égalité entre objets

Comparer deux objets

Pour comparer deux objets, il faut utiliser la méthode `equals` de la classe `Object` qui a pour signature :

`boolean equals(Object obj)`

- Par défaut, `equals` teste l'égalité référentielle, ce qui n'est pas intéressant...

```
1 Point p1=new Point(1,2);
2 Point p2=new Point(1,2);
3 Point p3=p1;
4 p1.equals(p2); // false : pas le même objet
5 p1.equals(p3); // true : même référence (même objet)
```

- Solution : redéfinir la méthode dans la classe fille pour tester l'égalité des variables d'instances
  - △ Il faut que la signature soit **exactement la même**, en particulier il faut que le paramètre soit de type `Object`
  - △ Si le paramètre n'est pas de type `Object`, ce n'est pas de la redéfinition, mais de la surchage (à éviter pour `equals`) !!!

Méthode equals

Exemple pour la classe Point.
Cette méthode peut en particulier réaliser les tests suivants :

1. Vérifier s'il y a égalité référentielle

```
1     public boolean equals(Object obj) {
2         if (this == obj) return true; // optionnel
```

2. Vérifier le type de l'object référencé par le paramètre obj

```
3         if (obj == null) return false;
4         if (getClass() != obj.getClass()) return false;
```

3. Caster la référence de l'object obj pour atteindre ses attributs

```
5         Point other = (Point) obj;
```

4. Vérifier l'égalité entre attributs

```
6         if (x != other.x) return false;
7         if (y != other.y) return false;
8         return true;
9     }
```


Méthode equals

Méthode equals

Il y a toujours un cast (sécurisé) dans equals pour pouvoir accéder aux attributs à comparer

Exemple sur la classe Point :

```
1     public boolean equals(Object obj) { // V1
2         if (this == obj)
3             return true;
4         if (obj == null)
5             return false;
6         if (getClass() != obj.getClass())
7             return false;
8         Point other = (Point) obj;
9         if (x != other.x)
10            return false;
11        if (y != other.y)
12            return false;
13        return true;
14    }
```


Méthode equals : getClass *vs* instanceof

Imaginons la redéfinition suivante de equals utilisant instanceof au lieu de getClass() :

```
1     public boolean equals(Object obj) {// V2
2         if (this == obj)
3             return true;
4         if ( !(obj instanceof Point))
5             return false;
6         Point other = (Point) obj;
7         if (x != other.x)
8             return false;
9         if (y != other.y)
10            return false;
11        return true;
12    }
```

Quelles sont les limites de l'implémentation v2 ?

Méthode equals : getClass *vs* instanceof

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

3 if(p.equals(pn))
4             System.out.println("ils sont égaux !!! ");
5 else
6             System.out.println("ils ne sont pas égaux !!! ");
```

- V1 : pas égaux
- V2 : égaux : est-ce légitime ?

Si on redéfinit equals dans PointNomme (il faut qu'elle appelle la méthode equals de sa classe mère) alors :

- p.equals(pn) appelle la méthode equals de Point
  - ♦ V2 : égaux (pn instance de Point)
- pn.equals(p) appelle la méthode equals de PointNomme
  - ♦ V2 : pas égaux (p n'est pas instance de PointNomme)

△ Problème de symétrie : p.equals(pn) ≠ pn.equals(p)

Méthode equals

- La méthode equals est dans Object ⇒ Tous les objets sont comparables entre eux !
- On peut donc rechercher un objet en particulier dans un tableau (ou une ArrayList) exploitant le polymorphisme :

```
1 ArrayList<Object> al = new ArrayList<Object>();
2 al.add( "toto");
3 al.add(10);
4 al.add(new Point(1,2));
5 al.add(new PointNomme("A",3,4);
```

- Sachant que la méthode contains de la classe ArrayList utilise equals. Quelle est la valeur de :

```
al.contains(new Point(1,2))
```

  - si la méthode equals n'a pas été redéfinit dans Point ?
    - **false** car la méthode equals de Object test l'égalité référentielle
  - si la méthode equals a été redéfinit dans Point ?
    - **true** car la méthode equals de Point test l'égalité structurelle

Copie d'objets : méthode clone()

Solution 1 : constructeur de copie (cours 3)

Solution 2 : méthode standard clone()

Méthode standard de la classe Object dont l'objectif est de retourner un nouvel objet qui est une copie du l'objet courant

- 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 Point p1 = new Point(1,2);
2 Point p2 = p1.clone();
```

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

Méthode clone() et redéfinition

Rappel : il est possible d'augmenter la visibilité d'une méthode dans la classe fille mais pas de la réduire

- dans Object : la méthode est protected

```
protected Object clone(){...}
```

- dans Point

```
// pour éviter les cast : le type de retour est Point
public Point clone(){return new Point(...);}
```

- dans PointNomme

```
public PointNomme clone(){return new PointNomme(...);}
```

```
1 Point [] tab={new Point(1,2),new PointNomme("A",3,4)};
2 Point [] tabCopie=new Point[tab.length];
3 for(int i=0;i<tab.length;i++)
4    tabCopie[i]=tab[i].clone();
```

Remarque : quand il y a de l'héritage, on ne peut pas faire cela avec des constructeurs de copie

Plan du cours

Rappel : final pour les attributs

Idée : protéger ses objets... Et ses programmes

Initialiser les valeurs des attributs sans pouvoir les modifier ensuite
Exemple : `String`

```java
public class Point{
  public final double x,y;
  public Point(double x, double y){
    this.x = x; this.y = y;
  }
  // interdiction de modifier x, y dans la suite
  // (et chez le client)
}
```

- Interdiction de modifier `x` et `y` dans les méthodes (pas de setter, pas de translation...)
- Modification d'un `Point` = création d'une nouvelle instance
- Possibilité de laisser les attributs `public`... Puisque non modifiable
- Sécurité lorsqu'un objet est passé en argument de méthode

Rappel : les constantes

Idée

Pour sécuriser le code, interdisons les modifications de certaines valeurs (notamment les constantes).

- final = sécurisation = impossibilité de modifier
- static = indépendante des instances
- *Usage :* une constante est définie en majuscule

```java
public class MaClasse{
  public final static int MA_CONSTANTE = 10;
  ...
```

Usage :

- constantes *universelles* (`Color.RED`, `Color.YELLOW`, `Math.PI`, `Double.POSITIVE_INFINITY...`)
- typologie (type de codage d'un pixel, organisation du BorderLayout)...
- bornes algorithmiques (`NB_ITER_MAX`, `TAILLE_MAX`...)

final : usages liés à l'héritage

- Méthode final : ne peut pas être redéfinie dans les classes filles

```java
public class Point {
    ...
    public final double getX(){ ... }
}

public class PointNomme extends Point{
    ...
    // Compilation impossible : méthode existante final
    public double getX(){ ... }
}
```

- Classe final : ne peut pas être étendue
  (par exemple : String, Integer, Double...)

```java
public final class Point{
    ...
}

// Compilation impossible : classe "mère" final
public class PointNomme extends Point{
    ...
}
```