L'héritage en Programmation Orientée Objet

Marc Champesme
mailto:Marc.Champesme@lipn.univ-paris13.fr

5 janvier 2021

Objectif

- L'héritage est une caractéristique essentielle de la POO
- Pas de POO sans héritage : présent dans tous les LOO
- L'objectif principal est la réutilisation du code : comment utiliser le même code (sans duplication) pour plusieurs classes ?

Qu'est-ce que l'héritage ?

- Partager du code entre des classes possédant une relation particulière : la relation *"est un"*
- Exemple : un carré *"est un"* rectangle (dont tous les côtés sont égaux)
- Cette relation *"est un"* traduit l'existence de caractéristiques ou comportements communs entre classes
- En POO on utilise les termes *"est sous-classe de"* et *"est super-classe de"* pour exprimer cette relation *"est un"* entre classes
- La classe `Carre` *"est sous-classe de"* la classe `Rectangle`
- La classe `Rectangle` *"est super-classe de"* la classe `Carre`

Exemple : logiciel de dessin

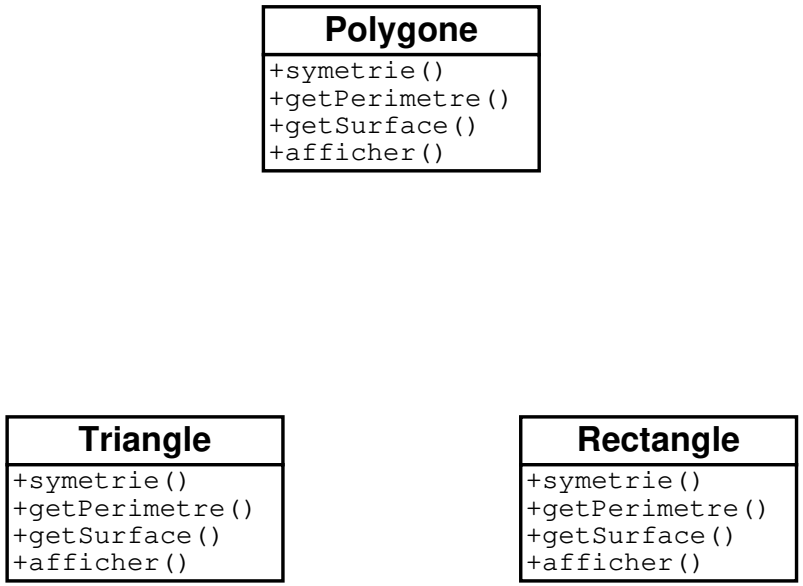

FIGURE – Sans héritage

Exemple : logiciel de dessin

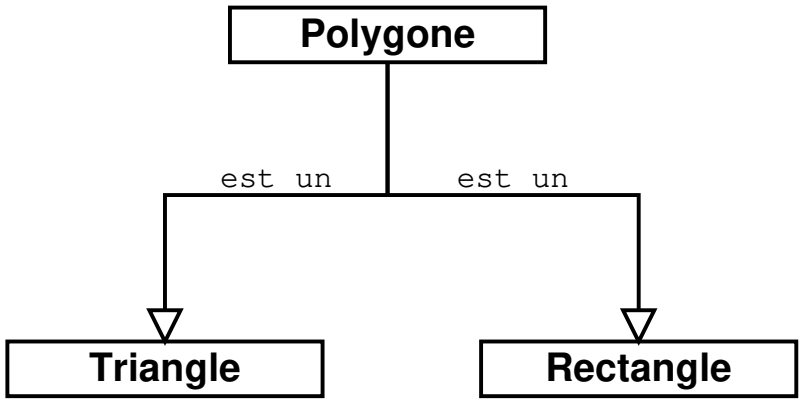

FIGURE – Relation d'héritage

Exemple : logiciel de dessin

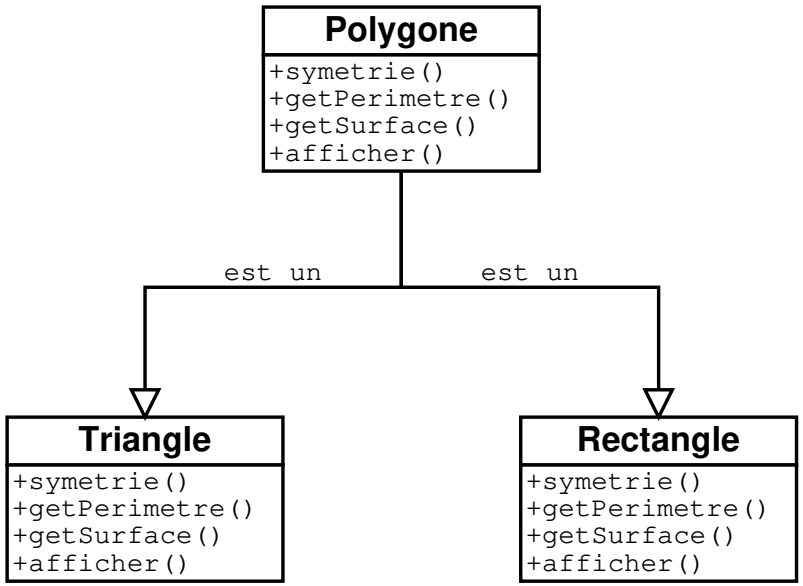

FIGURE – Fonctionnalités héritées

Quelques règles :

- La sous-classe hérite de toutes les méthodes et attributs de sa super-classe
- La sous-classe hérite du contrat de sa super-classe

Quelques règles :

- La sous-classe hérite de toutes les méthodes et attributs de sa super-classe
- La sous-classe hérite du contrat de sa super-classe
- Les méthodes héritées peuvent être redéfinies (adaptation à la sous-classe)

Quelques règles :

- La sous-classe hérite de toutes les méthodes et attributs de sa super-classe
- La sous-classe hérite du contrat de sa super-classe
- Les méthodes héritées peuvent être redéfinies (adaptation à la sous-classe)
- Les sous-classes peuvent définir de nouvelles méthodes. . .

Quelques règles :

- La sous-classe hérite de toutes les méthodes et attributs de sa super-classe
- La sous-classe hérite du contrat de sa super-classe
- Les méthodes héritées peuvent être redéfinies (adaptation à la sous-classe)
- Les sous-classes peuvent définir de nouvelles méthodes. . .
- . . . et de nouveaux attributs

Quelques règles :

- La sous-classe hérite de toutes les méthodes et attributs de sa super-classe
- La sous-classe hérite du contrat de sa super-classe
- Les méthodes héritées peuvent être redéfinies (adaptation à la sous-classe)
- Les sous-classes peuvent définir de nouvelles méthodes. . .
- . . . et de nouveaux attributs
- MAIS : les attributs ne peuvent pas être redéfinis. Chaque attribut défini s'ajoute aux attributs des super-classes (même s'ils portent le même nom et sont de même type).

Important

ATTENTION !

La sous classe hérite de TOUS les attributs de la super-classe

Important !

ATTENTION !

La sous classe hérite de TOUS les attributs de la super-classe

ATTENTION ! !

Même les attributs `private` de la super-classe

ATTENTION ! ! !

OUI : Même les attributs `private` de la super-classe et les attributs `private` des super-classes de la super-classe !

Exemple

```
public class Polygone {
        private int nbSommets;
        ...
}
public class Rectangle extends Polygone {
        private int nbSommets = 4;
        // Rectangle possède 2 attributs nbSommets !!!
        ...
}
```

Utilisation de extends

```
public class Polygone {
  ...
}
public class Rectangle extends Polygone {
  ...
}
public class Carre extends Rectangle {
  ...
}
```

Redéfinition

```
public class Polygone {
    ...
    public double getPerimetre() {
     ...
    }
}
public class Rectangle extends Polygone {
     ...
    @Override
    public double getPerimetre() {
     ...
    }
}
```

Redéfinition (suite)

```
public class Polygone {
    ...
    public double getPerimetre() {
     ...
    }
}
public class Rectangle extends Polygone {
     ...
    @Override
    public double getPerimetre() {
        ...
        double sup = super.getPerimetre();
        ...
    }
}
```

Constructeurs et héritage (1)

ATTENTION

Les constructeurs NE sont PAS hérités ! !

Constructeurs

Chaque sous-classe doit définir son ou ses propres constructeurs pour initialiser ses propres attributs après appel d'un constructeur de la super-classe.

Constructeurs et héritage (2)

Utilisation de super(...)

Pour appeler un constructeur de la super-classe on utilise `super(...)` en première instruction du constructeur. C'est le nombre de paramètres (et leurs types) de `super(...)` qui permet de désigner le constructeur de la super-classe à exécuter.

Pas d'appel super(...) ?

Dans ce cas c'est le constructeur sans paramètre (s'il existe) de la super-classe qui est appelé. Il y a un `super()` implicite en première instruction du constructeur.

Constructeurs et héritage (3)

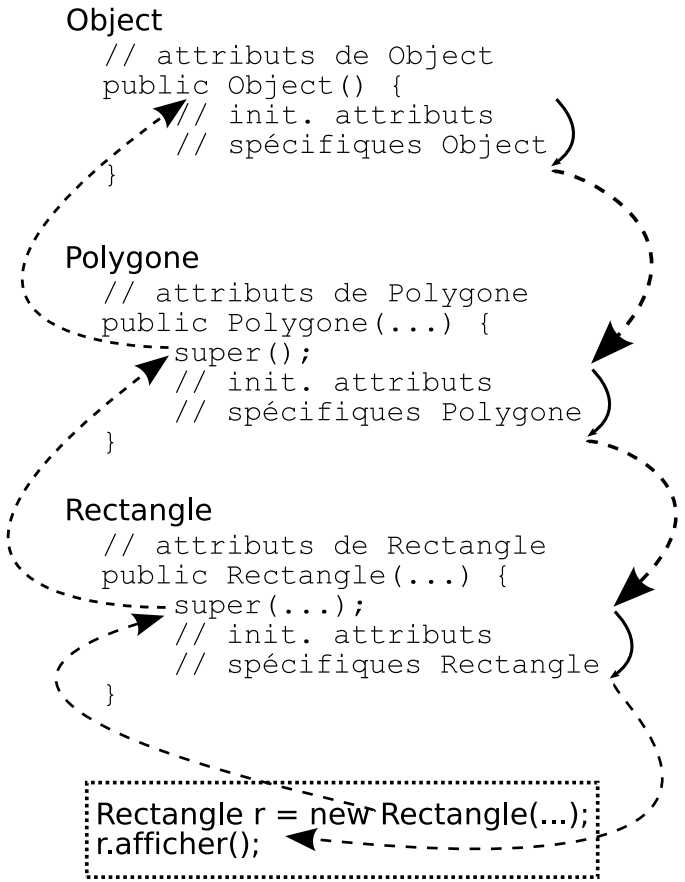

FIGURE – Chaînage des appels de constructeurs

Polymorphisme

C'est la possibilité offerte à une instance d'une classe de se faire passer pour une instance d'une autre classe.

```
Polygone poly1, poly2, poly3;
double surfaceTotale;
poly1 = new Polygone(...);
poly2 = new Rectangle(...);
poly3 = new Triangle(...);
surfaceTotale = poly1.getSurface()
              + poly2.getSurface()
              + poly3.getSurface();
```

Polymorphisme : comment ça marche ?

Quelles méthodes peuvent être appelées ?

```
double largeur;
Polygone poly = new Rectangle(...);
// la classe Rectangle définit une méthode getLargeur()
largeur = poly.getLargeur();
```

Definition (Typage statique)

C'est le type de la variable qui détermine les méthodes qu'il est possible d'appeler.

Erreur de compilation !

getLargeur() n'est pas définie dans Polygone, donc elle ne peut pas être appelée ici.

Polymorphisme : comment ça marche ?

Quelle version de la méthode est exécutée ?

```
double surface;
Polygone poly = new Rectangle(...);
// getSurface() est redéfinie dans Rectangle
surface = poly.getSurface();
```

Definition (Liaison dynamique)

C'est le type (= classe) de l'instance contenue dans la variable qui détermine quelle version de la méthode est exécutée.

Quelle version de getSurface() ?

Le type de l'instance est Rectangle donc c'est la version redéfinie dans Rectangle qui est exécutée.

Polymorphisme : comment ça marche ?

Definition (Statique)

La décision est prise au moment de la compilation (on ne connaît pas le type de l'instance contenue dans la variable).

Definition (Dynamique)

La décision est prise au moment de l'exécution (on connaît le type de l'instance contenue dans la variable).

Polymorphisme : comment ça marche ?

Definition (Statique)

La décision est prise au moment de la compilation (on ne connaît pas le type de l'instance contenue dans la variable).

Definition (Dynamique)

La décision est prise au moment de l'exécution (on connaît le type de l'instance contenue dans la variable).

Polymorphisme : pourquoi ça marche ?

```
double surface;
Polygone poly = new ???(...);
surface = poly.getSurface();
```

Que calcule getSurface() ?

L'héritage du contrat garantie que getSurface() calcule ce qui est indiqué dans le commentaire/contrat de getSurface() dans Polygone.

Attention !

Seul le programmeur peut garantir que le contrat hérité est respecté. Ici le compilateur ne peut pas aider.

Héritage du contrat : invariant

Héritage de l'invariant

Les invariants de toutes les super-classe se cumulent. Inutile donc de répéter les invariants des super-classes dans les sous-classes.

```
    * @invariant getNbCotes() > 2;
    * @invariant getNbSommets() == getNbCotes();
    */
public class Polygone { ... }

    * @invariant getNbCotes() == 4;
    * @invariant getLongueur() > 0;
    * @invariant getLargueur() > 0;
    */
public class Rectangle extends Polygone { ... }
```

Héritage du contrat : pré et post-conditions

Héritage des pré et post-conditions

Les couples pré et post-conditions de toutes les versions d'une méthode se cumulent.

```
// Classe A
    * @requires PA;
    * @ensures QA;
    */
public void meth(...) { ... }

// Classe B extends A
    * @requires PB;
    * @ensures QB;
    */
public void meth(...) { ... }
```

Héritage du contrat : pré et post-conditions (suite)

Contrat de la méthode meth de la classe A

Si PA alors QA

Contrat de la méthode meth redéfinie dans la classe B

- Si PA alors QA
- Mais aussi : Si PB alors QB

Attention aux contradictions !

$A : (x > 0) \Rightarrow (result > 0)$ vs $B : (x > 5) \Rightarrow (result < 0)$
PA $\Rightarrow$ QA ne doit pas être contradictoire avec PB $\Rightarrow$ QB

Avec le polymorphisme...

```
public class Dessin {
    private List<Polygone> listePoly;
    ...
    public void ajouterPolygone(Polygone p) {
        listePoly.add(p);
    }
    public void afficher() {
        for (Polynome p : listePoly)
            p.afficher();
    }
    public void symetrie() {
        for (Polynome p : listePoly)
            p.symetrie();
    }
}
```

Avec le polymorphisme... (suite)

```
Dessin monDessin = new Dessin(...);
monDessin.ajouterPolygone(new Polygone(...));
monDessin.ajouterPolygone(new Rectangle(...));
monDessin.ajouterPolygone(new Rectangle(...));
monDessin.ajouterPolygone(new Triangle(...));
monDessin.symetrie();
monDessin.afficher();
```

Mais aussi (nouvelle classe `HexagoneRegulier` héritant de `Polygone`)...

```
monDessin.ajouterPolygone(new HexagoneRegulier(...));
monDessin.afficher();
```

... sans changer la classe `Dessin` !

Plus loin avec le polymorphisme...

Et si on voulait ajouter des cercles, des ellipses... au dessin ?

- On peut définir de nouvelles classes `Cercle` et `Ellipse`
- On peut implémenter `afficher()` et `symetrie()` pour `Cercle` et `Ellipse`
- Mais `Cercle` et `Ellipse` ne sont pas des `Polygone`
- Définir une super-classe commune à `Cercle`, `Ellipse` et `Polygone` ? Une nouvelle classe `Figure` ? Puis remplacer Polygone par Figure dans `Dessin` ?

Plus loin avec le polymorphisme (2)

- On remplace :

```
public class Dessin {
    private List<Polygone> listePoly;
    ...
    public void ajouterPolygone(Polygone p) { ... }
```

Plus loin avec le polymorphisme (2)

- On remplace :

```
public class Dessin {
    private List<Polygone> listePoly;
    ...
    public void ajouterPolygone(Polygone p) { ... }
```

- par :

```
public class Dessin {
    private List<Figure> listeFig;
    ...
    public void ajouterFigure(Figure f) { ... }
```

Plus loin avec le polymorphisme (2)

- On remplace :

```
public class Dessin {
    private List<Polygone> listePoly;
    ...
    public void ajouterPolygone(Polygone p) { ... }
```

- par :

```
public class Dessin {
    private List<Figure> listeFig;
    ...
    public void ajouterFigure(Figure f) { ... }
```

- Et on pourrait faire :

```
monDessin.ajouterFigure(new Rectangle(...));
monDessin.ajouterFigure(new Cercle(...));
monDessin.afficher();
```

Plus loin avec le polymorphisme (3)

- Mais comment implémenter afficher() et symetrie() dans `Figure` ? vraiment compliqué ! !
- Mais est-ce vraiment nécessaire ?
- As t'on besoin de créer des instances de `Figure` ?
- Nous avons seulement besoin que l'instance présente dans une variable de type Figure ait accès à une implémentation de afficher() et symetrie()

Classes abstraites

Une classe abstraite est une classe :

- Dont on ne peut pas créer d'instance
- Qui définit un contrat, des entêtes de méthodes. . .
- . . . mais pas d'obligation de donner une implémentation aux méthodes (i.e. méthodes abstraites)
- Dont les sous-classes non abstraites (i.e. concrètes) ont obligation de donner une implémentation aux méthodes abstraites de la super-classe abstraite

Classes abstraites en Java

```
public abstract class Figure {
    private String nom;
    public Figure(String nom) {
        this.nom = nom;
    }
    public String getNom() {
         return nom;
    }
    public abstract void afficher();
    public abstract void symetrie();
    public abstract double getPerimetre();
    public abstract double getSurface();
}
```

Mais ne pas oublier le contrat (commentaires + assertions) ! !

Et l'héritage multiple ?

Supposons définie une classe `Losange` (i.e. quadrilatère avec quatre côtés égaux)

```
public class Losange extends Quadrilatere {
    ...
}
public class Rectangle extends Quadrilatere {
    ...
}
```

Alors pourquoi pas :

```
public class Carre extends Losange, Rectangle {
    ...
}
```

Héritage multiple : nous avons un problème !

On pourrait avoir :

```
public class Losange extends Quadrilatere {
    public double getPerimetre() { ... }
}
public class Rectangle extends Quadrilatere {
    public double getPerimetre() { ... }
}
```

Mais, si `Carre` ne redéfinit pas getPerimetre() :

```
Carre unCarre = new Carre(...);
double p = unCarre.getPerimetre(); // Quel code exécuter???
```

Héritage multiple : quel est le problème ?

- Problème : conflit d'héritage entre plusieurs implémentations, laquelle choisir ?
- Solutions : règles de priorité (C++), hériter plusieurs implémentations en renommant (EIFFEL), ...
- Problème de la solution : complexe à gérer et à utiliser !
- Autre solution (Java) : empêcher l'héritage d'implémentations concurrentes !
- Moyen : interfaces Java = classes abstraites sans aucune implémentation

Interfaces Java

Une interface définie uniquement un contrat

```
public interface Figure {
    // Pas d'attributs
    // Pas de constructeurs
    public void afficher(); // Toutes les méthodes sont
    public void symetrie(); // abstraites donc abstract inutile
    public double getPerimetre();
    public double getSurface();
}
```

Interfaces Java : mode d'emploi

```
// Une seule classe (concrète ou abtraite) par extends
public class ClasseConcrete extends AutreClasse
             implements UneInterface, AutreInterface {
             // Nombre illimité d'interfaces par implements
    ...
}
```

Exemple :

```
public class LinkedList<E>
        extends AbstractSequentialList<E>
        implements Deque<E>, Cloneable, Serializable {
    ...
}
```

Interfaces Java : recommandations

Il est plus facile de définir une classe par héritage d'une interface que par héritage d'une classe abstraite ou concrète, par conséquent :

- Utiliser de préférence les interfaces pour typer les arguments d'une méthode/constructeur ou la valeur de retour :

```
public List<String> concat(List<String> l) {...}
```

- plutôt que :

```
public ArrayList<String> concat(ArrayList<String> l) {...}
```

- Définir une interface en complément d'une classe abstraite :

```
public interface List<E> implements Collection<E> { ... }
public abstract AbstractList<E> implements List<E> { ... }
```

Exemple d'utilisation des interfaces et classes abstraites

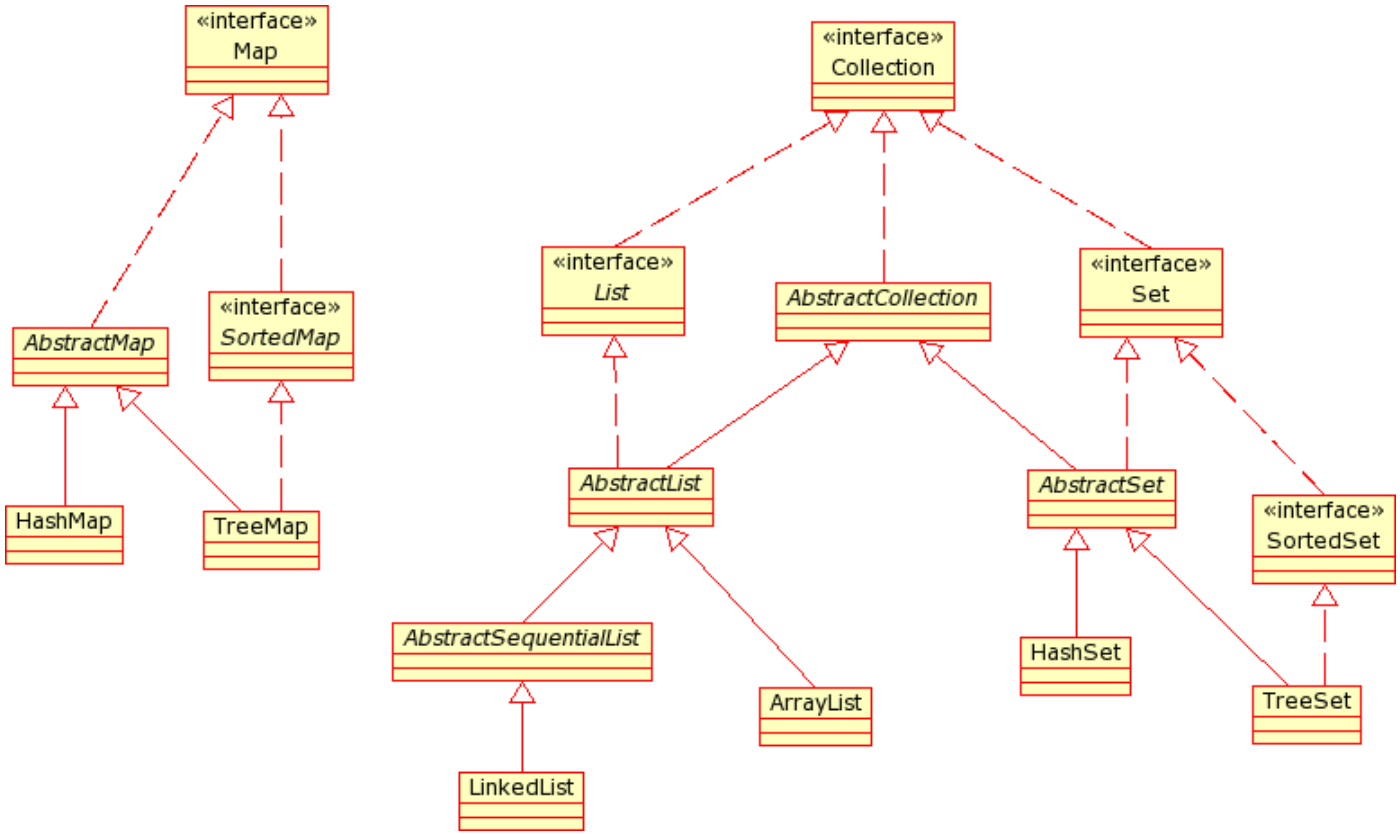

FIGURE – Hiérarchie des classes Collection et Map