


Concept. et prog. objet

TD n°1 : héritage

Détails

Écrit par stéphane Domas

Catégorie : M3105 - Concept. et prog. objet avancée (/index.php/menu-cours-s3/menu-mmi3-test)

 Publication : 21 octobre 2014

 Affichages : 53

1°/ Héritage et construction

1.1°/ Pourquoi hériter et de qui ?

- L'héritage est **généralement** considéré comme un moyen de modéliser un cas particulier d'une classe, sans avoir besoin de réécrire totalement le code.
- En effet, si une classe B hérite d'une classe A, c'est **généralement** parce que B constitue un cas particulier de A et que la plupart des attributs et méthodes de A sont également valables pour B.
- Cependant, cette généralité peut être contredite par les besoins applicatifs ou le souci "d'optimiser" le code.
- Par exemple, considérons les classes Carré et Rectangle. En géométrie, le carré est un cas particulier du rectangle. On est donc tenté de créer Rectangle comme super-classe et Carré comme sous-classe et c'est souvent la meilleure solution à prendre.
- Pourtant, il est parfaitement possible de faire l'inverse à **juste titre** !
- En effet, les opérations liées à un carré reposent sur une seule variable : la largeur du côté, alors que celles du rectangle reposent sur deux : largeur ET longueur. Si Carré hérite de Rectangle, alors il va hériter de longueur, ce qui est totalement inutile. Cela gâche de la place en mémoire.
- Si à l'inverse Rectangle hérite de Carré, on hérite de l'attribut largeur et on définit le nouvel attribut longueur. Le problème de cette solution est que la plupart des fonctions pour un carré ne sont pas valables sur un rectangle, justement à cause du second attribut. On est donc obligé de **redéfinir** la plupart des méthodes qui sont héritées de Carré (cf. section 2). Dans ce cas, autant faire 2 classes séparées !

1.2°/ Limiter l'héritage

- En POO, on peut définir des membres comme public, protégés ou privés.
- L'héritage ne concerne que les membres publics ou protégés.
- En effet, une sous classe n'a pas accès aux membres privés de la super-classe. Par exemple :

```
1 class A {  
2     protected int val;  
3     private void setVal() { val = 5; }  
4 }  
5 class B extends A { ... }  
6  
7 B b = new B();  
8 b.val = 8; // OK  
9 b.setVal(); // ERREUR COMPILATION
```

Remarque : il peut paraître étrange qu'une sous-classe n'ait pas un accès total aux membres de la super-classe. En pratique, même si le cas est rare, il existe, notamment quand les sous-classes n'ont pas besoin d'hériter de tous les attributs de la super-classe. Par exemple, on peut définir la classe Rectangle dont l'attribut largeur est protected et l'attribut longueur privé. Ainsi, si la classe Carré hérite de Rectangle, elle n'hérite que de largeur : elle n'a besoin que d'un seul attribut sur les deux.

1.3°/ Principes de construction

- En POO, l'héritage a une énorme importance dans la façon de construire un objet. Il faut en gros penser à un oignon et ses couches successives.

- Prenons une classe C qui hérite de B, qui hérite de A. Il y a donc 3 couches. On a coutume de représenter la chose avec A la couche la plus interne et C la plus externe (faire un petit dessin)
- Pour construire un objet de la classe C, il faut construire ces trois couches, sinon on obtiendrait un oignon creux, ce qui n'est pas accepté en POO (ni par la nature d'ailleurs).
- Pour caricaturer, construire un objet de classe C revient à partir de la couche la plus interne donc construire un objet de classe A, puis de construire par dessus la couche de B et enfin celle de C.
- **ATTENTION !** Cette image est parfois trompeuse car elle laisse penser qu'à chaque couche, on ajoute des choses, donc des attributs et/ou des méthodes. Ce n'est pas forcément le cas, notamment à cause du principe de redéfinition de membres (cf. section 2).
- En POO, construire un objet revient à appeler une des méthodes constructeur de la classe à instancier.
- D'après les remarques précédentes, cela implique que le constructeur de C doit appeler un des constructeurs de B qui appelle un de ceux de A.
- Comme on doit absolument construire un objet de classe A en premier, il faut que la première instruction des constructeurs de B soit d'appeler un de ceux de A.
- De même, les constructeurs de C doivent d'abord appeler un de ceux de B.
- En Java, on appelle un constructeur de la super-classe avec le mot-clé `super`. Par exemple :

```

1  class A {
2      protected int i;
3      protected double d;
4      public A() { i=0; d=0; }
5      public A(int i) { this.i = i; d = 0; }
6      public A(double d) { this.d = d; i = 0; }
7      public A(int i, double d) { this.i = i; this.d = d; }
8      ...
9  }
10
11 class B extends A {
12     ...
13     public B() { super(-1,-1); ... }
14     public B(double d) { super(d); ... }
15     ...
16 }
17
18 class C extends B {
19     ...
20     public C() { super(3.14); ... }
21     ...
22 }

```

Dans l'exemple ci-dessus, on voit que la création d'une instance de C va conduire à initialiser les attributs hérités i et d à 0 et 3.14

1.4°/ Surcharge (= overload en anglais)

- Dans l'exemple précédent, on remarque qu'il existe plusieurs constructeurs pour A et B, chacun ayant un prototype (= signature, entête, ... différents noms sont équivalents) différent.
- Cette possibilité s'appelle la **surcharge de méthode** et se retrouve dans tous les langages objets (et d'autres).
- La surcharge consiste donc à définir plusieurs fois la même méthode **dans une même classe** mais avec des **types** de paramètres (entrée et/ou sortie) différents.

Attention : le nom des paramètres n'a aucune importance, seul le type compte. Par exemple, la classe suivante provoque une erreur de compilation car il y a deux méthodes qui ont le même prototype (les 2 premières) :

```

1 | class A {
2 |     public int toto(int i, double d) { ... }
3 |     public int toto(int val, double frac) { ... }
4 |     public int toto(double min, int moy) { ... }
5 | }

```

- La surcharge peut s'appliquer à n'importe quel type de méthode, normale, constructeur, abstraite, ...

Attention : la surcharge ne doit pas être confondue avec la redéfinition abordée ci-dessous.

2°/ Redéfinition (= override en anglais)

La redéfinition permet de modifier le "comportement" d'un membre hérité. Les objectifs sont différents selon que le membre est un attribut ou une méthode.

2.1°/ Redéfinition d'attribut

- Pour une raison de modélisation, il se peut que l'on désire "arrêter" l'héritage d'un attribut de la super-classe pour ne pas le rendre accessible aux sous-classes.
- Dans ce cas, on le redéfinit dans la sous-classe où doit s'arrêter la transmission en le spécifiant comme privé.
- Par exemple :

```

1 | class A {
2 |     protected int val;
3 |     ...
4 | }
5 | class B extends A {
6 |     private int val; // redéfinition de val
7 |     ...
8 | }
9 | class C extends B {
10 |     // C n'a plus accès à val
11 | }

```

- **ATTENTION !** Cela implique que la classe B possède son propre attribut val qui porte le même nom que l'attribut val de A dont elle hérite. Elle peut donc potentiellement manipuler deux attributs portant le même nom.
- Pour éviter la confusion entre les deux, l'attribut de B se manipule en utilisant `val`, et celui hérité de A en utilisant `super.val`.
- Par exemple :

```

1  class A {
2      protected int val;
3      public A(int val) {this.val = val;}
4      public void printVal() {
5          System.out.println(val);
6      }
7      ...
8  }
9  class B extends A {
10     private int val;
11     public B(int val) {
12         super(val-1); // init. de l'attribut hérité
13         this.val = val; // init. de l'attribut local
14     }
15     public void printSuperVal() {
16         System.out.println(super.val);
17     }
18 }
19 ...
20 B b = new B(10);
21 b.printVal(); // affiche 10
22 b.printSuperVal(); // affiche 9

```

ATTENTION ! ce n'est pas parce que B hérite de la méthode printVal() qu'elle va afficher l'attribut val hérité. C'est le code qui prime : comme il est écrit val, alors c'est l'attribut local à B qui est manipulé.

Remarque : on voit bien que cette situation est confuse et peut engendrer de nombreuses erreurs de codage. Elle doit donc être évitée au maximum.

2.2°/ Redéfinition de méthode

- Contrairement à la redéfinition d'attribut, celle de méthode est très fréquente.
- On l'utilise par exemple quand un traitement fait dans la super-classe n'est pas optimal/efficace dans la sous-classe. Par exemple :

```

1  class Rectangle {
2      protected double largeur;
3      protected double longueur;
4      ...
5      double perimetre() {
6          return 2*(largeur+longueur);
7      }
8  }
9  class Carre extends Rectangle {
10     ...
11     public double perimetre() {
12         return 4*largeur;
13     }
14     ...
15 }

```

- La différence n'est pas fondamentale car il n'y a qu'une seule opération arithmétique au lieu de deux. Néanmoins, elle illustre le fait que pour un Carré, on a une méthode plus adaptée pour calculer le périmètre que si on utilise celle de Rectangle.

ATTENTION ! Pour redéfinir une méthode, il faut qu'elle ait exactement le même prototype (type retour + nom + type/ordre paramètres)

ATTENTION ! la surcharge est une notion différente. Par exemple, le fait d'écrire plusieurs constructeurs ayant des paramètres différents est une surcharge, donc pas une redéfinition.

3°/ Héritage multiple et interfaces.

3.1°/ L'héritage multiple

- En Java, on ne peut hériter que d'une seule classe. Ce n'est pas le cas en C++ où l'on peut hériter de plusieurs classes.
- Néanmoins, l'héritage multiple est source de problèmes et il vaut mieux l'éviter si on ne maîtrise pas totalement le sujet.
- Par exemple, prenons une classe A avec une méthode toto(). Soient B et C qui héritent de A mais qui redéfinissent toto(). Soit enfin D, qui hérite de B et C (= modèle en losange). Dans ce cas, D hérite de deux méthodes toto(). Laquelle est la bonne ? Comment le compilateur sait laquelle appeler ? Bien entendu, le C++ propose des solutions pour régler ce type de problème, mais il y en a d'autres, plus subtils qui apparaissent lorsque l'on fait appel au polymorphisme.
- Le problème est que la modélisation nous pousse parfois vers des solutions à héritage multiple.
- Par exemple, prenons les polygones. Si on veut calculer le périmètre d'un polygone régulier, la méthode est la même qu'il soit convexe ou croisé.
- Si on fait des sous-classes RégulierCroisé et RégulierConvexe, elles ont donc le même code pour la méthode périmètre.
- En POO où le but est de réutiliser un maximum le code : copier/coller une méthode est mauvais.
- Pour régler le problème, il suffirait de faire de l'héritage multiple. Par exemple, on crée une classe Régulier qui contient la méthode perimetre() adéquate, et la classe RégulierCroisé hérite aussi bien de Croisé que de Régulier.
- En Java, la question ne se pose pas puisque l'héritage multiple est impossible.
- Il existe cependant un simili héritage multiple sous la forme de classes spéciales appelées interface.

3.2°/ Les interfaces

- Une interface est une classe qui se contente de déclarer des méthodes. La classe n'a donc **aucun attribut ni de code**.
- Une classe Java n'hérite pas d'une interface. On dit qu'elle l'implémente. Cela vient du fait qu'elle **doit** définir le code de toutes les méthodes déclarées dans l'interface.
- On voit également qu'une interface est une "coquille vide" donc on peut se poser la question de son intérêt.
- En fait, une interface permet de déclarer des fonctionnalités communes à des classes peu importe comment ces classes les définissent. Ces fonctionnalités peuvent donc être appelées par d'autres classes pour faire des traitements qui sont indépendants du type des objets manipulés.
- Le meilleur exemple est le tri : on peut trier n'importe quel type d'objet pourvu que l'on puisse comparer 2 objets et dire lequel est avant/après l'autre. Cette comparaison est donc une fonctionnalité commune à tous les objets triables. On définit donc une interface déclarant cette fonctionnalité. Par exemple :

```
1 | interface Triable {
2 |     public int comparer(Triable t); // renvoie valeur <0 si this < t, 0 si this = t, et >0 si
3 | }
```

- Prenons maintenant la classe Banane. Si l'on veut comparer des bananes, il suffit que les bananes soient triables, donc implémentent l'interface Triable, à savoir définir la méthode comparer().
- Cette méthode va utiliser un ou plusieurs attributs de Banane (par ex. sa longueur) pour calculer la valeur à renvoyer. Cela donne :

```

1  class Banane implements Triable {
2      protected int longueur;
3
4      public Homme(int longueur) { this.longueur = longueur }
5
6      public int comparer(Triable t) {
7          if (t instanceof Banane) {
8              Banane b = (Banane)t;
9              return (longueur - b.longueur);
10         }
11         return 0;
12     }
13 }

```

- On peut maintenant définir une méthode de tri qui opère sur des objet Triable, peut importe l'instance réelle de ces objets. Par exemple :

```

1  public void trier(List<Triable> list) {
2      boolean swap = true;
3      int lim = list.size();
4      Triable t1,t2;
5      while (swap == true) {
6          swap = false;
7          for(int i=0;i<lim-1;i++) {
8              t1 = list.get(i); t2 = list.get(i+1);
9              if (t1.comparer(t2) > 0) {
10                 list.set(i,t2); list.set(i+1,t1);
11                 swap = true;
12             }
13         }
14         lim -= 1;
15     }
16 }

```

- Grâce à cela, on peut créer une liste de Banane et les trier, sans pour autant faire une méthode de tri spécifique à la classe Banane.
- C'est d'autant plus pratique que les méthodes de tris sont déjà écrites pour les collections et qu'il existe une interface similaire à Triable qui s'appelle Comparable qui déclare une méthode compareTo().
- Si on veut créer des objets que l'on doit pouvoir trier, c'est donc une bonne idée que les classes de ces objets implémentent l'interface Comparable.