Chapitre 6

Héritage en Java

1. Généralités

L'héritage est le troisième des paradigmes de la programmation orientée objet (le 1er étant l'encapsulation, le 2e la structure de classe).

L'emploi de l'héritage conduit à un style de programmation par raffinements successifs et permet une programmation incrémentielle effective.

L'héritage peut être simple ou multiple.

Il représente la relation: EST-UN

Exemple:

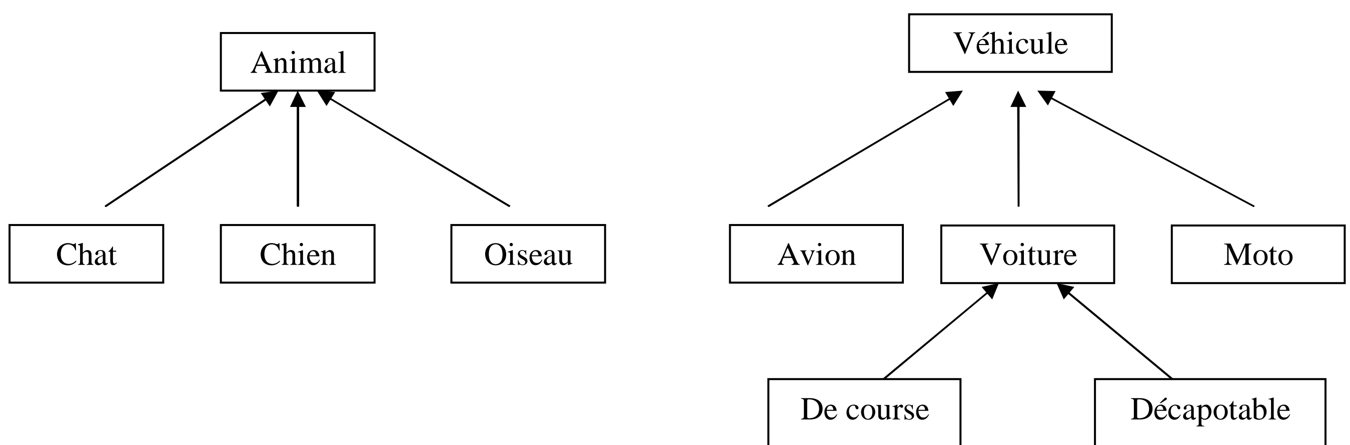
Un chat est un animal
Une moto est un véhicule
Un cercle est une forme

Alors que l'objet membre représente la relation: A-UN

Une voiture a un moteur

L'héritage est mis en œuvre par la construction de classes dérivées.

Le graphe de l'héritage est comme suit:



2. Hiérarchisation

- la classe dont on dérive est dite CLASSE DE BASE :

Animal est la classe de base (classe supérieure),

- les classes obtenues par dérivation sont dites CLASSES DÉRIVÉES :

Chat, Chien et Oiseau sont des classes dérivées (sous-classes).

3. Intérêt

- vision descendante => la possibilité de reprendre intégralement tout ce qui a déjà été fait et de pouvoir l'enrichir.

- vision ascendante => la possibilité de regrouper en un seul endroit ce qui est commun à plusieurs.

4. Utilisation

- vers le haut (en analyse O.O.) => on regroupe dans une classe ce qui est commun à plusieurs classes. Dans la classe Véhicule, on regroupe les caractéristiques communes aux Camions et aux Automobiles.

- vers le bas (lors de la réutilisabilité) => la classe de base étant définie, on peut la reprendre intégralement pour construire la classe dérivée. La classe Véhicule étant définie, on peut la reprendre intégralement, pour construire la classe Bicyclette.

5. Classe dérivée

Une classe dérivée modélise un cas particulier de la classe de base, et est enrichie d'informations supplémentaires.

La classe dérivée possède les propriétés suivantes:

- contient les données membres de la classe de base,

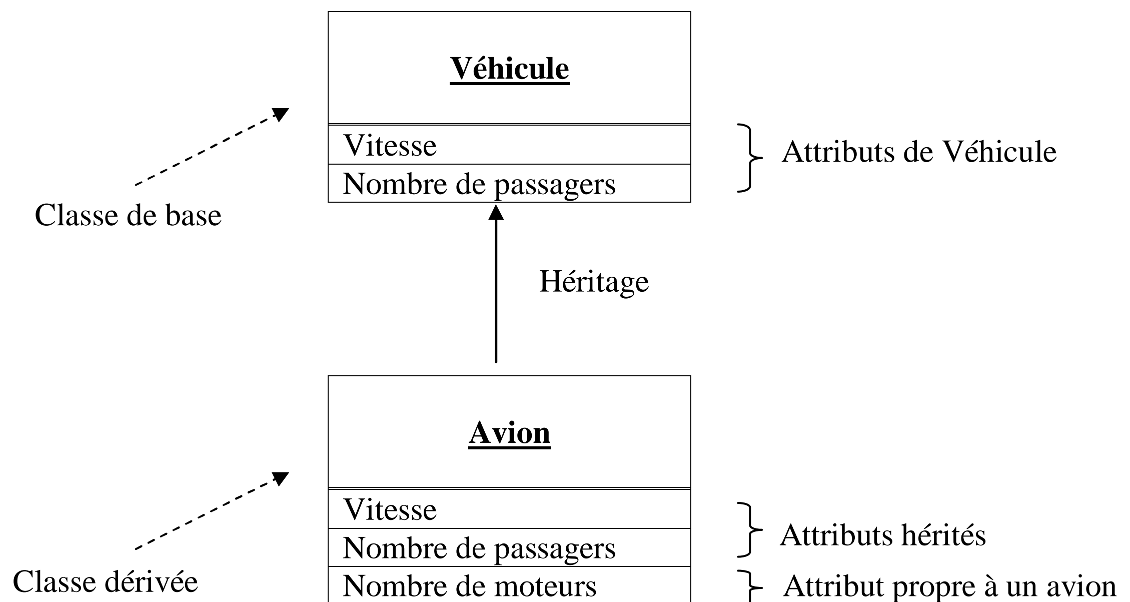
- peut en posséder de nouvelles,

- possède (à priori) les méthodes de sa classe de base,

- peut redéfinir (masquer) certaines méthodes,

- peut posséder de nouvelles méthodes.

La classe dérivée hérite des membres de la classe de base.



6. Syntaxe de l'héritage

Protection class classe_dérivée **extends** classe_de_base {/* etc. */}

Protection: droits d'accès attribués au niveau de la classe. **public** ou bien **néant** (i.e. rien).

Voir exemple : herex1.java

7. Constructeur

- La classe dérivée doit prendre en charge la construction de la classe de base.

Pour construire un `Avion`, il faut construire d'abord un `Véhicule`;

Le constructeur de la classe de base (`Véhicule`) est donc appelé **avant** le constructeur de la classe dérivée (`Avion`).

Si un constructeur de la classe dérivée appelle explicitement un constructeur de la classe de base, cet appel doit être obligatoirement la première instruction de constructeur. Il doit utiliser pour cela, le mot clé **super**.

```
class A {

      int i;

      // Constructeur de la classe A
      public A(int x){
            i = x;
      }
}

// Héritage

class B extends A {
      double z;

      // Constructeur de la classe B
      public B(int f, double w) {

            // Appel explicite du constructeur de A,
            // et en 1ère ligne.

            super(f);

            z = w;
      }
}
```

- Lors de la construction de la classe dérivée, nous devons tenir compte de la présence du constructeur dans la classe de base et/ou dans la classe dérivée.

-1- La classe de base et la classe dérivée ont au moins un constructeur public, c’est le cas général. De ce fait, le constructeur de la classe dérivée doit appeler le constructeur de la classe de base disponible.

-2- La classe de base n’a aucun constructeur. La classe dérivée peut ne pas appeler explicitement le constructeur de la classe de base. Si elle le fait, elle ne peut appeler que le constructeur par défaut, vu que c’est le seul qui est disponible dans ce cas dans la classe de base.

-3- La classe dérivée ne possède aucun constructeur. Dans ce cas, la classe de base doit avoir un constructeur public sans argument (par défaut ou un explicite).

Voir exemple : herex2.java

8. Droits d'accès

Les droits d'accès protègent les données et les méthodes, et réalisent aussi l'encapsulation.

Les droits d'accès sont accordés aux fonctions membres, ou aux fonctions globales.

L'unité de protection est la classe: tous les objets de la classe bénéficient de la même protection.

- un membre **public** est accessible à toutes les classes,
- un membre **«rien»** est accessible à toutes les classes du même paquetage,
- un membre **private** n'est accessible qu'aux fonctions membre de la classe.

Si les membres de la classe de base sont :

- public ou « rien » : les membres de la classe dérivée auront accès à ces membres (champs et méthodes),

- private : les membres de la classe dérivée n'auront pas accès aux membres privés de la classe de base.

En plus des 3 niveaux de protection (public, « rien » et private), il existe un 4e niveau de protection : **protected**. Un membre de la classe de base déclaré protected est accessible à ses classes dérivées ainsi qu’aux classes du même paquetage.

9. Phases d'initialisation d'un objet de la classe de base et de la classe dérivée

Le tableau suivant représente la création d'un objet à partir des classes de base (A) et dérivée (B).

Un objet de la classe de base A	Un objet b de la classe dérivée B
allocation mémoire pour un objet du type A	allocation mémoire pour un objet du type B (donc A+B)
initialisation par défaut des champs	initialisation par défaut des champs (A+B)
initialisation explicite des champs	initialisation explicite des champs hérités de A
exécution des instructions du constructeur de A	exécution des instructions du constructeur de A
	initialisation explicite des champs hérités de B
	exécution des instructions du constructeur de B

10. Redéfinition et surdéfinition des fonctions membres

Dans l'exemple herex1.java, la fonction `affiche` est membre de la classe de base `véhicule`. Elle n'affiche que les membres privés de cette classe. On ne peut donc pas afficher par exemple le nombre de moteurs.

Pour faire cela, nous allons définir dans la classe dérivée une fonction portant le même nom, et qui aura pour rôle d'afficher les données privées de la classe dérivée. On parle alors de redéfinition d'une fonction de la classe de base. Cette nouvelle méthode va masquer la présence de la méthode héritée.

Voir exemple : herred.java

Surdéfinir[1] une méthode consiste à lui donner plusieurs significations. Nous choisirons la bonne signification en fonction du contexte dans lequel cette méthode sera utilisée.

Voir exemple : hersurd.java

Nous pouvons nous servir aussi de la surdéfinition et de la surcharge au même temps.

Voir exemple : herredsurd.java

[1] Ou bien surcharger une méthode.

En résumé :

Surdéfinition	Redéfinition
Consiste à cumuler plusieurs méthodes ayant le même nom.	Consiste à substituer une méthode par une autre.
même nom, mais signature différente et peu importe le type de retour.	même nom, même signature, même type de retour.

Contraintes sur la redéfinition

a- signature identique :

```
class A {
     public void unefonction(int x) { //etc.}
}
class B extends A {
     public void unefonction(int z) { //etc.}
}
```

b- droits d'accès : la redéfinition ne doit pas diminuer les droits d'accès à une méthode.

```
class A {
     private void unefonction(int x) { //etc.}
}
class B extends A {
     public void unefonction(int z) { //etc.}
}
```

11. Typage statique vs. typage dynamique

- type statique d'une variable: type à la compilation, type déclaré.

- type dynamique d'une variable: type à l'exécution, type en mémoire.

12. Compatibilité entre objets d'une classe de base et objets d'une classe dérivée

Un objet d'une classe dérivée peut toujours être utilisé au lieu d'un objet de sa classe de base.

Par exemple, un `avion` est un `véhicule`. Mais l'inverse n'est pas vrai, un `véhicule` n'est pas nécessairement un `avion`.

Soit l'exemple suivant:

```
vehicule v = new vehicule(300.,4);

avion a = new avion(3,800.,350);
```

1er cas: `v = a`

Conversion implicite de tout `avion` EST-UN `véhicule`. Le compilateur fait une copie en ignorant les membres excédentaires (`nbre_moteurs`).

Supposez que tous les membres des classes base et dérivée ont été déclarés `public`, nous aurons ce qui suit:

```
v.vitesse = 500.6; // => ok.

// Erreur car véhicule n'a pas d'information sur le nombre de moteurs.
v.nbre_moteurs = 3;
```

2e cas: `a = v; // erreur`

Un `véhicule` n'est pas forcément un `avion`. On ne peut pas deviner quelles seront les valeurs manquantes (dans cet exemple: `nbre_moteurs`). Un `véhicule` n'a pas toutes les données d'un `avion`.

Pour que ça marche, il faut "caster" (forcer le changement de type)

```
// Ok à la compilation mais pas à l'exécution
a = (avion) v;
```

Voir exemple : herconv.java

13. Exemple de Balounes[2]

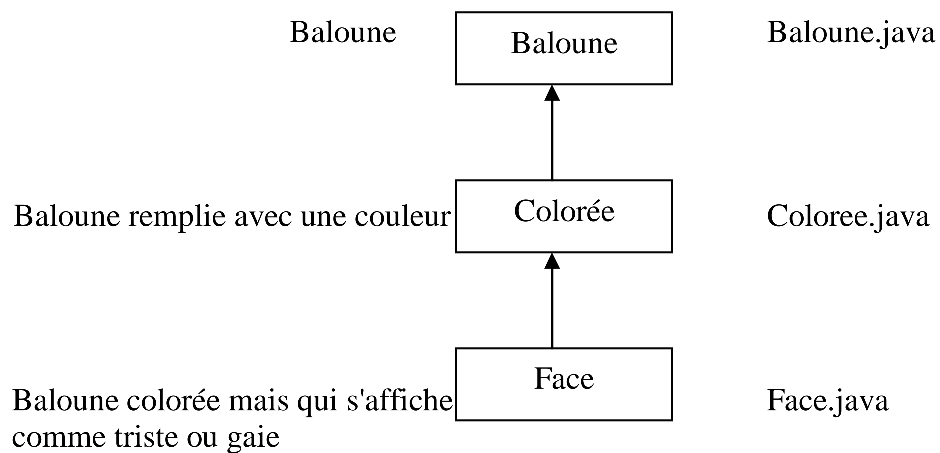
Exemple -1- : Balounes.html

L'applet est codée dans : Balounes.java

Exemple -2- : PetageDeBalounes.html

L'applet est codée dans : PetageDeBalounes.java

Les deux applets utilisent le schéma d'héritage suivant :



[2] Code source par Guy Lapalme.

14. Classes et méthodes finales

- Une méthode finale ne peut être redéfinie dans une classe dérivée.

- Une classe finale ne peut plus être dérivée.

15. Classes abstraites

a. Intérêt

- Placer dans une classe abstraite toutes les fonctionnalités que nous souhaitons disposer lors de la création des descendants.

- Une classe abstraite sert comme classe de base pour une dérivation.

b. C'est quoi au juste?

- Une classe abstraite ne permet pas l'instanciation des objets.

- Une classe abstraite peut contenir des méthodes et des champs (qui peuvent être hérités) et une ou plusieurs méthodes abstraites.

- une méthode abstraite est une méthode dont la signature et le type de la valeur de retour sont fournis dans la classe et rien d'autre (pas le corps de la méthode i.e. définition).

c. Syntaxe

```
abstract class A { // etc.}
```

d. Particularités

- Une classe ayant une méthode abstraite est par défaut abstraite. Donc pas besoin de "abstract" à ce stade.

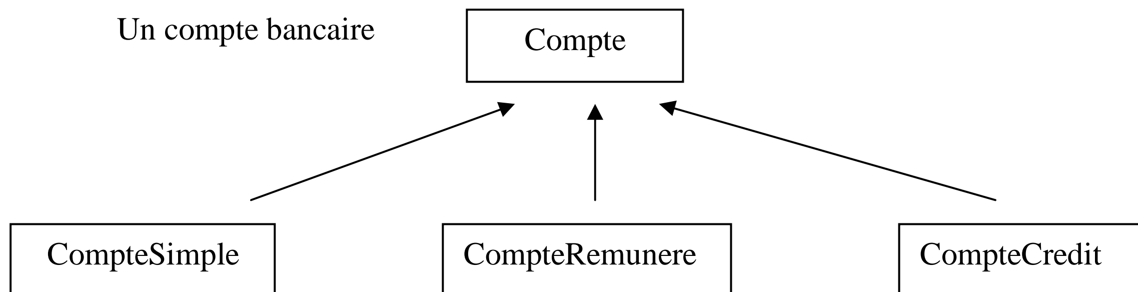
- Les classes abstraites doivent être déclarées "public" sinon pas d'héritage!

- Le nom d'argument muet doit figurer même s'il est sans intérêt!

- Une classe dérivée d'une classe abstraite n'est pas obligée de redéfinir toutes les méthodes abstraites. Elle peut même n'en définir aucune. Si c'est le cas, elle reste abstraite.

- Une classe dérivée d'une classe non abstraite peut être déclarée abstraite.

Voir exemple : herabst.java



un compte simple	un compte rémunéré	un compte crédit
Il n'autorise pas le débit on redéfinit donc l'opération de retrait	on redéfinit donc la méthode de mise à jour	le comportement de ce compte est ici celui de la classe mère, on ne le modifie pas.

Nous avons utilisé la méthode abstraite abstract `public void maj()` pour la mise à jour du compte.

15. Interfaces

Un complément d'informations sur le lien entre interface, classe abstraite et héritage.

- L'interface n'est qu'une classe abstraite particulière.

- L'interface est une notion indépendante de l'héritage et donc de la classe dérivée. Une classe dérivée peut implémenter une ou plusieurs interfaces.

- Une interface peut dérive d'une autre interface en utilisant `extends`, réservé pour l'héritage. En réalité, l'opération a simplement permis de concaténer les déclarations. Les droits d'accès ne sont pas pris en compte ce qui n'est pas le cas lors des héritages des classes.