Programmation Orientée Objet Java

Chapitre 2 : L'héritage

Omar EL MIDAOUI

Héritage

- Il existent plusieurs types de vélos.
  - Ex. : vtt, vélo route, vélo ville
- Ils ont plusieurs caractéristiques communes.
- Mais, certaines caractéristiques sont propres à un type spécifique.
  - Ex. : un vélo ville a un garde bout
- Il est possible de créer une **sous-classe** qui hérite l'état et le comportement d'une **superclasse**.
- Cela permet de réutiliser le code de la superclasse.

Hiérarchie de classes

```
class VeloRoute extends Velo
{
// Nouvelles variables et méthodes ...
}
```

Héritage

- Définir une nouvelle classe à partir d'une autre ⇒ Dérivation
  - Nouvelle classe : **classe dérivée**
  - Classe originale : **superclasse** ou classe de base.
- On définit une nouvelle classe en utilisant une autre en utilisant le mot **extends**

```
class Personne {
Membres de la classe Personne... }
```

```
class Medecin extends Personne {
Membres de la classe Medecin.... }
```

- **Conséquence :**
  - Un objet de type Medecin contiendra :
    - Les membres hérités de la classe Personne
    - Les membres propres à Medecin (Membres spécifiques caractérisants un medecin).
  - Un objet Medecin est un objet spécialisée d'un objet Personne.

Héritage

- Bonne modélisation de la vie réelle.
- La dérivation d'une nouvelle classe d'une classe de base est un processus additif :
  - Les membres supplémentaires définissent ce qui fait que l'objet de la classe dérivée est différent de celui de la classe de base.
  - Tous les membres déclarés dans la nouvelle classe sont ajoutés à ceux qui sont déjà membres de la classe de base.

Héritage

- à l'exception de **la classe Object** (qui n'a pas de superclasse) toute classe possède une superclasse.
- En absence d'une superclasse explicite, toute classe crée est automatiquement une sous-classe de Object.

- Une classe peut dériver d'une classe qui dérive d'une autre classe et ainsi de suite, jusqu'à la superclasse Object.
- Une telle classe est dite **descendante** des toutes les classes dans la chaine d'héritage.

Héritage

- Attention : une sous-classe hérite tous les membres de sa superclasse.
- Les constructeurs ne sont pas des membres !
- Donc, ils ne sont pas hérités par la sous-classe, mais ils peuvent être évoqués par la sous-classe.

L'arbre d'héritage

Soit une classe B qui hérite d'une classe A.

On dira que A est la classe mère, ou super-classe, et B `la classe fille, ou sous-classe`.

- Toute instance de B est une instance de A.
- Toute instance de B posséde tous les membres de A plus les membres dèfinis dans B.
- Au niveau de la classe, tous les membres statiques de A sont des membres statiques de B (et B posséde en plus les membres statiques dèfinis dans B).
- On peut redéfinir dans B les mèthodes de A.

L'accès aux membres d'une classe

- Cela n'implique pas que tous les membres de données définis dans la classe de base sont accessibles aux méthodes spécifiques de la classe dérivée. Certains le sont d'autres non.
- Un **membre hérité** d'une classe de base est **accessible** au sein de la classe dérivée.
- S'il n'est pas accessible alors il ne s'agit pas d'un membre hérité.
- Toutefois les membres de classe qui ne sont pas hérités font quand même partie de l'objet de la classe dérivée.

Membres "private" dans la superclasse

- La sous-classe n'a pas accès aux membres **"private"** de la superclasse.
- Mais **les classes imbriquées** de la superclasse ont accès aux membres **private** de la superclasse.
- Donc, **les classes imbriquées** public et protected dans la superclasse (auxquelles la sous-classe a accès) fournissent une possibilité d'accès indirect aux membres private de la superclasse.

Membres "private" dans la superclasse

Exemple

```
class A {
private int x;
public InterneA obj = new InterneA();
protected class InterneA {
public int getX() { return x; } }
}
class B extends A {
public int getX() { return this.obj.getX(); } }
class C {
public static void main(String[] args) {
System.out.println((new B()).getX());
} }
```

L'accès aux membres d'une classe

Conclusion :

Une instance d'une sous-classe ne peut pas accéder directement aux membres privés de ses super-classes.

L'accès ne peut se faire que via des méthodes **public**, **protected** ou **package** suivant la localisation de la sous-classe.

Masquer des membres de données

- Membre de donnée peut avoir le même nom dans la classe de base et dérivée. (Pas recommandée)
- $\Rightarrow$ les membres de données de la classe de base sont hérités. mais masqués par les membres de la classe dérivée.
- Toute utilisation du nom du membre de la classe dérivée fera toujours référence au membre défini en tant que partie de la classe dérivée.
- Pour faire référence au membre de la classe héritée, vous devez la qualifier à l'aide du mot clé super.
- C'est pas possible d'utiliser super.super.quelqueschose.

Méthodes héritées

- Les méthodes déclarées comme **private** ne sont pas héritées.
- Celle que vous déclarez **sans attribut d'accès** ne sont pas héritées que si vous définissez la classe dérivée dans le même paquetage que la classe de base.
- Les autres méthodes sont toutes hérités.
- Les **constructeurs** sont différents des autres méthodes ordinaires. Les constructeurs d'une classe de base ne sont jamais hérités.

Exemple d'héritage (1/3)

Animal.java

```
package animals;
public class Animal {
private int poids;
public void dormir () { System.out.println ("Méthode dormir de
Animal"); }
public void jouer () { System.out.println ("Méthode jouer de
Animal");
public void seReproduire () { System.out.println ("Méthode
sereproduire de
Animal"); }
}
```

Exemple d'héritage (2/3)

Mammifere.java

```
public class Mammifere extends Animal
{
public void seReproduire () { System.out.println ("Méthode
seReproduire de
Mammifère");}
}
```

Chat.java

```
public class Chat extends Mammifere
{
public void jouer () { System.out.println ("Méthode jouer de
Chat");}
public void miauler () { System.out.println ("Méthode miauler de
Chat");}
}
```

Exemple d'héritage (3/3)

RunChat.java

```
public class RunChat {
public static void main ( String[] args) {
Chat minet = new Chat();
minet.dormir();
minet.seReproduire();
minet.jouer(); }
}
```

Exemple d'héritage (3/3)

RunChat.java

```
public class RunChat {
public static void main ( String[] args) {
Chat minet = new Chat();
minet.dormir();
minet.seReproduire();
minet.jouer(); }
}
```

L'exécution de RunChat donnera :

Méthode dormir de Animal
Méthode seReproduire de Mammifère
Méthode jouer de Chat

Objets d'une classe hérité

- Les **constructeurs de la classe** de base peuvent être appelé dans la classe dérivée même s'ils ne sont pas hérités.
- Même si vous n'appelez pas le constructeur de la classe de base à partir du constructeur de la classe dérivée, **le compilateur** essaiera de le faire pour vous.
- Puisqu'un objet de la classe dérivée englobe un objet de la classe de base, l'utilisation d'un constructeur de la classe de base constitue un bon moyen d'initialiser la partie de base d'un objet d'une classe dérivée.

Objets d'une classe hérité

Animal.java

```
public class Animal {
public Animal(String unType) {
type = new String(unType);
}
public String toString() {
return "il s'agit d'un "+type;
}
private String type;
}
```

Objets d'une classe hérité

Chien.java

```
public class Chien extends Animal {
public Chien(String unNom){
super("Chien"); //Appelle du constructeur de base
NomduChien = unNom;
RaceduChien = "Inconnu";
}
public Chien(String unNom, String uneRace) {
super("Chien"); //Appelle du constructeur de base
NomduChien = unNom;
RaceduChien = uneRace;
}
private String NomduChien;
private String RaceduChien;
}
```

super est utilisé pour faire référence au constructeur de la classe de base.

Héritage: mot clé "super"

- Il est possible d'accéder aux données/méthodes de la classe de base grâce au mot clé super.

Exemple 1 :

```
class Vtt extends Velo
{ private String suspension;
// Constructeur
public Vtt (double vitesse, String couleur,String suspension)
{
super (vitesse,couleur) ; // Appel du constructeur de Velo
// Si cet appel est utilisé, c'est toujours
// la première instruction du constructeur
this.suspension=suspension; }
... }
```

Redéfinition de méthodes

Redéfinir une méthode d'une classe de base

- Vous pouvez redéfinir une méthode de la classe de base dans une classe dérivée
- L'attribut d'accès de la méthode dans une classe dérivée peut être **identique** à celui de la classe de base, ou **moins restrictif**, mais il ne peut pas être plus restrictif
  - $\Rightarrow$ Ex. Toute méthode est déclarée comme `public` dans une classe de base doit être également comme `public` dans la classe dérivée.
  - $\Rightarrow$ vous ne pouvez pas omettre l'attribut d'accès de la classe dérivée ou le spécifier comme privée ou protected.

La redéfinition de méthodes

Impératif :On doit respecter la signature de la méthode qu'on redéfinit!

- ses paramètres (nombre, type, ordre)
- son type de retour
- son attribut d'accessibilité : on peut élargir son accès.

La redéfinition de méthodes

Impératif :On doit respecter la signature de la méthode qu'on redéfinit!

- ses paramètres (nombre, type, ordre)
- son type de retour
- son attribut d'accessibilité : on peut élargir son accès.

Une méthode `package` peut être redéfinie en une méthode `public`.

Une méthode `public` ne peut pas devenir `private` dans une sous-classe.

La redéfinition de méthodes

Exemple 2 :

```
class Oiseau {
...
void speak( ) {
System.out.println ("pépie") ; }
...
}
class Peroke extends Oiseau {
...
public void speak( ) {
super.speak( ) ;
System.out.println("Hello") ; }
...
}
```

Polymorphisme

Polymorphisme

- Le polymorphisme en Java est la capacité des sous-classes d'avoir leur propre comportement et, en même temps, de partager quelques fonctionnalités avec d'autres sous-classes.
- Exemple :

```
class Vtt extends Velo {
private String suspension;
...
void imprimeEtat() {
System.out.println("vitesse : " + vitesse);
System.out.println("suspension : " + suspension);
}
}
```

Polymorphisme

```
class DemoVelo {
public static void main(String[] args) {
Velo velo1 = new Velo(20);
Velo velo2 = new Vtt(20, "dual");
velo1.imprimeEtat();
velo2.imprimeEtat();
}
}
```

Résultat

vitesse : 20

vitesse : 20

suspension : dual

Polymorphisme

- La JVM évoque la méthode appropriée de l'objet référencé par chaque variable.
- Cela sappelle **virtual method invocation**, ou "late binding".

Méthodes, Variables et Classes "final"

- Le modificateur **final** est utilisé pour indiquer qu'une méthode ne peut pas être redéfinie ou surchargée.
- Ils sont utiles dans le cas des méthodes appelées par les constructeurs d'une classe. Si une sous-classe redéfinie une telle méthode, cela peut avoir des conséquences indésirables.
- Appliqué à une variable, celle ci ne peut pas être modifiée (constante).
- Une classe entière peut être déclarée comme "final". Cela veut dire que la classe ne peut pas avoir des sous-classes (par exemple, la classe String est final).

Graphe de la relation de typage : Conversions de type

supertype d'une classe B =
l'ensemble des classes "au-dessus" de B dans l'arborescence de la relation d'hèritage + l'ensemble des interfaces que B implèmente.

Toute instance de B est une instance de tous les èlèments de son supertype.

Pour les types objets comme pour les types primitifs, la conversion de type est possible si on respecte la règle d'èlargissement du domaine.

Conversions de type

A ab=new B();
Un objet **B** peut être affecté a une variable d'un type **A** si **A** fait partie du surpertype de **B**.

Personne p=new Etudiant();
Toute instance d'`Etudiant` est une instance de `Personne`, mais la rèciproque est fausse.

Par contre, dans la suite du code, seuls les mèthodes et attributs accessibles de **A** pourront etre invoquès sur **ab**.

Exemple

```
Personne etud = new Etudiant("Durand", "Paul",
"Licence Informatique", "Universite d'Artois"); // OK

System.out.println(etud.getFormation());
// refuse !!  erreur a la compilation!!
// getFormation() n'est pas une methode de Personne

Etudiant p = new Personne("Dupont","Jacques");
// !!  refuse // !!  erreur a la compilation
```

Le polymorphisme

Le polymorphisme :

a l'exècution, Java choisit la mèthode a interprèter en fonction du type de l'objet a qui est envoyè le message.

Exemple

```
// quelque part
public static void afficheToi(Personne p){
System.out.println(p);
}
// et plus loin...
Personne p1 = new Personne("Dupont","Jacques");
Personne p2 = new Etudiant("Durand", "Paul", "Licence
Informatique", "Universite d'Artois");
afficheToi(p1);
afficheToi(p2);
```

Exemple

provoquera l'affichage suivant :

```
Jacques Dupont
Paul Durand inscrit en Licence Informatique a
Universite d'Artois
```

Hèritage multiple?

Et si on veut crèer la classe des ètudiants-salariès?

```
public class EtudiantSalarie extends Etudiant,
Salarie  // !!  Refuse !!  Erreur a la compilation !!
```

Il n'y a pas d'héritage multiple en Java.
Une classe ne peut hériter que d'une seule classe.

Hèritage multiple?

L'hèritage multiple pose le problème du conflit de code : que faire lorsqu'on hèrite de deux classes qui proposent deux implèmentations diffèrentes d'une même mèthode?

Java refuse donc l'hèritage multiple.

Les **interfaces** permettent de lever cette limitation, car une classe peut implèmenter plusieurs interfaces.