Programmation Orientée Objet
M2 CCI 2008–2009
hoenen@icps.u-strasbg.fr

Université de Strasbourg
Département d'informatique
14 janvier 2009

TP3 : l'héritage

1 Héritage simple

Pourquoi hériter ? Lors de l'écriture d'une nouvelle classe, si il existe déjà une autre classe qui possède une partie des caractéristiques et fonctionnalités voulues, la nouvelle classe peut hériter de la classe déjà implantée. Cela permet de réutiliser des champs et des méthodes existantes sans avoir de les coder et de les debugger. On peut par conséquent se concentrer uniquement sur l'ajout des nouvelles fonctionnalités que ne possède pas la classe existante.

Comment hériter ? La programmation orientée objet permet à une classe `B` d'hériter d'une autre autre classe `A`. On appelle `B` une sous-classe (ou classe dérivée) de `A`, et on appelle `A` la super-classe de `B`. La définition de la classe `B` se fait alors par

```
class B extends A {
  ...
}
```

La sous-classe hérite de tous éléments (attributs, méthodes) qui sont déclarés `public` ou `protected` dans la super-classe, quelque soit le paquetage dans lequel se trouve cette super-classe. Les constructeurs ne sont pas hérités mais peuvent être invoqués depuis la sous-classe par `super(...);`. Attention : lorsque le constructeur de la suepr-classe est appelé dans le corps du constructeur de la sous-classe, cet appel doit se trouver sur la première ligne du nouveau constructeur.

Une seule classe est orpheline En Java, chaque classe possède une et une seule super-classe, à l'exception de la classe `Object` qui n'en possède aucune. L'héritage permet d'établir une hiérarchie des classes au somment de laquelle se trouve la classe `Object`, dont hérite toutes les classes pour lesquelles l'héritage n'est pas spécifié explicitement par le mot-clé `extends`.

Redéfinition de méthodes Toutes les classes héritent des méthodes définies dans `Object`, comme par exemple `public boolean equals(Object o)` ou `public String toString()`. Ces méthodes sont souvent redéfinies, par exemple pour afficher correctement l'état d'un objet d'une certaine classe. Pour cela, la méthode héritée doit à nouveau être définie au niveau de la sous-classe. Le super-classe peut interdire la possibilité d'effectuer cette redéfinition dans une sous-classe en déclarant des méthodes avec le modificateur `final`.

Dans la classe `Personne` du TP2, on avait déjà mis en place une première approche de la réutilisation de code : l'agrégation (une `Personne` *a une* `Adresse`). On va voir maintenant comment réutiliser cette classe avec l'héritage.

1. Les informations fournies par la classe `Personne` ne permettent pas d'en déterminer le sexe. Étendre cette classe (par exemple avec la classe `PersonneSexuée`) pour ajouter cette information sans modifier la classe d'origine.
2. On souhaite désormais gérer un arbre généalogique, il faut donc étendre cette classe pour y ajouter des informations sur la filiation (avec uniquement des informations simples : père, mère, mari–épouse, enfants). Écrire cette classe et toutes les méthodes qui en découlent.
3. Tester ces classes en construisant un petit arbre généalogique.

2 Polymorphisme

***Cast* d'objets** Soit une classe `B` qui hérite de la classe `A`, on peut utiliser une référence de type d'une super-classe vers un objet de type une sous-classe. Donc les instructions de gauche sont autorisées par le compilateur alors que l'instruction de droite provoque une erreur.

```
Object o = new B();
A a = new B();
```

```
B b = a;
  b = o;
```

Le *cast* se fait implicitement dans le premier cas car il n'y a aucun risque d'erreur. Dans le deuxième cas, il y a un risque que l'objet référencé par `a` ou `o` ne soit pas de type convenable. On peut forcer le compilateur à accepter ces instructions avec un *cast* explicite, mais le code pourra générer des erreurs à l'exécution. Il existe cependant un moyen de s'assurer du type d'un objet, grâce à l'opérateur `instanceof` :

```
if (a instanceof B) {
  B b = (B)a
}
```

On veut à présent modéliser le comportement d'animaux domestiques. En allant au plus simple, un animal domestique possède un nom et peut crier :

- un oiseau est une sorte d'animal qui peut apprendre la faculté de voler. Si il sait voler il criera *cui-cui*, sinon il est trop jeune et il criera *piou-piou*.
- un perroquet est une sorte d'oiseau qui peut apprendre la faculté de parler. Il dira alors son nom lorsque qu'il criera.
- un chien est un animal qui remue la queue uniquement quand il est content. Il fait *ouaf-ouaf* quand il est content, sinon il grogne.

1. Écrire une classe pour chaque type d'animal. Chercher les relations d'héritage entre les différentes classes pour réutiliser le code.
2. On a la classe suivante pour gérer des animaux dans un unique tableau.

```
public class Test {
  public static void main(String[] args) {
   Animal a = new Animal("l'animal");
   Oiseau o = new Oiseau("l'oiseau");
   Perroquet p = new Perroquet("coco");
   Animal[] tab = new Animal[6];
   tab[0] = a;
   tab[1] = o;
   tab[2] = p;
   tab[3] = new Oiseau("Birdy");
   tab[4] = new Perroquet("Roger");
   tab[5] = new Chien("Rex");
  }
}
```

   Ajouter une méthode `sonnette()` qui provoque, lorsqu'elle est appelée, le crie de chaque animaux du tableau.
3. Ajouter une méthode `apprentissage()` qui permet à un animal du tableau d'apprendre toutes ses capacités et tester à nouveau la sonnette avec des animaux ayant toutes les capacités et un chien en colère.

3 Interfaces et héritage

Interfaces Une *interface* permet de passer un *contrat* entre des programmeurs, pour s'assurer que des programmes et classes puissent communiquer de la même façon. En Java, une interface est un type référence (comme les classes) et ne peut contenir que des constantes et des signatures de méthodes. Une interface ne peut pas être instanciée, mais elle peut être implémentée (mot-clé `implements`) par une classe ou étendue (mot-clé `extends`) par une autre interface. Il est alors possible de considérer un objet d'une classe qui implémente une interface comme un objet du type de cette interface. L'utilisation d'interface permet de pallier d'une certaine manière l'absence d'héritage multiple en Java.

Le but de cet exercice est de comprendre comment apporter des améliorations à des programmes Java sans avoir à modifier toutes les classes et en gardant les compatibilités entre les types d'objets manipulés.

1. Construire une classe `Couple` qui possède :
   - deux attributs entiers,
   - une méthode `addition` qui prend en argument un autre couple et qui renvoie le couple égal à la somme des deux couples,
   - une redéfintion de la méthode toString pour décrire l'objet.

   Étendre cette classe en une classe `Triplet` qui fait de même mais pour trois entiers et en réutilisant au maximum les méthodes de `Couple`.
2. Définir une interface `Nuplet` qui assure que la classe qui l'implémente possède des méthodes `somme`, `produit` et `moyenne`. Ces méthodes calculent respectivement la somme, le produit et la moyenne des éléments d'un n-uplet. Construire les classes `CouplePlus` et `TripletPlus` qui étendent respectivement les classes `Couple` et `Triplet` et implémentent l'interface `Nuplet`.
3. Écrire une classe de test qui contient une méthode `sommeProduit` qui prend en paramètre un n-uplet et renvoie un couple dont le premier élément est égal à la somme des éléments du n-uplet, et le second élément est égal au produit des éléments du n-uplet.
4. On veut avoir la possibilité d'ordonner un ensemble de n-uplets selon la valeur moyenne de leur éléments. Écrire les classes `CoupleOrdre` et `TripletOrdre` qui contiennent une méthode `plusGrand` qui indique si le n-uplet est plus grand ou plus petit que celui passé en argument de la méthode. Il faudra réutiliser au maximum le code et ne pas modifier les classes `CouplePlus` et `TripletPlus`.