Travaux Dirigés de Java n°3

Maîtrise d'Informatique

— *Etienne Duris* - `Etienne.Duris@inria.fr` —

▸ **Exercice 1. (Surcharge *vs* Redéfinition)** *À partir de l'exercice de la feuille précédente où la classe* `Student` *hérite de la classe* `Person`*, rappellez vous ce que sont la surcharge et la redéfinition des membres d'une classe. Pour chacun des membres (attributs ou méthodes), quel est le processus de résolution utilisé par Java ?*

Définissez une classe `GrantHolder`*, qui hérite de* `Student`*, représentant un étudiant qui bénéficie d'une bourse. Dans cette classe, surchargez les attributs* `year` *(qui devient le nombre d'années depuis le début d'attribution de la bourse) et* `grant` *(qui devient un entier représentant le montant annuel en francs de la bourse).*

Définissez des constructeurs (un par défaut et un avec les paramètres nécessaires) qui utilisent explicitement les constructeurs de `Student` *et initialisent l'attribut* `grant` *de cette classe mère (mot clé* `super`*).*

Redéfinissez la méthode `toString` *pour la classe* `GrantHolder`*, et définissez dans cette classe deux méthodes surchargées* `setGrant`*: l'une qui ne prend pas de paramètre et qui positionne l'attribut* `grant` *(masqué) de* `Student` *à vrai* (1)*, et l'autre qui prend un entier en paramètre et l'affecte à l'attribut* `year` *de la classe* `GrantHolder`*.*

Quelles sont les visibilités de chacune des méthodes en fonction du type des objets sur lesquels on les appelle ? Si on redéfinit la méthode `getYear` *de* `Student` *pour qu'elle retourne le nombre d'années depuis le début d'attribution de la bourse dans* `GrantHolder`*, comment faire pour connaître le nombre d'années d'études d'un objet de type* `GrantHolder` *?*

▸ **Exercice 2. (Visibilité)**

Dans les classes Java suivantes, déterminez quelles sont les visibilités des membres de chaque classe dans chacune des autres, en fonction des modificateurs de visibilité et des appartenances aux différents paquetages.

```
/* Fichier Work.java */
package pack1.pack2;
class Work {
  private int v;
  protected double special;
  int myWorkingValue;
  WorkBis() {..}
  private void bidouille() {..}
```

(1) Je sais, c'est pas très réaliste mais bon ... c'est un exo :-)

```
  public void pourquoi() {..}
}
/* Fichier MyWork.java */
package pack1.pack2;
class MyWork extends Work {
  ...
}
/* Fichier Mother.java */
package pack1.pack2;
public class Mother {
  private float myFloat;
  protected int myInt;
  boolean myBool;
  public String official;
  public Mother() {..}
  private void convert() {..}
  protected void setTag() {..}
  void compute() {..}
  public void displayString() {..}
}
/* Fichier Daughter.java */
package pack1.pack3.pack4;
import pack1.pack2.Mother;
class Daughter extends Mother {
  protected boolean myBool;
  Daughter() {..}
  private void localConvert() {..}
  protected void convert() {..}
}
```

▸ **Exercice 3. (Interface)** *Définissez une interface* `Comestible` *qui déclare les méthodes* `sonGout()` *et* `sonNom()`*. Définissez l'interface* `Mangeable` *qui étend* `Comestible` *en déclarant une méthode* `commentManger()`*. Faire la même chose en définissant l'interface* `Buvable` *qui déclare la méthode* `commentBoire()`*. Déclarez une classe* `Orange` *qui implémente les interfaces* `Mangeable` *et* `Buvable`*. Définissez une classe* `Fruit`*, avec un attribut de type* `Comestible`*, initialisé par le constructeur, et qui possède une méthode permettant d'appeler* `commentBoire()` *et/ou* `commentManger()` *sur son attribut s'il est mangeable et/ou buvable. Que procure en plus le fait de faire implémenter l'interface* `Comestible` *par* `Fruit` *?*