TP1 : Héritage

Introduction

Le code écrit pour les exercices de ce TP devra être remis sur la page Moodle de ce module d'ici le début de la prochaine séance. Le code devra être clair et commenté. Vous devrez également rédiger un rapport contenant les réponses aux questions posées, ainsi que tout ce qui vous semblera pertinent : remarques sur votre code, stratégie d'implémentation, difficultés rencontrées, ...

Exercice 1 : Animaux

La classe `Animal` comporte deux sous-classes `Mammifere` et `Arachnide`. La classe `Mammifere` est elle-même dérivée par `Chien` et `Homme`, tandis que la classe `Arachnide` est dérivée par `Araignée` et `Scorpion`. Modélisez et créez ces différentes classes de telle sorte que la fonction suivante :

```
public static void main(String[] args)
{
   Animal[] animaux = new Animal[6] ;
   animaux[0] = new Chien("Ziggy") ;
   animaux[1] = new Homme("David") ;
   animaux[2] = new Araignee() ;
   animaux[3] = new Scorpion() ;
   animaux[4] = new Chien() ;
   animaux[5] = new Araignee("Bowie") ;
   for (int i = 0; i < 6; i++)
   {
      System.out.println(animaux[i].getPresentation()) ;
   }
}
```

affiche le résultat suivant :

```
Je suis un animal de nom Ziggy. Je suis un mammifere. Je suis un chien.
Je suis un animal de nom David. Je suis un mammifere. Je suis un homme.
Je suis un animal sans nom. Je suis un arachnide. Je suis une araignée.
Je suis un animal sans nom. Je suis un arachnide. Je suis un scorpion.
Je suis un animal sans nom. Je suis un mammifere. Je suis un chien.
Je suis un animal de nom Bowie. Je suis un arachnide. Je suis une araignée.
```

Questions

1. Un animal ne peux PAS être qu'un animal. Comment empêcher que la ligne suivante :

   ```
   Animal animal = new Animal() ;
   ```

   puisse compiler ?

Exercice 2 : Péage autoroutier

Un service autoroutier a installé un péage sur une de ses routes principales. Dans une première version, chaque péage applique la même tarification, à savoir :

- Les camions paient une taxe qui dépend du nombre d'essieux et de son poids total. La facturation est de 7 euros par essieu, plus 15 euros par tonne arrondi au supérieur.
- Les voitures paient une taxe fixe de 4 euros.

Pour cette version simplifiée,
- une voiture ne nécessite aucun attribut particulier de façon obligatoire ;
- un camion possède 2 attributs privés *nbreEssieu* et *poidsTotal* (exprimé en tonnes), dont les valeurs pourront être données par les accesseurs *getNbreEssieu()* et *getPoidsTotal()* ;
- un péage possède 2 attributs privés *nbreVehicule* et *totalCaisse* qui mémorisent le nombre de véhicules (camion ou voiture) ayant franchi le péage, et le montant total des taxes perçues. Ces deux variables sont mises à jour lors du passage d'un véhicule.

Pour réaliser cette application,
- définissez les classes `Peage`, `Voiture` et `Camion`. Il serait judicieux de définir une classe abstraite `Vehicule`, dont dériveraient `Voiture` et `Camion`.
- définissez une classe `Global` avec la méthode *main* permettant d'instancier des péages, des voitures et des camions, et de simuler les passages des véhicules aux péages.

Variantes (liste non exhaustive)
- Le coût est variable suivant les péages : la facturation par essieu et par tonne ou par voiture n'est plus identique pour tous les péages, mais spécifique à chacun d'entre eux.
- Des statistiques plus fines peuvent être proposées telles que remplacer *nbreVehicule* par deux attributs *nbreVoiture* et *nbreCamion*, pour compter le nombre de voitures et de camions ayant franchi le péage.
- Les motos ont une tarification différente.
- La voiture du chef a la propriété d'être une voiture, mais ne paye rien aux péages.

Exercice 3 : Lapins

Créez une classe `Lapin`. Tout lapin possède un attribut *vivant* permettant de savoir s'il est en vie, ainsi qu'une méthode publique *passeALaCasserole()* qui le fait mourir. Pour éviter toute surpopulation, il est impossible de créer un nouveau lapin s'il y a déjà plus de 50 lapins vivants.

1. Dans une première version, vous pouvez, en cas de surpopulation, créer un lapin sans vie (l'attribut *vivant* est initialisé à *false*).

   Il y a en fait 3 cas de figures : (1) un lapin est vivant, (2) un lapin était vivant à un moment donné, mais est passé à la casserole, et (3) un lapin n'a jamais pu être créé à cause de la surpopulation. Dans cette version, les situations (2) et (3) ne peuvent pas être différenciée.
2. Dans une seconde version, rendez le constructeur de `Lapin` privé, et créez une méthode publique *Lapin creationLapin()*, qui appelle le constructeur *Lapin()* s'il n'y pas de surpopulation, ou renvoie un pointeur *null* sinon. Dans ce cas, les 3 situations décrites ci-dessus peuvent être distinguées.

Questions
1. Comment compter à tout instant le nombre de lapins en vie ?
2. Pourquoi, dans la première version, ne peut-on pas simplement renvoyer *null* en cas de surpopulation ?

Exercice 4 : Résolution d'une équation du second degré

Réalisez un programme calculant les solutions d'une équation du second degré de la forme $ax^2 + bx + c = 0$. Pour cela, vous créerez 3 classes `RacinesSol0`, `RacinesSol1` et `RacinesSol2`, définissant les trois cas possibles (0, 1 ou 2 solutions). Ces trois classes dériveront d'une même classe-mère, `Racines`. Vous utiliserez également la classe `UsineARacines`, qui permet de créer la bonne classe RacinesSol à partir des cœfficients de l'équation. Vous testerez les différents cas dans une fonction *main*.

Le code de `Racines` et de `UsineARacines` est donné ci-dessous et disponible sous forme d'archive en ligne :

```
/** Classe de base de toutes mes Racines. */
public class Racines
{
   /** Les coefficients et le discriminant de l'equation. */
   private double m_a, m_b, m_c, m_dis ;
```

```
   /** Constructeur. */
   protected Racines(double a, double b, double c, double dis)
   {
      m_a = a; m_b = b; m_c = c; m_dis = dis ;
   }

   /** Calcule les racines de l'equation. */
   public void calculeRacines()
   {
      System.out.println("Erreur si ici ! ") ;
   }

   /** Renvoie le nombre de racines. */
   public int nbRacines()
   {
      System.out.println("Erreur si ici ! ") ;
      return 0 ;
   }

   /** Renvoie la valeur de la i-eme racine. */
   public double valeurRacine(int i)
   {
      System.out.println("Erreur si ici!") ;
      return 0.0 ;
   }

}
```

```
/** Createur de racines. */
public class UsineARacines
{
   /** Creation d'une Racines en fonction des coefficients. */
   static Racines CreeRacines(double a, double b, double c)
   {
      double delta = b*b - 4.0 * a*c ;
      if (delta < 0.0)
         return new RacinesSol0(a, b, c, delta) ;
      else if (delta > 0.0)
         return new RacinesSol2(a, b, c, delta) ;
      else
         return new RacinesSol1(a, b, c, delta) ;
   }
}
```

Questions

1. Pour quelle raison le constructeur de `Racines` est-il déclaré comme **protected** ?
2. À quoi sert le mot-clé **static** de la méthode *CreeRacines* ?