L2 info S3
UE : Programmation Orientée Objets 1 - Java

TP3 - Héritage versus composition

Objectifs

A la fin de ce TP, vous devez être capables de :

1. Distinguer la composition de l'héritage.
2. Programmer les 2 formes.
3. Consolider les apprentissages du TD3 sur l'héritage, à savoir :
   — Utiliser les modificateurs `private` et `protected` pour un attribut.
   — Faire des affectations en respectant la compatibilité classe parente-classe dérivée.
   — Appliquer le polymorphisme.

1 Point et cercle (composition et héritage)

On dispose de la classe suivante :

```
class Point
{
  double x, y ;

  public Point(double x, double y)
  {
     this.x = x ;
     this.y = y;
  }

  public void deplace(double dx, double dy)
  {
     x += dx ;
     y += dy ;
  }

  public String toString()
  {
     return "Point de coordonnees : " + x + ", " + y ;
  }
}
```

On souhaite réaliser une classe `Cercle` disposant des méthodes suivantes :

— constructeur recevant en argument les coordonnées du centre du cercle et son rayon,
— `deplaceCentre` pour modifier les coordonnées du centre du cercle (faire appel à la méthode `deplace` de la classe `Point`),

— `changerRayon` pour modifier le rayon du cercle,
— `getCentre` qui fournit en résultat un objet de type `Point` correspondant au centre du cercle,
— `toString` qui renvoie sous forme de `String` les coordonnées du centre du cercle et de son rayon.

Questions

1. Composition (avoir)
   (a) Définir la classe `Cercle` comme possédant un membre de type `Point`
   (b) Ecrire un petit programme mettant en jeu les différents fonctionnalités de la classe `Cercle` comme par exemple :

```
public class TestCercle
{
    public static void main(String[] args)
    {
       Cercle c = new Cercle(3, 8, 2.5) ;
       System.out.println(c);
       c.deplaceCentre(1, 0.5) ;
       c.changeRayon(5.25);
       System.out.println(c);
       Point pt = c.getCentre();
       System.out.println(pt);
    }
}
```

2. Héritage (être)
   (a) Définir la classe `Cercle` comme classe dérivée de `Point` (mot clé `extends`).
   (b) Reprendre le programme mettant en jeu les différents fonctionnalités de la classe `Cercle`.
3. Visibilité des attributs
   Dans la suite de l'exercice, la classe `Cercle` est dérivée de `Point`.
   (a) Modifier les deux classes afin que les attributs deviennent privés. Ajouter des accesseurs à la classe `Point` : `pubic double getX()` et `public double getY()`. Modifier la classe `Cercle` en conséquence.
   (b) Modifier la classe `Point` afin que ses attributs soient `protected`. Modifier la classe `Cercle` en conséquence. Tester l'accessibilité des attributs de `Point` dans le programme principal.
4. Affectation, liaison dynamique
   La classe `Cercle` est dérivée de `Point`.
   Parmi les instructions suivantes, dire quelles sont acceptées, refusées. Expliquer pourquoi.

```
public class TestCercle
{
    public static void main(String[] args)
    {
       Point pt1, pt2 ;
       Cercle c1, c2 ;

       pt1 = new Point(4.2, 7.3) ;
       pt2 = new Cercle(14.7, 62, 5.2) ;
       c1 = new Cercle(3.56, 7.84, 2.25) ;
       c2 = new Point(56.1, 76.9) ;

       pt1.deplace(0.5, 0.5) ;
       pt2.deplace(0.3, 0.8) ;
       pt2.changerRayon(6.21) ;
       c1.deplace(0.4, 0.5) ;
       c1.changeRayon(2.52) ;
       c1.x = 4.78 ;
    }
}
```

2 Moteur et Voiture (composition)

1. Ecrire une classe `Moteur` comportant
   pour attribut
   — `carburant` (de type entier)
   des constructeurs
   — sans argument : la quantité de carburant vaut 0
   — avec un argument : la quantité initiale de carburant dans le moteur
   les méthodes
   — `demarrer` : réduit d'une unité le carburant disponible et retourne un booléan indiquant si l'opération est possible ou pas. Affiche à la console "Moteur démarre avec ... litres".
   — `utiliser` : prend en argument la consommation de carburant et retourne le niveau de carburant après consommation. Affiche à la console "Moteur utilise ... litres". Si le carburant n'est pas suffisant pour la consommation demandée, le moteur utilise le carburant disponible.
   — `arreter` : affiche à la console "Moteur arrêté".
   — `fairePlein` : a pour argument la quantité de carburant ajoutée. Affiche à la console "Plein avec ... litres".
2. Créer une classe `Voiture` comportant
   pour attributs
   — `modele` (du véhicule)
   — un attribut de type `Moteur`
   des constructeurs
   — avec un seul argument : le nom du modèle
   — avec deux arguments : le nom du modèle et la quantité initiale de carburant dans le moteur
   — avec deux arguments : le modèle et le moteur
   les méthodes
   — `demarrer` qui retourne une valeur booléenne pour savoir si l'opération est possible ou non.
   — `rouler` qui prend en argument la consommation de carburant prévue. Elle consiste à utiliser le moteur. Elle afficher un message d'erreur "Panne d'essence" lorsque la consommation prévue n'est pas possible.

— `arreter` qui arrête le moteur
— `fairePlein` qui fait le plein avec la quantité de carburant transmis en argument.

Prévoir un jeu de tests pour valider le comportement d'une voiture, par exemple, qui a 40 litres de carburant et qui effectue 5 trajets consommant 10 litres chacun, faire le plein quand c'est nécessaire.

Exemple d'exécution :

```
Moteur démarre avec 40 litres
Moteur utilise 10 litres
Moteur utilise 10 litres
Moteur utilise 10 litres
Moteur utilise 10 litres
Moteur utilise 9 litres
Panne
Plein avec 50 litres
Moteur utilise 10 litres
Moteur arrete
```

3. Un parc de voitures nécessite régulièrement de faire tourner les moteurs et de faire le plein si nécessaire. Modéliser ce problème par une classe `ParcVoiture` comportant un tableau de voitures et ajouter une méthode tester faisant tourner le moteur (`demarrer`) de chaque voiture. La valeur de l'attribut sera mis à jour par un constructeur. Faire le test avec 3 voitures.

   Un exemple d'exécution :

```
Test parc voitures :
Demarrer la voiture 1
Moteur demarre avec 5 litres
Moteur arrete
Demarrer la voiture 2
Moteur demarre avec 6  litres
Moteur arrete
Demarrer la voiture 3
Panne
Plein avec 20 litres
```