TD Concepts objet et Java

2014-15

Feuille 2

Exercice 1.

Soit la classe simple pour représenter un cercle à partir de son centre de son rayon suivante. Trouver et corriger l'ensemble des défauts de cette classe.

```
package td3;

import java.awt.Point;

/**
 * Un cercle defini par son centre et son rayon
 */
public class Cercle {

  private Point centre;
  private double rayon;

  /**
   * Construit un Cercle.
   * @param c le centre de ce cercle
   * @param r le rayon de ce cercle
   */
  public Cercle(Point c, double r) {
    Point centre = new Point(c);
    rayon = r;
  }

  /**
   * Translate un cercle par le vecteur (dx, dy)
   * @param dx la translation en abscisse
   * @param dy la translation en ordonnée
   */
  public void deplacer(int dx, int dy) {
    centre.setLocation(centre.getX() + dx, centre.getY() + dy);
  }

  /**
   * Renvoie le centre de ce cercle.
   * @return le centre du cercle.
   */
  public Point getCentre() {
    return centre;
  }

  /**
   * Change le centre du cercle.
   * @param centre le nouveau centre
   */
  public void setCentre(Point centre) {
    this.centre = centre;
```

```
  }

  /**
   * Renvoie le rayon de ce cercle.
   * @return le rayon de ce cercle.
   */
  public double getRayon() {
    return rayon;
  }

  /**
   * Change le rayon de ce cercle.
   * @param rayon le nouveau rayon
   */
  public void setRayon(double rayon) {
    this.rayon = rayon;
  }

  /**
   * Une description textuelle de ce cercle.
   * @return une chaine de caractere decrivant ce cercle.
   */
  public String toString() {
    return ("Cercle de centre " + centre + " et de rayon " + rayon);
  }

  /**
   * Le programme principal.
   * @param args les arguments du programme principal
   */
  public static void main(String[] args) {
    Point p = new Point(0, 0);
    Cercle c= new Cercle(p, 20);
    System.out.println("Cercle c avant déplacement: " + c);
    c.deplacer(5, 5);
    System.out.println("Cercle c après déplacement: " + c);
  }
}
```

Exercice 2.

On dispose déjà d'une classe (compilée) représentant un nombre complexe, définie dans un package qui ne nous appartient pas et dont nous ne disposons pas du code source. Nous souhaitons utiliser cette classe pour lui ajouter deux comportements, pour obtenir la valeur du module et de l'argument du complexe représenté.

1. Télécharger l'archive `complexe.jar` sur le site du cours, et l'ajouter comme une librairie au projet courant (`Project properties > Java Build Path > Librairies > Add External JARs`). La classe apparaît désormais dans l'onglet `'Referenced librairies'` : parcourir les éléments auquel on a accès de la sorte.
2. Créer une nouvelle classe `MonComplexe` réutilisant la classe précédente de la manière la plus appropriée. Lui ajouter des méthodes pour obtenir la valeur du module et de l'argument :
   — $module = \sqrt{re^2 + im^2}$
   — $argument = acos(\frac{re}{module})$
3. Essayer l'affichage actuel d'un complexe. Modifier le code de la classe (et pas le code appelant) afin que l'affichage indique désormais également le module et l'argument du complexe.
4. Faire ce qu'il faut pour qu'il soit désormais possible de préciser directement un nouvel argument ou un nouveau module pour un complexe, sans modifier l'implémentation précédente de la classe.
   — $re = module * cos(argument)$
   — $im = module * sin(argument)$

5. Refaire la même chose, mais cette fois-ci en définissant une nouvelle classe héritant de la classe `td1.Complexe` définie précédemment. Comment peut-on réutiliser cette classe, et faut-il redéfinir l'égalité entre complexes qui y avait été définie ?

Exercice 3.

On souhaite mettre en place un éditeur simple de dessins à base de formes géométriques, sous forme d'un package `geometrie` regroupant les classes permettant de manipuler des formes géométriques bidimensionnelles simples, ici : des cercles, des rectangles, et des carrés. On a les contraintes suivantes :

- Chaque forme possède un centre de gravité (instance de `java.awt.Point`) ainsi qu'une couleur (instance de `java.awt.Color`). Les attributs correspondants ne devront pas être directement partagés avec l'extérieur de la classe.
  - un cercle possède en outre un rayon
  - un rectangle possède en outre une largeur et une hauteur.
- Toute forme géométrique doit pouvoir avoir les comportements suivants :
  - *translation*, prenant en paramètres deux nombres représentant un déplacement horizontal et vertical
  - *homothétie*, prenant en paramètre un nombre représentant le ratio de l'homothétie
  - toute forme est capable de donner sa représentation sous la forme d'une chaîne de caractères donnant le nom de la forme (classe) et la description textuelle de chacun de ses attributs. Par exemple, la chaîne de caractères produite pour un cercle pourrait être :

```
[Cercle
   [centre de gravité : x=10 , y=4]
   [rayon : 20]
   [couleur : r=82 , g=255 , b=0]
]
```

- Des constructeurs utiles devront être proposés.

1. Implémenter la spécification demandée.
2. On souhaite ajouter une forme de carré, *via* la spécialisation de la classe pour un rectangle. On veillera à ce que cette classe ne puisse plus être dérivée.
3. On souhaite à présent pouvoir construire des listes de formes géométriques pour opérer des regroupements d'objets.
   (a) Commencer par créer une classe de test définissant un programme créant une collection sous forme de liste utilisant la classe `java.util.ArrayList<E>`. Y ajouter un objet de chaque type de forme définie, et afficher le contenu de la liste à l'aide d'un *itérateur* (cf. `http://docs.oracle.com/javase/6/docs/api/java/util/Collection.html#iterator()`). Quelles sont les méthodes `toString` qui sont appelées pour chaque objet ? Que se passe-t-il si l'on enlève un élément de la collection à un indice qui n'existe pas ?
   (b) Créer à présent une classe pour représenter une simple collection pour des formes géométriques uniquement par réutilisation de la classe `java.util.ArrayList` .
   (c) Ajouter à la classe précédente des méthodes `translation` et `homothetie` qui déplacent ou modifient la taille de l'ensemble des objets d'une telle collection de formes géométriques, et redéfinir la méthode `toString` de manière à ce qu'elle produise un affichage plus approprié.
   (d) Tester l'ensemble des méthodes définies dans un programme principal associé à la classe définie.
   (e) On souhaite à présent définir une collection dont tous les membres seront des formes géométriques, mais de même type. Définir ainsi une nouvelle classe sur le modèle de la précédente, avec notamment la définition de méthodes pour l'homothétie et la translation, ainsi que la redéfinition

de la méthode `toString`. Un programme de test devra confirmer qu'il n'est pas possible de mélanger des formes géométriques de types différents. Confirmer qu'une opération illicite (un ajout d'un objet d'un autre type) est bien détectée à la compilation.