TD2 : Héritage, Polypmorphisme 2022-2023

1 Interfaces et Classes

Nous souhaitons modéliser des figures géométriques en deux dimensions (Cercles, Rectangles,...). Toute figure géométrique devra implémenter l'interface suivante :

```java
public interface Figure {
    /**
     * @return le genre de la figure geometrique
     */
    public String genre();

    /**
     * @param x et y : coordonnees du point a tester
     * @return vrai si le point est contenu dans la figure geometrique
     */
    public boolean inside(double x, double y);
}
```

Question 1 : Proposez l'implémentation de deux classes de figures géométriques :
- La classe **Cercle** disposant d'un constructeur prenant les coordonnées x et y du centre du cercle ainsi que son rayon.
- La classe **Rectangle** disposant d'un constructeur prenant les coordonnées x et y du centre du rectangle ainsi que sa largeur et sa hauteur.

Question 2 : Les instructions suivantes sont-elles valides ou non ? (Justifiez votre réponse)

```
1) Cercle c = new Cercle(1.0, 1.0, 0.2);
2) Rectangle r = new Rectangle(1.0, 2.0, 1.0, 1.0);
3) Figure f = new Figure();
4) Figure f2 = new Cercle(1.0, 2.0, 2.0);
5) Rectangle r2 = new Cercle(1.0, 2.0, 0.2);
6) Cercle c2 = f2;
```

Question 3 : Nous souhaitons créer une liste (type **List<T>**) pouvant contenir des cercles et des rectangles, est-ce possible ? Si oui, quel doit être le type de cette liste ?

Question 4 : Nous souhaitons écrire une fonction qui à partir de la liste de la question précédente renvoie une table de correspondance (type **Map<TCle, TVal>**) dont la clé correspondra à un genre de figure géométrique et la valeur associée au nombre d'occurrences de ce genre de figure dans la liste passée en paramètre. Proposez une implémentation de cette fonction.

Question 5 : Une figure géométrique peut être définie comme résultant de l'intersection de deux autres figures géométriques (par exemple l'intersection d'un cercle et d'un rectangle). Proposez une implémentation de la classe **Intersection** définissant une figure géométrique comme l'intersection de deux autres figures géométriques.

Question 6 : On souhaite introduire d'autres opérations combinant 2 figures (union, différence, différence symétrique, . . .) ; proposez une nouvelle organisation des classes qui mutualise une partie des informations communes à ces différentes opérations.

2 Héritage

Introduction : On considère l'interface **SpecifArticle** et la classe concrète **Article** qui implémente cette interface.

```java
public interface SpecifArticle {
    // Designation de l'article
    public String getDesignation();
    // Quantite en stock
    public int getQuantite();
    // Prix HT
    public double getPrixHT();
    // Prix TTC = prix HT * taux de TVA (1.196)
    public double getPrixTTC();
    // Augmenter le stock de la quantite q ; q > 0
    public void ajouter(int q);
    // Reduire le stock de la quantite q ; 0 < q <= quantite
    public void retirer(int q);
}
```

```java
public class Article implements SpecifArticle {
    private String designation;
    private int quantite;
    private double prixHT;
    private static final double TAUX_TVA = 1.196;
    // constructeur
    public Article(String designation, int quantite, double prixHT) {
        this.designation = designation;
        this.quantite = quantite;
        this.prixHT = prixHT;
    }
    // accesseurs
    public String getDesignation() { return this.designation; }
    public int getQuantite() { return this.quantite; }
    public double getPrixHT() { return this.prixHT; }
    public double getPrixTTC() { return this.prixHT * TAUX_TVA; }
    // ajout / retrait
    public void ajouter(int q) { this.quantite += q; }
    public void retirer(int q) { this.quantite -= q; }
}
```

Question 1 : Complétez la classe Article afin que le programme ci-dessous donne le résultat attendu.

```java
Article art = new Article("Lampe", 10, 58.50);
System.out.println(art);
// output :
// Lamp, 58.50 euros (10 en stock)
```

Question 2 : Modifier la classe Article afin d'implémenter l'interface **java.lang.Comparable<T>** ; la comparaison entre articles se fait sur la base de leurs désignations respectives. On notera que la classe **String** implémente l'interface **Comparable<String>**.

Question 3 : Écrivez une classe concrète **Livre** héritant de **Article** et implémentant l'interface **SpecifLivre** donnée ci-après. Le constructeur de cette classe doit initialiser une instance à l'aide des valeurs de désignation, quantité, prix HT, nombre de pages, et n°ISBN données en paramètre.

```java
public interface SpecifLivre extends SpecifArticle {
    // Nombre de pages du livre
    public int getNombrePages();
    // Numero ISBN du livre
    public String getNumeroISBN();
}
```

Question 4 : Complétez la classe **Livre** de manière à rendre possible cette séquence :

```
Article art = new Livre("Le saigneur des agneaux", 200, 24.30, 654, "2 7457 1234 7");
System.out.println(art);
// output :
// "Le saigneur des agneaux", 24.30 (200 en stock) ISBN 2 7457 1234 7 (654 pages)
```

Question 5 : La classe Catalogue permet de gérer une collection d'articles à l'aide d'un tableau d'articles et un nombre d'articles :

```
public class Catalogue {
    private SpecifArticle [] listeArticles;
    private int nbArticles;
    ...
    public void afficher();
    public void trier();
}
```

Écrivez la méthode **public void trier()** permettant de trier les articles par ordre lexicographique croissant suivant la désignation.

NB : Le tri d'un tableau peut-être obtenu par un appel de cette méthode de classe de la classe **java.util.Arrays** :

```
public static <T> void sort(T[] tab, int inf, int sup);
```

Cette méthode réordonne le tableau tab donné en paramètre entre les indices inf (inclus) et sup (exclu), en comparant entre eux les éléments à l'aide de la méthode **compareTo()**.

Question 6 : Offrir la possibilité de spécifier ses propres comparateurs (en plus de la comparaison par défaut ci-dessus) ; un comparateur est une classe qui implémente l'interface **Comparator<T>**, sachant que le résultat de **compare()** a la même signification que celui de **compareTo()**.

```
public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

Il faudra donc programmer un comparateur de désignation, un comparateur de prix, etc... Le tri d'un tableau, à l'aide d'un comparateur, peut-être obtenu par un appel de cette méthode de classe de la classe **java.util.Arrays** :

```
public static <T> void sort(T[] tab, int inf, int sup, Comparator<T> c);
```

Client

Dans cette section, on s'attarde à l'écriture d'une méthode main. Tout d'abord, programmer les comparateurs d'**Articles** dont nous aurons besoin sous forme de classes externes à la classe **Article** : comparateur par désignation, par prix, par quantité, etc... ;

Bonus 1 : donner la possibilité de choisir le sens de tri.

L'algorithme de la méthode main est le suivant :
- créer une instance de chaque comparateur ;
- Instancier un certain nombre d'Article et de Livre, et les ajouter à un Catalogue ;
- trier :
  - d'abord avec le comparateur par défaut : **public void trier()**
  - puis avec chacune des implémentations de comparator : **public void trier(Comparator<T> c)**
- affiche le contenu du Catalogue à chaque tri.

Bonus 2 : définir un comparateur à l'aide d'une lambda.

Conclusion On voit que cette solution est souple et extensible (par le client) ; Nous avons ici nous-même définit plusieurs Comparator, mais on pourrait laisser la charge au client, celui qui va utiliser notre code de gestion d'Article, d'implémenter ses propres Comparator. Avec les méthodes que nous proposons, le client peut alors trier comme il le souhaite les articles. On appelle cette ensemble de méthode une API : Application Programmation Interface.