

# TD2 : Héritage, Polymorphisme

04/10/21

## 1 Interfaces et Classes

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

```
1 public interface Figure {
2     /**
3      * @return le genre de la figure geometrique
4      */
5     public String genre();
6
7     /**
8      * @param x et y : coordonnees du point a tester
9      * @return vrai si le point est contenu dans la figure geometrique
10    */
11    public boolean inside(double x, double y);
12 }
```

**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<TClé, 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.

```

1 public interface SpecifArticle {
2     // Designation de l'article
3     public String designation();
4     // Quantite en stock
5     public int quantite();
6     // Prix HT
7     public double prixHT();
8     // Prix TTC = prix HT * taux de TVA (1.196)
9     public double prixTTC();
10    // Augmenter le stock de la quantite q ; q > 0
11    public void ajoute(int q);
12    // Reduire le stock de la quantite q ; 0 < q <= quantite
13    public void retirer(int q);
14 }

```

```

1 public class Article implements SpecifArticle {
2     private String designation;
3     private int quantite;
4     private double prixHT;
5     private static final double tauxTVA = 1.196;
6     // constructeur
7     public Article(String designation, int quantite, double prixHT) {
8         this.designation = designation;
9         this.quantite = quantite;
10        this.prixHT = prixHT;
11    }
12    // accesseurs
13    public String designation() { return this.designation; }
14    public int quantite() { return this.quantite; }
15    public double prixHT() { return this.prixHT; }
16    public double prixTTC() { return this.prixHT * tauxTVA; }
17    // ajout / retrait
18    public void ajouter(int q) { this.quantite += q; }
19    public void retirer(int q) { this.quantite -= q; }
20 }

```

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

```

1 Article art = new Article("Lampe", 10, 58.50);
2 System.out.println(art);
3 // output :
4 // Lampe, 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.

```

1 public interface SpecifLivre extends SpecifArticle {
2     // Nombre de pages du livre
3     public int nombrePages();
4     // Numero ISBN du livre
5     public String numeroISBN();
6 }

```

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

```
1 Article art = new Livre("Le saigneur des agneaux", 200, 24.30, 654, "2 7457 1234 7");
2 System.out.println(art);
3 // output :
4 // "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 :

```
1 public class Catalogue {
2     private SpecifArticle [] listeArticles;
3     private int nbArticles;
4     ...
5     public void afficher();
6     public void trier();
7 }
```

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** :

```
1 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 au client la possibilité de définir 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()**.

```
1 public interface Comparator<T> {
2     public int compare(T o1, T o2);
3 }
```

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** :

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

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

**Bonus 2** : définir le comparateur avec une lambda fonction.

## Remarques

### Exo 2, question 6

#### Comparateurs externes

1) Mise en place dans la classe Article :

- programmer **compareTo** de façon à effectuer la comparaison avec un critère par défaut (ici, la désignation, cf question 5).

## 2) Utilisation par le client :

- programmer les comparateurs d'**Articles** dont il a besoin sous forme de classes externes à la classe **Article** : comparateur par désignation, par prix, etc...;
- créer une instance de chaque comparateur nécessaire;
- trier,
- soit avec le comparateur par défaut : **public static void sort(T[] tab, int inf, int sup)**
- soit avec un autre comparateur : **public static void sort(T[] tab, int inf, int sup, Comparator<T> c)**
- puis afficher.

**Conclusion** Cette solution est souple et extensible (par le client) ; l'implémenteur d'**Article** n'a pas besoin de prévoir toute une panoplie de comparateurs qui n'intéresseront pas le client et qui le limiteront dans ses possibilités.