HMIN215 - Exercices sur la généricité bornée

1 Rayon

Nous disposons d'une classe non générique `Rayon` destinée à représenter des rayons dans un magasin. Un rayon contient des produits. L'une des méthodes consiste en particulier à réaliser un listing des produits en rayon et pour cela nous supposons que les éléments dans les rayonnages ont une méthode `String etiquette()`.

```
public class Produit{
.......
     public String etiquette(){.....}
}

public class Rayon{
     private ArrayList<Produit> contenu=new ArrayList<Produit>();    ....
     public String listingContenu()
     {
          String listing="";
          for (Produit c:contenu)
               listing += c.etiquette();
          return listing;
     }     ....
}
```

Question 1 *Complétez la classe* `Produit` *par un attribut représentant son étiquette, des accesseurs et des constructeurs. Complétez la classe* `Rayon` *par un constructeur et une méthode permettant de mettre un produit en rayon.*

Question 2 *Transformez la classe* `Rayon` *en classe générique, paramétrée par le type des éléments que l'on veut mettre en rayon. Ces éléments doivent disposer d'une étiquette mais ne sont pas forcément des produits (créez une interface* `ObjetAvecEtiquette` *pour représenter cette contrainte). Cela permettra de créer des rayons d'objets d'un type homogène. La classe* `Produit` *doit implémenter cette interface* `ObjetAvecEtiquette`*.*

Question 3 *Créez une classe* `Livre` *comme sous-classe de* `Produit`*. Dans un main, créez une classe représentant des rayons de* `Livres`*, puis des rayons de* `Produits`*. Mettez quelques objets dans ces rayons, puis affichez le listing correspondant.*

2 Compléments sur la file d'attente

Nous continuons à travailler avec la classe générique `FileAttente` et nous nous donnons quelques classes partielles représentant des personnes. Vous testerez au fur et à mesure les méthodes créées dans un programme.

```
public class Personne{...}
public class Adulte extends Personne{...}
public class Enfant extends Personne{...}
```

Question 4 *Peut-on écrire les deux instructions suivantes (justifier) ?*

```
FileAttente<Personne> fp = new FileAttente<Adulte>(); //1
FileAttente<Adulte> fa = new FileAttente<Personne>(); //2
```

Question 5 *Déclarer et créer une file d'attente d'enfants ($f1$), une file d'attente d'adultes ($f2$) et une file d'attente de personnes ($f3$).*

Question 6 *On complète à présent la méthode* `entre` *de manière à vérifier que l'objet qui rentre dans une file d'attente y est invité.*

```
public void entre(E e)
   {
       if (e.isInvite()) contenu.add(e);
      else System.out.println("non invite");
   }
```

1. *Ecrire une interface représentant les objets disposant d'une méthode* `isInvite`.
2. *Modifiez la classe* `Personne` *de manière à ce qu'elle implémente cette interface*
3. *Modifiez la classe générique* `FileAttente` *de manière à ce que les objets du type passé en paramètre disposent de la méthode* `isInvite`.

Question 7 *Ecrivez une méthode d'instance qui consiste à déplacer le premier élément d'une file d'attente (receveur du message) dans une autre file d'attente passée en paramètre. Cette méthode doit permettre de déplacer le premier élément de $f1$ (ou de $f2$) dans $f3$ (ces files sont toujours celles de la question 5).*

Question 8 *Ecrivez une méthode symétrique qui consiste à faire entrer dans une file d'attente (receveur) le premier élément d'une file d'attente passée en paramètre. Cette méthode doit permettre de faire entrer dans $f3$ un élément qui sort de $f1$ ou de $f2$ (ces files sont toujours celles de la question 5).*

3 Comparaison, opérations sur des objets comparables : une application de la généricité bornée (à faire uniquement si vous êtes en avance - pas de questions aux contrôles portant sur ce point)

En Java, une interface de l'API permet de représenter des objets *comparables*. Elle est munie d'une opération de comparaison qui retourne 0 si les deux objets sont égaux (`equals` est appelée), un nombre négatif si le receveur précède l'argument, et un nombre positif si le receveur est un successeur de l'argument.

```
public interface Comparable<T>
{
   int compareTo(T o);
}
```

Elle est implémentée par un grand nombre de classes, dont la classe `String`, ce qui veut dire que cette dernière contient les éléments suivants. Remarquons que l'on déclare que les `String` sont comparables avec d'autres `String`.

```
public final class String extends Object implements Comparable<String> (...)
{
     ....
     public int compareTo(String anotherString){
     // compare par ordre lexicographique
     }
     ....
}
```

Lorsqu'une classe implémente l'interface `Comparable`, l'ordre total défini par la méthode `compareTo` est appelé l'ordre naturel pour les objets de cette classe.

Cette interface permet de réaliser différents traitements sur une collection, tels que trouver le minimum, le maximum ou trier la collection. De telles méthodes se trouvent dans la classe `Collections`.

Pour vous faire comprendre son principe, nous montrons ci-dessous une méthode qui imite l'une des méthodes existantes. Elle retourne l'élément maximum pour toutes les collections vérifiant *l'interface List* et paramétrées par des éléments implémentant *l'interface Comparable*.

```
public static<E extends Comparable<E>> E max(List<E> c)
{
    if (c.isEmpty())
        return null;
    E max = c.get(0);
    for (E e : c)
        if (e.compareTo(max)>0)
            max = e;
    return max;
}
```

Le code ainsi écrit n'utilise que des interfaces. On remarque aussi que l'algorithme n'est pas une méthode d'instance. On peut s'en servir sur des ArrayList, des LinkedList, etc. On a écrit un code "générique" au sens où il s'applique à un large ensemble de collections, comme le montre le programme ci-dessous (testez-le).

```
ArrayList<Integer> listeEntiers = new ArrayList<Integer>();
listeEntiers.add(4);  listeEntiers.add(8);
System.out.println(max(listeEntiers));

LinkedList<String> listeChaines = new LinkedList<String>();
listeChaines.add("galette");
listeChaines.add("crèpes");
listeChaines.add("bugnes");
System.out.println(max(listeChaines));
```

Question 9

On veut appliquer la méthode `max` *sur une liste de personnes (disposant d'un nom et d'un âge). On compare les personnes d'après l'ordre lexicographique de leur nom.*

- *Comment devez-vous compléter la classe représentant les personnes pour que ce soit possible ?*
- *Ecrivez un programme qui crée une liste de personnes, puis affiche une personne dont le nom est le plus grand dans l'ordre lexicographique.*
- *Recherchez dans la documentation de la classe* `Collections` *la méthode* `max`*, regardez sa signature et appliquez-la à votre liste de personnes pour obtenir (en principe) le même résultat que la méthode* `max` *de cet énoncé.*
- *Recherchez dans la documentation de la classe* `Collections`*, les méthode* `sort`*, identifiez celle qui peut s'appliquer ici et appliquez-la à votre liste de personnes pour obtenir une liste triée (testez en affichant la liste après le tri).*

Maintenant imaginons que l'on ne veuille pas utiliser l'ordre "naturel" sur les objets (par exemple sur les String), mais un autre ordre pour rechercher le maximum dans une liste ou pour trier celle-ci.

Dans un premier temps, on définit une classe représentant un comparateur de chaînes. Cette classe implémente une interface `Comparator` qui demande de définir une méthode `compare`. On ne se préoccupera pas de la méthode `equals` que contient cette interface.

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

Le comparateur de chaîne est donc comme suit.

```
class comparateurTailleChaines implements Comparator<String>
{
    public int compare(String s1, String s2) {
        boolean egal = (s1.length() == s2.length());
        boolean inf = (s1.length() < s2.length());
        if (egal) return 0;
        else
        if (inf) return -1;
            else return 1;
    }
}
```

On peut à présent définir une autre version de la méthode `max` admettant en paramètre une liste et un comparateur (cette méthode imite également une méthode de la classe `Collections`).

```
public static<E> E max(List<E> c, Comparator<E> comp)
{
    if (c.isEmpty())
        return null;
    E max = c.get(0);
    for (E e : c)
    if (comp.compare(e, max)>0)
        max = e;
    return max;
}
```

On peut alors l'utiliser pour trouver une plus longue chaîne dans une liste de chaînes (testez le programme ci-dessous).

```
LinkedList<String> listeChaines = new LinkedList<String>();
listeChaines.add("galette"); listeChaines.add("crèpe"); listeChaines.add("bugne");
listeChaines.add("crèpe dentelle");
System.out.println(max(listeChaines));
System.out.println(max(listeChaines,new comparateurTailleChaines()));
```

Question 10

- *Créez une classe comparateur de personnes, qui compare deux personnes suivant leur âge.*
- *Recherchez dans la liste de personnes une personne qui est la plus âgée à l'aide de la méthode* ***`max`*** *de l'énoncé.*
- *Réalisez le même traitement avec la méthode* ***`max`*** *apparentée de la classe* ***`Collections`****.*
- *Triez la liste de personnes suivant leur âge (vérifiez en affichant la liste après le tri).*