Interfaces

1 Billets de train

Nous présentons quelques éléments d'un logiciel pour une agence de voyage. Nous étudierons seulement la représentation des billets.

Question 1. Ecrivez en Java une interface `IBilletTrain` représentant un billet de train.

Un billet de train correspond à un trajet entre une origine et une destination. Une énumération interne à l'interface définit 3 types de tarifs : `Première, ProSeconde, Seconde`. Pour un billet de train, on doit disposer de méthodes pour connaître la valeur (1) de la date et heure (au format entier AAMMJJHHMnMn), (2) de l'origine et de la destination (au format chaîne de caractères <`origine`>-<`destination`>, par exemple ‘‘`Montpellier-Nîmes`’’), (3) d'un type de tarif, (4) d'un prix. On doit également disposer (5) d'un taux de remboursement pour chaque type de tarif, mis en oeuvre par une `Map` et dont les clefs sont les types de tarif, (6) d'une méthode permettant de donner une valeur à ces 3 réels, et (7) d'une méthode calculant le tarif de remboursement : celui-ci s'obtient en appliquant le taux de remboursement au prix.

Question 2. Dans une interface, les attributs ne peuvent être que `public final static`. Expliquer pourquoi vous avez pu mettre en place une structure stockant les taux de remboursement et écrire une méthode modifiant les taux.

Question 3. Ecrivez en Java une classe `BilletTrain` implémentant l'interface `IBilletTrain` de la question 1. On ne souhaite pas pouvoir créer d'instance propre de cette classe (seulement des instances de ses sous-classes). L'entête doit comporter l'information correspondante. Ecrivez ensuite en Java *l'entête* d'une classe `BilletTrainTGV` qui spécialise la classe `BilletTrain`.

2 Extraction d'une interface

Question 4. On suppose connue une classe `Personne` et ci-dessous on vous donne le code d'une classe représentant des files d'attente de personnes. Proposez une interface décrivant le type de cette classe, c'est-à-dire ce qu'elle peut exposer de manière publique aux autres classes qui veulent l'utiliser.

```
public class FileAttente {
      private String nomFile;
      private static String reglementationFile = "sans priorité";
      private ArrayList<Personne> contenu;
      public FileAttente(){contenu=new ArrayList<Personne>();}
      public void entre(Personne p){contenu.add(p);}
      public Personne sort()
      {
            Personne p=null;
            if (!contenu.isEmpty())
                  {p=contenu.get(0);
                  contenu.remove(0);}
                  return p;
      }
      public boolean estVide(){return contenu.isEmpty();}
      public int taille(){return contenu.size();}
      public String toString(){return ""+descriptionContenu();}
      private String descriptionContenu()
      {
            String resultat = "";
            for (Personne p:this.contenu)
                  resultat += p + " ";
            return resultat;
      }
      public void vider(){
            int taille = taille();
            for (int i=0; i<taille; i++)
                  this.sort();
      }
}
```

Question 5. Comment modifiez-vous la classe `FileAttente` pour qu'elle implémente l'interface que vous avez créée ?

Question 6. Proposez une interface pour représenter les files d'attente avec des statistiques. Des opérations supplémentaires permettent de connaître le nombre d'entrées et le nombre de sorties qui ont eu lieu depuis la création de la file.

Question 7. Proposez une classe qui implémente cette nouvelle interface (file d'attente avec statistiques).

Question 8. Dessinez le diagramme UML de vos classes et interfaces avec leurs relations (associations, spécialisation, implémentation).

3 Interfaces `Comparable` et `Comparator`

3.1 Ordre naturel entre objets avec l'interface `Comparable`

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){
     // ici code qui compare this et anotherString 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, pourvu qu'elles implémentent l'interface `List`. 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 (ce serait l'ordre naturel entre les personnes).

- Comment devez-vous compléter la classe représentant des personnes (vous pouvez utiliser votre classe `Etudiant`) 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 méthodes `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).

3.2 Définitions de comparaisons entre objet avec l'interface `Comparator`

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

Pour cela, on définit une classe qui 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);
}
```

Par exemple, pour définir un autre opérateur de comparaison entre chaînes de caractères, basé sur la longueur des chaînes plutôt que sur leur ordre lexicographique, on définit une classe implémentant l'interface `Comparator<String>` comme suit.

```
public 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 l'une des plus âgées à l'aide de la méthode `max` de l'énoncé.
- Réalisez le même traitement avec la méthode `max` apparentée de la classe `Collections`.
- Trier la liste de personnes suivant leur âge, en recherchant une méthode `sort` appropriée (vérifiez en affichant la liste après le tri).