Examen

3 heures - documents écrits autorisés
janvier 2005

Problème TOUR DE FRANCE
Le fil directeur de ce sujet est la modélisation d'un programme qui permet de gérer le classement du Tour de France. Cependant les réponses aux exercices sont indépendantes et il est possible d'utiliser les réponses à une question non traitée pour résoudre les autres questions (au sein d'une même question, les éléments peuvent dépendre les uns des autres). Les classes ou interfaces que vous coderez devront appartenir à un paquetage `janvier2005.solution` alors que les classes ou interfaces dont le code est supposé connu ou est donné dans le sujet seront dans le paquetage `janvier2005.sujet`.

Le Tour de France est une course à étapes. Dans une telle course, le résultat d'un coureur est déterminé par le cumul des temps qu'il a réalisés à chacune des étapes et le cumul des points qu'il a obtenu lors de ces étapes dans différents classements. Différents classements existent selon différents critères : meilleur temps ("maillot jaune", plus petit temps cumulé), meilleur sprinter (maillot vert, plus grand nombre de "points verts"), meilleur grimpeur ("maillot à pois rouges", plus grand nombre de "points montagne"), meilleur jeune (meilleur temps des -25 ans), etc.

Temps Le temps réalisé par un coureur est défini par la donnée de 3 entiers représentant respectivement un nombre d'heures, de minutes et de secondes. Le nombre d'heures n'est pas borné, alors que les nombres de minutes et secondes sont strictement inférieurs à 60. De manière évidente 63 minutes correspondra à 1 heure et 03 minutes.
Il doit être possible de comparer 2 temps entre eux et d'ajouter un temps à un autre. La classe doit disposer d'une méthode `toString` retournant l'heure sous la forme `"xhymnzsec"`.

Q 1 . Donnez le code JAVA d'une classe `Temps` respectant ces contraintes. Cette classe implémentera l'interface `java.lang.Comparable` (la javadoc de la méthode `compareTo` de cette interface est donnée en annexe) et disposera de 2 constructeurs : l'un sans paramètre met le temps à 0 et le second prend pour paramètre les valeurs initiales des nombres d'heures, minutes et secondes.

Coureur Les coureurs participant à une course sont définis par la donnée, à la création, de

▷ leurs nom et prénom représentés par des chaînes de caractères,

▷ leur numéro de licence (supposé unique pour chaque coureur[1]), une chaîne de caractères,

▷ leur âge en années, un entier.

On souhaite pouvoir accéder à ces informations mais ne pas pouvoir les modifier.

Q 2 . Donnez le diagramme UML pour une telle classe `Coureur`.

Liste de coureurs On souhaite disposer d'une classe `ListeCoureurs` qui permette de gérer une liste de coureurs et qui respecte (et complète) le diagramme UML ci-dessous, seuls les attributs ont été omis, **à vous de les proposer**.
Cette classe utilise la classe `CoureurIterator` dont le code vous est fourni, ses méthodes sont similaires à celles définies dans l'interface `java.util.List` mis à part le typage des éléments en `Coureur`.

ListeCoureurs
...
+ListeCoureurs() +add(c:Coureur) +remove(c:Coureur):boolean +size():int +iterator():CoureurIterator

[1] aucune vérification sur ce point n'est demandée

```
package janvier2005.sujet;
import java.util.Iterator;
import janvier2005.solution.Coureur;
public class CoureurIterator {
   private Iterator iterator;
   /**
   * @param un itérateur sur un objet de type List que l'on supposera
   *         ne contenir que des objets Coureur
   */
   public CoureurIterator(Iterator iterator) {
      this.iterator = iterator;
   }
   public boolean hasNext() {
      return this.iterator.hasNext();
   }
   public Coureur next() {
      return (Coureur) this.iterator.next();
   }
}
```

Q 3 . Donnez le code JAVA de la classe `ListeCoureurs`.

Course à étapes On suppose donnée la classe `janvier2005.sujet.Etape` qui dispose des 3 méthodes suivantes :

```
/**
* simule le déroulement de l'étape courue par les coureurs de la liste lc :
* après appel de cette méthode l'étape est considérée comme courue.
* @param lc liste des coureurs prenant le départ de l'étape
*/
public void disputer(ListeCoureurs lc) { ...  }
/**
* @return une table associant à chaque coureur finissant l'étape
*         son résultat dans l'étape.
* @exception janvier2005.sujet.EtapeNonCourueException si l'étape n'a pas
*         encore été courue */
public Map resultats() throws EtapeNonCourueException { ...  }
/**
* @return la liste des coureurs ayant abandonné au cours de l'étape
* @exception janvier2005.sujet.EtapeNonCourueException si l'étape n'a pas
*         encore été courue */
public ListeCoureurs abandons() throws EtapeNonCourueException { ...  }
```

Les résultats d'un coureur sont donnés par une instance de la classe `Resultat` également supposée donnée et dont le diagramme de classe est donné ci-dessous. La méthode `add(Resultat r)` permet de cumuler à l'objet `Resultat` sur lequel cette méthode est invoquée les scores définis dans `r` :

janvier2005::sujet::Resultat
...
+Resultat(t:Temps, pointsVert :int, pointsMontagne :int) +getTemps():Temps +getPointsVert():int +getPointsMontagne():int +add(r:Resultat)

Une course à étapes est définie par

- ▷ la liste des étapes qui la composent, ordonnées selon leur déroulement dans la course,
- ▷ la donnée des coureurs qui participent à cette course ainsi que leur résultat : cette information est rangée dans une table associant à un coureur son résultat.

Q 4 . Problème de "`Map`" : Pour que l'objet `Map` résultat d'un appel à la méthode `resultats` de `Etape` puisse être correctement exploité, il est nécessaire de compléter la classe `Coureur`. Sans réécrire la classe `Coureur`, **donnez le code JAVA** correspondant.

Q 5 . On souhaite la classe `TourDeFrance` telle que :

- ▷ le constructeur prenne en paramètre une liste d'étapes et une liste de coureurs, chaque coureur aura initialement un temps de 0h0mn0s et 0 point dans chacun des classements.
- ▷ l'on ait une méthode `disputeCourse`, sans paramètre, qui fait disputer successivement chacune des étapes aux coureurs.
- ▷ l'on ait une méthode `disputeEtape` qui prend en paramètre une étape, fait disputer cette étape à tous les coureurs et modifie leurs résultats en conséquence. Les coureurs ayant abandonné en cours d'étape sont éliminés et donc supprimés des coureurs participant à la course.
- ▷ l'on ait une méthode `abandons`, sans paramètre, qui retourne la liste des coureurs ayant abandonné pendant les étapes déjà disputées. La liste retournée est vide si aucun abandon n'a encore eu lieu.
- ▷ l'on ait une méthode privée qui fournisse une instance de `ListeCoureurs` contenant les coureurs encore en course.

Q 5.1. Donnez votre avis (concisément) sur le statut `public` ou `private` de la méthode `disputeEtape`.

Q 5.2. Donnez le code JAVA nécessaire à la classe `TourDeFrance`.

Classements Un classement correspond au tri d'une liste de coureurs selon le critère choisi. La classe `java.lang.Collections` possède la méthode statique suivante :

```
public static void sort(List l, Comparator comp)
```

qui trie, en la modifiant, la liste `l` selon la relation d'ordre définie par l'objet `comp` de type `Comparator`. `Comparator` est défini ainsi[2] (la javadoc de la méthode `compare` est donnée en annexe) :

```
package java.util;
public interface Comparator {
   public int compare(Object o1, Object o2);
}
```

Q 6 . Tri de liste de coureurs :

Q 6.1. Expliquez clairement et concisément pourquoi la méthode `sort` ci-dessus ne peut prendre en premier argument une instance de la classe `ListeCoureurs`.

Q 6.2. Pour contourner ce problème on décide de définir, dans la classe `ListeCoureurs`, la méthode de signature :

```
public ListeCoureurs tri(Comparator comp)
```

qui retourne une version triée de la liste sur laquelle cette méthode est invoquée.

Donnez un code JAVA pour cette méthode.

Q 7 . Afin de pouvoir effectuer les classements, il est nécessaire de disposer d'objet de type `Comparator` :

- ▷ pour le maillot jaune ou le meilleur jeune, ce comparateur doit définir une relation d'ordre sur les coureurs selon les temps croissants, on appellera `CompateurTemps` ce comparateur,
- ▷ pour le maillot vert et le maillot à pois rouges, chacun des comparateurs doit définir une relation d'ordre sur le coureurs selon des nombres de points croissants, respectivement les "points verts" et les "points montagne", on appellera respectivement `ComparateurVert` et `ComparateurMontagne` ces comparateurs.

Afin de pouvoir effectuer leur comparaison, les constructeurs de ces objets comparateurs prendront en paramètre une table associant des coureurs à leurs résultats.

Q 7.1. Donnez le diagramme UML liant `Comparator` et les trois comparateurs décrits ci-dessus.

Q 7.2. Donnez le code JAVA de `ComparateurTemps`.

Q 7.3. Donnez le code JAVA de `ComparateurVert`.

Q 8 . On ajoute à la classe `TourDeFrance` les méthodes suivantes dont vous **donnerez le code JAVA** :

Q 8.1. `public Coureur maillotJaune()` qui renvoie le coureur détenteur du maillot jaune. En cas d'égalité, n'importe quel coureur meilleur temps conviendra.

[2] On utilisera cette version pour ce sujet.

Q 8.2. `public Coureur meilleurJeune()` qui renvoie le meilleur jeune parmi les coureurs en course. En cas d'égalité, n'importe quel coureur meilleur temps conviendra. Si aucun coureur de la course ne convient (ie. pas de coureur de moins de 25 ans en course) cette méthode lève une exception `NoSuchElementException`.

Q 8.3. `public Coureur maillotVert()` qui renvoie le coureur détenteur du maillot vert. En cas d'égalité, n'importe quel coureur en tête du classement des sprinters conviendra.

Annexe

interface `java.lang.Comparable` : `public int compareTo(Object o)`

```
Compares this object with the specified object for order. Returns  a
negative integer, zero, or a positive integer as this object is less than,
equal to, or greater than the specified object.

The implementor must ensure sgn(x.compareTo(y)) ==
-sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must
throw an exception iff y.compareTo(x) throws an exception.)

The implementor must also ensure that the relation is transitive:
(x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.

Finally, the implementer must ensure that x.compareTo(y)==0 implies that
sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all  z.

It is strongly recommended, but not strictly required that
(x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that
implements the Comparable interface and violates this condition should
clearly indicate this fact. The recommended language is "Note: this class
has a natural ordering that is inconsistent with equals."

Parameters:
    o - the Object to be compared.
Returns:
    a negative integer, zero, or a positive integer as this object is less
    than, equal to, or greater than the specified object.
Throws:
    ClassCastException - if the specified object's type prevents
    it from being compared to this Object.
```

interface `java.util.Comparator` : `public int compare(Object o1, Object o2)`

```
Compares its two arguments for order. Returns a negative integer, zero, or a
positive integer as the first argument is less than, equal to, or greater
than the second.

The implementor must ensure that sgn(compare(x, y)) == -sgn(compare(y, x))
for all x and y. (This implies that compare(x, y) must throw an exception if
and only if compare(y, x) throws an exception.)

The implementor must also ensure that the relation is transitive:
((compare(x, y)>0) && (compare(y, z)>0)) implies compare(x, z)>0.

Finally, the implementer must ensure that compare(x, y)==0 implies that
sgn(compare(x, z))==sgn(compare(y, z)) for all z.

It is generally the case, but not strictly required that (compare(x, y)==0)
== (x.equals(y)). Generally speaking, any comparator that violates this
condition should clearly indicate this fact. The recommended language is
"Note: this comparator imposes orderings that are inconsistent with equals."

Parameters:
    o1 - the first object to be compared.
    o2 - the second object to be compared.
Returns:
    a negative integer, zero, or a positive integer as the first
    argument is less than, equal to, or greater than the second.
```