UE Conception Orientée Objet

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. Les classes ou interfaces devront appartenir à un paquetage `tourdefrance`.

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 (plus petit temps cumulé 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.
Il doit être possible de comparer 2 temps entre eux et d'ajouter un temps à un autre.

Q 1 . Donnez la définition d'une classe `Temps` respectant ces contraintes. Cette classe implémentera l'interface `java.lang.Comparable` 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.

Pour le second constructeur les données de l'instance `Temps` créé sont adaptées pour respecter les contraintes. Ainsi un objet créé à partir des valeurs (h,m,s)=(0,63,0) aura pour données (1,03,0).

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.

Q 2 . Donnez la définition d'une telle classe `Coureur`.

Résultats Les résultats d'un coureur sont donnés par une instance de la classe `Resultat` dont le diagramme de classe est donné ci-dessous. La méthode `cumule(Resultat r)` permet de cumuler aux données de l'instance de `Resultat` sur lequel cette méthode est invoquée les données définies dans l'objet `r` :

tourdefrance::Resultat
...
+Resultat(t:Temps, pointsVert :int, pointsMontagne :int) +getTemps():Temps +getPointsVert():int +getPointsMontagne():int +cumule(r:Resultat)

Etapes Une étape est caractérisée par un numéro d'ordre dans la course et ne peut être disputée (courue) qu'une seule fois. Au cours de cette étape les résultats de chaque coureur ayant terminé l'étape sont établis ainsi que la liste des coureurs ayant abandonné au cours de l'étape.
On suppose disposer de la classe `tourdefrance.Etape` qui dispose, au minimum, 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(List<Coureur> lc) { ...  }
/**
```

[1] aucune vérification sur ce point n'est demandée. Cependant comment pourrait-on faire pour garantir la production de numéros de licence tous distincts ?

```
* @return une table associant à chaque coureur finissant l'étape
*          son résultat dans l'étape.
* @exception tourdefrance.EtapeNonCourueException si l'étape n'a pas
*          encore été courue
*/
public Map<Coureur,Resultat> resultats() throws EtapeNonCourueException { ... }
/**
* @return la liste des coureurs ayant abandonné au cours de l'étape
* @exception tourdefrance.EtapeNonCourueException si l'étape n'a pas
*          encore été courue
*/
public List<Coureur> abandons() throws EtapeNonCourueException { ... }
```

Q 3 . Que faut-il impérativement définir dans la classe `Coureur`, pour que l'objet `Map` renvoyé par un appel à la méthode `resultats` de `Etape` puisse être correctement exploité ?

Donnez le code associé à votre réponse.

Course à étapes 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 . On souhaite définir 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 0h, 0mn, 0s et 0 point dans chacun des classements,
- ▷ l'on ait une méthode `disputeCourse` qui fait disputer successivement et dans l'ordre chacune des étapes aux coureurs,
- ▷ l'on puisse récupérer la liste des coureurs encore en course ainsi que la liste de ceux ayant abandonné pendant les étapes déjà disputées.

Définissez une telle classe.

Classements Un classement correspond au tri d'une liste de coureurs selon un critère choisi. Comme indiqué précédemment plusieurs classements sont établis en fonction des temps réalisés et différents points acquis.
On souhaite donc ajouter à la classe `TourDeFrance` les méthodes suivantes :

- `public List<Coureur> classementGeneral()` qui renvoie la liste des coureurs encore en course triée par ordre croissant des temps réaliés dans les étapes déjà parcourues.
- `public Coureur maillotJaune()` qui renvoie le coureur détenteur du maillot jaune. En cas d'égalité, n'importe quel coureur meilleur temps conviendra.
- `public Coureur meilleurJeune()` qui renvoie le meilleur jeune (moins de 25 ans) parmi les coureurs en course. En cas d'égalité entre deux jeunes, n'importe quel coureur jeune meilleur temps conviendra. Si aucun coureur de la course n'est jeune, cette méthode lève une exception `NoSuchElementException`.
- `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.
- `public Coureur maillotMontagne()` qui renvoie le coureur détenteur du maillot à pois rouges. En cas d'égalité, n'importe quel coureur en tête du classement des grimpeurs conviendra.

On rappelle que la classe `java.util.Collections` possède les méthodes statiques suivantes qui peuvent s'avérer utiles :

```
public static void sort(List<T> list, Comparator<?  super T> comp)
```

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

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

Q 5 . Faites une proposition d'implémentation de ces méthodes évitant au maximum la répétition de code.

Q 6 . On pourra ajouter à `TourDeFrance` une méthode `palmares` qui permet l'affichage des coureurs détenteurs de chacun des maillots.

Travaux pratiques

Il s'agit de coder les classes définies précédemment en commençant par la documentation et les tests.
Vous utiliserez les codes fournies sur le portail. En particulier les classes

- `tourdefrance.util.InitListeCoureurs` qui permet de créer une liste de coureurs à partir des informations contenues dans un fichier (voir la méthode `main` pour les tests),
- `Etape` (et sa classe de test) qui correspond au cahier des charges fournis ci-dessus, en particulier la méthode `disputer` a été codée de manière à simuler le déroulement de l'étape à partir des informations fournies dans un fichier (voir la méthode `main` pour un exemple),
- `EtapeNonCourueException` qui est la classe d'exception annoncée dans le sujet,
- `TourDeFranceMain` qui crée la liste des coureurs et les étapes (en conformité avec les fichiers de données ci-dessous),

et les fichiers de données :

- `data/coureurs.txt` qui fournit les informations pour la création des coureurs participant au tour de france,
- `data/etape?.txt` qui contiennent les données qui permettent la simulation des étapes de numéro `?` pour les coureurs du fichier précédent.

Vous pouvez bien évidemment définir d'autres fichiers de données en plus de ceux fournis[2].
Vous réaliserez le travail en binôme en vous répartissant préalablement (et librement) le travail de codage des différents éléments (classes et/ou méthodes) du programme.
Vous procéderez ainsi pour chaque élément que doit coder *binome_1* (respectivement *binome_2*) :

1. *binome_1* écrit la javadoc,
2. *binome_2* écrit les tests,
3. *binome_1* écrit le code et effectue les tests.

L'entête de chaque fichier précisera le nom du binôme auteur du code.
Rendez par binôme via `PROF` une archive respectant le format demandé. Le nom du fichier d'archive rendu contiendra les logins des deux binômes.

Annexe

interface `java.lang.Comparable<T>` : `public int compareTo(T 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.
```

[2]qui ne doivent pas être modifiés pour permettre le bon fonctionnement des tests de `tests.EtapeTest`.

interface java.util.Comparator<T> :

public int compare(T o1, T 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.