UE Programmation Orientée Objet

Devoir Surveillé

Mardi 14 mai 2013 – 16h30-18h30
Copies des diapositives de cours autorisées
Une feuille recto-verso de notes personnelles
Dictionnaire de langue (papier ou électronique “dédié”) autorisé
Tout autre document interdit

- *La classe* `Horaire` *définie dans le premier exercice est utilisée dans le second. Il n'est cependant pas nécessaire d'avoir traité le premier exercice pour faire le second, il suffit d'utiliser les informations fournies dans l'énoncé. Une simple lecture attentive de l'énoncé de l'exercice 1 suffit donc pour traiter l'exercice 2.*
- *L'exercice 3 est totalement indépendant des deux premiers.*

Exercice 1 : Horaires.

On appelle *horaire* une donnée caractérisant un moment dans une journée, indépendamment de la date de cette journée. Un *horaire* permet par exemple de préciser le moment du départ d'un train : “*le train de 17h42*” où *17h42* représente l'horaire du train évoqué. Nous allons représenté un *horaire* par une instance de la classe `util.Horaire`.

Une instance de `Horaire` est donc représentée par la donnée de deux nombres : le premier représente l'heure (entre 0 et 23) et le second les minutes (entre 0 et 59).

La classe `Horaire` implémente l'interface `Comparable` (voir en annexe) en réalisant la relation d'ordre intuitive sur ces horaires. Elle propose en plus les méthodes suivantes :

- les accesseurs pour les heures et les minutes ;
- la méthode `equals`, deux horaires étant égaux s'ils ont mêmes heures et mêmes minutes ;
- une méthode

  ```
  public int ecart(Horaire h)
  ```

  dont le résultat est l'écart en minutes entre l'objet horaire invoquant la méthode et l'horaire `h` passé en paramètre. Cette méthode déclenche une exception `IllegalArgumentException` si l'horaire `h` est plus tôt (“dans la journée”) que l'horaire invoquant.

Q 1 . Donnez le diagramme UML détaillé de la classe `Horaire`.

Q 2 . Donnez le code java de la méthode `equals`.

Q 3 . Donnez la javadoc et le code java de la méthode `ecart`.

Q 4 . Pour **toutes** les méthodes de votre classe `Horaire` autres que les accesseurs et le constructeur donnez le code des **méthodes de test** que vous proposez.

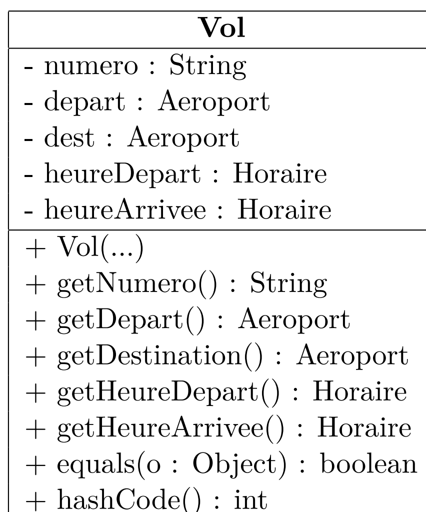
Seules les codes des méthodes de test sont demandés, pas les autres éléments de la classe de test.

Exercice 2 : Aéroport et vols.

- *Dans cet exercice nous utiliserons la classe* `util.Horaire` *définie à l'exercice précédent (cf. introduction au sujet).*
- *La javadoc et le code des méthodes de test ne sont pas demandés dans cet exercice.*

On s'intéresse à la représentation de vols entre aéroports. Les vols sont supposés être tous journaliers et il s'agit de “vols courts” donc les horaires de départ et d'arrivée sont toujours dans la même journée. On ne s'occupe donc pas des jours de départ, seul l'horaire compte

Vols. Un vol est caractérisé par un numéro unique (une chaîne de caractères), un horaire de départ et un horaire d'arrivée (de type `Horaire`), les aéroports de départ et de destination. Voici le diagramme UML de la classe `avion.Vol` qui permet de représenter ces données :



Aéroports. Un aéroport est caractérisé par un identifiant unique (une chaîne de caractères) et une table de hachage qui associe pour chacun des vols qui partent de cet aéroport, le numéro de vol à l'objet vol correspondant. Deux objets `Aeroport` sont donc égaux s'ils ont le même identifiant (on supposera que dans ce cas la liste des vols est nécessairement la même pour les deux objets).
Dans les questions suivantes si pour résoudre une question vous pensez qu'il manque une méthode, vous en donnerez le code en précisant brièvement pourquoi vous ajoutez cette méthode.

Q 1 . Donnez le code java de l'entête de la classe `avion.Aeroport` ainsi que la déclaration de ses attributs et de son constructeur sachant qu'initialement la table des vols est vide.

Q 2 . Donnez le code d'une méthode `ajouteVol` qui prend en paramètre un vol et l'ajoute à cet aéroport. Cette méthode déclenche une `IllegalArgumentException` si cet aéroport n'est pas l'aéroport de départ de ce vol.

Q 3 . Donnez le code java d'une méthode `volsDirects` qui prend en paramètre un aéroport destination `dest`. Cette méthode retourne la liste des vols qui partent de cette instance d'aéroport et dont la destination est `dest`.

Q 4 . Donnez le code java d'une méthode `prochainVolDirect` qui prend en paramètre un aéroport destination `dest` et un horaire `h` et dont le résultat est le numéro du premier vol de cette instance d'aéroport qui part dans la journée pour `dest` après l'horaire `h` fourni. Une exception `NoSuchElementException` est déclenchée si aucun vol ne convient.

Q 5 . On ajoute à la classe `Horaire` la méthode suivante :

```
/** renvoie l'horaire correspondant à l'instant courant
* @return l'horaire correspondant à l'instant courant
*/
public static Horaire maintenant() {
   ...
}
```

Donnez le code java du corps de la méthode suivante :

```
/** renvoie true ssi le prochain vol direct à destination de dest
*     part dans moins de delai minutes de cet aéroport
* @param dest la destination
* @param delai le délai maximal avant le départ du prochain vol pour dest
* @return true si le prochain vol direct à destination de dest
```

```
    *     part dans moins de delai minutes de cet aéroport et
    *     false si ce n'est pas le cas ou s'il n'y pas de prochain vol direct
    */
    public boolean volEnPartance(Aeroport dest, int delai) {
       ...
    }
```

Q 6 . Donnez le code java d'une méthode `volsParDestination` dont le résultat est la table de hachage qui pour cet (`this`) aéroport associe à chaque aéroport `a` la liste des vols directs partant de `this` dont `a` est la destination.

Exercice 3 : Transformations de textes.

- *La javadoc et le code des méthodes de test ne sont pas demandés dans cet exercice.*
- *Tous les types de cet exercice appartiennent au paquetage* `texte`

Dans cet exercice les textes seront représentés par des objets de la classe `String` et on s'intéresse à des opérations de transformation de ces textes. Une opération de transformation de texte est caractérisée par l'interface suivante :

```
package texte;
public interface Transformation {
   /** applique la transformation au texte fourni en paramètre
    * @param txt le texte à transformer (texte initial)
    * @return le texte transformé
    */
   public String transforme(String txt);
}
```

Q 1 . Donnez le code d'une opération de transformation qui a pour effet de passer en majuscules tous les caractères du texte initial.

On rappelle l'existence de la `public String toUpperCase()` dans `java.lang.String` dont le résultat est la chaîne dont tous les caractères ont été transformés en majuscules.

D'autres transformations de textes sont facilement envisageables, telles que supprimer les sigbnes de ponctuation d'un texte, appliquer un codage de César au texte, etc.

Q 2 . Une chaîne de transformation de textes consiste à appliquer séquentiellement plusieurs traitements en cascade sur un même texte. On peut par exemple transformer le texte en majuscules et (puis) en supprimer la ponctuation et ensuite appliquer un codade de César sur ce texte.

Donnez le code d'une classe `ChaineTransformations` qui dispose des méthodes

- pour ajouter une tranformation à la fin de la chaîne ;
- pour transformer le texte, par application séquentielle de chacune des transformations de la chaînes.

Annexe

`java.lang`
`public interface Comparable<T>`

compareTo

```
int compareTo(T o)
```

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.