Examen

3 heures - documents écrits autorisés
jeudi 29 janvier 2004

- il est possible d'utiliser les réponses à une question non traitée pour résoudre les autres questions.

Exercice 1 : Location de voitures

(Les classes/interfaces de cet exercice seront à ranger dans un paquetage `agence`.)

Une agence de location de voitures offre à ses clients la possibilité de choisir la voiture louée en fonction de différents critères.
Les voitures sont définies par une marque, un nom de modèle, une année de production et un prix de location à la journée. Pour simplifier les deux premiers paramètres seront des objets de la classe `String` et les deux derniers seront des `int`. Deux voitures sont considérées égales si tous leurs attributs sont égaux.

Q 1 . Donner le code de la classe `Voiture`

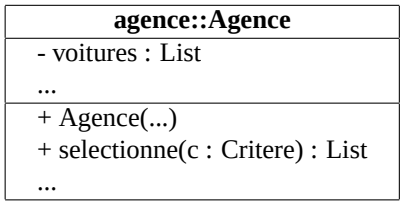
Il est possible de sélectionner dans la liste des voitures à louer toutes les voitures satisfaisant un critère donné.
On définit l'interface `Critere` ainsi :

```
public interface Critere {
  /** @return true si et seulement si l'objet o est conforme au
   * critère (on dit que o satisfait le critère)
  */
  public boolean correspond(Object o);
}
```

Q 2 . Donnez le code d'une classe `CritereMarque` qui est un critère satisfait par toutes les voitures d'une marque donnée, précisée à la construction du critère.

Q 3 . Donnez le code d'une classe `CriterePrix` qui est un critère satisfait par toutes les voitures dont le prix est inférieur à un prix fixé à la construction du critère.

Q 4 . On suppose une classe `Agence` définie (au minimum) ainsi :



Donnez le code de la méthode `selectionne` qui sélectionne parmi toutes les voitures de l'agence (contenues dans l'attribut `voitures`) celles qui satisfont le critère donné.

Q 5 . En supposant que la référence `agence` est de type `Agence` et a été initialisée, donnez la ou les lignes de code permettant de sélectionner toutes les voitures de cette agence dont le prix est inférieur à `100`.

Q 6 . On peut naturellement souhaiter faire des intersections c'est-à-dire appliquer le et logique, de critères (on pourrait de même souhaiter des unions de critères). On obtient alors un nouveau critère satisfait si tous les critères qui le compose sont satisfaits.

Définissez une classe `InterCritere` qui permet de définir des critères par intersection d'un nombre quelconque de critères. On demande à pouvoir ajouter un nouveau critère à l'intersection mais pas de pouvoir en retrancher.

Q 7 . Donnez la ou les lignes de code permettant de créer un critère intersection d'un critère pour la marque `"Timoléon"` et d'un critère pour un prix inférieur à `100`.

Exercice 2 : Occurrences

(Les classes/interfaces de cet exercice seront à ranger dans un paquetage `occurrence`.)

On définit une classe `Texte` disposant d'un attribut `leTexte`, de type `String`, initialisé par le constructeur.
On définit le type `CompteurOccurrences`, comme le type des objets permettant de compter des occurrences d'éléments dans une instance de `Texte`. Ce calcul se fait par une méthode

```
public Map calculOccurrences(Texte txt)
```

qui renvoie la table associant à chaque élément compté, son nombre d'occurrences trouvé dans l'objet `txt` passé en paramètre.
Pour notre problème, on souhaite disposer de deux sortes de `CompteurOccurrences`, un compteur d'occurrences où les éléments sont les caractères du texte, et un compteur où les éléments sont les mots.

Q 1 . Que proposez vous pour la définition du type `CompteurOccurrences` ?
Donnez le code associé à votre proposition.

Q 2 . Donnez le code de la méthode `afficheOccurrences` de la classe `Texte` fournie en annexe.

Cette méthode demande à l'objet `CompteurOccurrences` passé en paramètre de calculer la table des occurrences pour son attribut `leTexte`, puis affiche pour chacun des éléments trouvé son nombre d'occurrences, à raison d'un élément par ligne.

Q 3 . Donnez le code permettant un calcul d'occurrences de caractères dans l'attribut `leTexte` d'un objet `Texte` : les clés sont les caractères (ascii) apparaissant dans `leTexte` et les valeurs associées les nombres d'occurrences de chacun de ces caractères dans cette chaîne.
Rappel : la méthode `charAt(index i)` permet de connaître le "`char`" à la position i d'une chaîne de caractères stockée dans un objet `String`.

Q 4 . Donnez le code permettant un calcul d'occurrences de mots dans l'attribut `leTexte` d'un objet `Texte` : les clés sont les mots apparaissant dans `leTexte` et les valeurs associées les nombres d'occurrences de chacun de ces mots dans cette chaîne.
On appelle mot, toute séquence de caractères comprises entre deux délimiteurs, nous considérerons comme délimiteurs les éléments de la chaîne suivante : `".,; :!?"` (y compris l'espace).
Pour votre calcul vous utiliserez la classe `java.util.StringTokenizer` qui permet le découpage en tokens (*token* est synonyme de *mot* pour notre exercice ici) d'une chaîne de caractères. Un extrait de la javadoc de cette classe est donné en Annexe.

Q 5 . Ecrire un "`main`" qui crée un objet de type `Texte` (vous mettrez `"···"` comme argument), puis fait afficher le nombre d'occurrences de chaque caractère du texte, puis de chaque mot du texte.

Exercice 3 : Bataille !

(Les classes/interfaces de cet exercice seront à ranger dans un paquetage `bataille`.)

Il s'agit de coder le déroulement (automatique) de parties du jeu de bataille.
Rappelons en brièvement les règles (au cas où ...) : le jeu se joue à 2 joueurs qui disposent chacun d'un tas de cartes. Ce tas est une file (FIFO). À chaque tour de jeu, chacun des joueurs prend la première carte de son tas (début de sa file) et la pose sur la table. Si les valeurs des 2 cartes sont différentes (on ne tient pas compte des couleurs dans ce jeu), le joueur qui a mis la carte de plus forte valeur[1] ramasse (toutes) les cartes qui sont sur la table et les met (peu importe dans quel ordre) sous son tas (à la fin de sa file). Si les deux cartes sont de même valeur, il y a "bataille", dans ce cas, chacun des joueurs pose la première carte de son tas sur la table. On recommence alors un nouveau tour de jeu en laissant les cartes sur la table et en appliquant à nouveau les règles précédentes. Le prochain qui gagne un tour de jeu récupère toutes les cartes qui ont été posées sur la table (et les place sous son tas). Dès qu'un joueur ne peut plus jouer parce qu'il n'a plus de carte dans son tas, il a perdu.

Q 1 . La classe `Carte` :

Q 1.1. Définissez la classe `Couleur` permettant de gérer les quatre couleurs : *cœur*, *carreau*, *trèfle* et *pique*.

On supposera définie de manière similaire la classe `Valeur` gérant les valeurs des cartes et implémentant l'interface `java.lang.Comparable`.

Q 1.2. Définir la classe `Carte`, une carte est définie par une valeur et une couleur. Cette classe doit implémenter l'interface `java.lang.Comparable`.

Q 2 . Définissez une classe permettant de gérer les tas de cartes.

Vous utiliserez la classe `java.util.LinkedList` dont un extrait de la javadoc est donnée en annexe.

Q 3 . Définissez une méthode `joue` pour une classe `Bataille` qui permet d'exécuter le déroulement d'une partie du jeu de bataille en supposant que les tas des deux joueurs ont été initialisés.

[1] L'ordre des valeurs des cartes de la plus forte à la moins forte est : As, Roi, Dame, Valet, 10, 9, 8, 7, etc.

Annexes

La classe `Texte`

```
package occurrence;
public class Texte {
  private String leTexte;
  public Texte(String texte) {
    this.leTexte = texte;
  }
  public String getLeTexte() {
    return leTexte;
  }
  public void afficheOccurrences(CompteurOccurrences cpteurOcc) {
    // ... à compléter
  }
}
```

Extrait de javadoc pour `java.util.StringTokenizer`

`public StringTokenizer(String str, String delim, boolean returnDelims)`

Constructs a string tokenizer for the specified string. All characters in the delim argument are the delimiters for separating tokens.

If the `returnDelims` flag is `true`, then the delimiter characters are also returned as tokens. Each delimiter is returned as a string of length one. If the flag is `false`, the delimiter characters are skipped and only serve as separators between tokens.

`int countTokens()` Calculates the number of times that this tokenizer's `nextToken` method can be called before it generates an exception.

`boolean hasMoreTokens()` Tests if there are more tokens available from this tokenizer's string.

`String nextToken()` Returns the next token from this string tokenizer.

Extrait de javadoc pour `java.util.LinkedList`

`java.util.LinkedList implements List ...`

`Object getFirst()` Returns the first element in this list.

`Object getLast()` Returns the last element in this list.

`Object removeFirst()` Removes and returns the first element from this list.

`Object removeLast()` Removes and returns the last element from this list.

`void addFirst(Object o)` Inserts the given element at the beginning of this list.

`void addLast(Object o)` Appends the given element to the end of this list.