UE Programmation Orientée Objet

Examen mai 2007

3 heures
documents de cours, TD et TP autorisés
livres, portables et calculatrices interdits

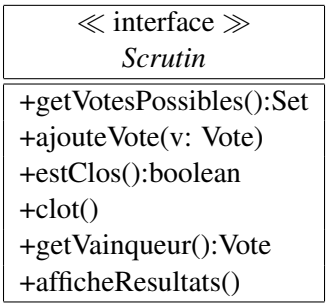
▷ Les 2 exercices sont totalement indépendants et l'ordre dans lequel ils apparaissent ne préjuge pas de leur difficulté réelle ou supposée.

Exercice 1 : Scrutins électoraux

On s'intéresse à la programmation de différents modes de scrutins électoraux.
Un scrutin consiste pour des électeurs à choisir un vote parmi une liste de votes possibles. Il est possible de voter tant qu'un scrutin n'est pas clos. Ensuite, une fois que celui-ci est clos, il est possible de déterminer le résultat de ce scrutin. Le calcul du résultat, et notamment du vainqueur, dépend du type de scrutin. Dans ce sujet 2 modes de scrutin seront considérés :

- le scrutin à la majorité des suffrages exprimés (type `ScrutinMajoritaire`) : dans ce cas est déclaré vainqueur le vote qui aura obtenu plus de 50% des votes autres que les votes blancs ou nuls.
- le scrutin à majorité relative (type `ScrutinRelatif`) : est déclaré vainqueur le vote (autre que les votes blancs ou nuls) qui a reçu le plus de suffrages pour peu qu'au moins 15% des inscrits au scrutin aient exprimé ce vote.

Il peut bien sûr n'y avoir aucun vainqueur, dans chacun de ces cas.
Un scrutin est défini par l'interface suivante du paquetage `scrutin` :



- La méthode `getVotesPossibles` retourne l'ensemble des votes possibles pour ce scrutin.
- La méthode `ajouteVote(v)` ajoute le vote `v` à la liste des votes effectués pour le scrutin (on ne s'occupe pas ici des votants ni du fait que chacun ne peut voter qu'une fois, ceci est supposé géré par ailleurs).
- La méthode `ajouteVote` accepte tous les votes tant que le scrutin n'est pas clos, que le vote fasse partie des votes autorisés (ou "possibles") ou non.
- La méthode `ajouteVote` lève une exception `ScrutinClosException` si le scrutin est clos. Les méthodes `getVainqueur()` et `afficheResultats()` lèvent une exception `ScrutinNonClosException` dans le cas contraire.
  Dans les 3 cas, si une exception est levée, la méthode n'a aucun autre effet.
- La méthode `afficheResultats` affiche pour chaque vote possible, ainsi que les votes `NUL` et `BLANC` le nombre de suffrage reçus, une fois le scrutin clos.

- La méthode `clot` clôt le scrutin **et** établit le résultat du scrutin. Les résultats sont mémorisés sous la forme d'une **table de hachage** associant à un objet `Vote` le nombre de suffrages qu'il a reçus. Si l'un des votes ne fait pas partie des votes autorisés par le scrutin alors il est considéré comme un vote `NUL`.
- Le résultat de `getVainqueur` est `null` s'il n'y a aucun vainqueur.

La suite de ce sujet s'intéresse à divers aspects d'une telle application. Toutes les classes que vous définirez devront appartenir au paquetage `scrutin`.

Votes

Q 1 . Donnez le code d'une classe `Vote`. Un vote est défini par le texte (ou nom) représentant ce vote, évidemment non modifiable. Deux votes portant le même texte sont naturellement égaux. Pensez à prévoir qu'il faut gérer la table des résultats (voir description de la méthode `clot` ci-dessus). La méthode `toString` doit également être définie.

Enfin, il existe en plus 4 constantes correspondant à des votes particuliers : les votes `OUI`, `NON`, `BLANC` et `NUL`.

Scrutins

Q 2 . Donnez le code de la classe `ScrutinMajoritaire`, sachant que le nombre d'inscrits (électeurs) et l'ensemble des votes soumis au scrutin sont communiqués à la création d'une instance. A ces votes doivent systématiquement être ajoutés les votes `BLANC` et `NUL`.

Q 3 . En supposant que le reste de la classe reste tout à fait similaire à `ScrutinMajoritaire`, donnez pour la classe `ScrutinRelatif` le code nécessaire à la méthode `getVainqueur`.

Q 4 . Créez une classe contenant comme seule méthode une méthode `main` qui :

1. crée un référendum, qui est un *scrutin majoritaire* dans lequel les votes possibles sont les votes `OUI` et `NON`, on fixera le nombre de votants à 110,
2. ajoute 100 votes choisis aléatoirement entre `OUI` et `NON`,
3. clôt le scrutin et annonce le vainqueur.

Exercice 2 : Compression d'images

On s'intéresse à la représentation d'images rectangulaires sous la forme d'une matrice de pixels. Un pixel (pour "*pic*ture *el*ement") est notamment caractérisé par sa couleur. Ces couleurs seront représentées par des objets de la classe `java.awt.Color` dont la méthode `equals` se comporte de la manière attendue. Une image sera donc caractérisée par un tableau à 2 dimensions d'objets `Color`.
On va en particulier mettre en œuvre une technique simple de compression de telles images.
La classe `Image` dont il est question est définie ainsi :

Image
- lesPoints : java::awt::Color[][]
+Image(hauteur : int, largeur : int) +init(nomFichier : String) +compresse() : List +decompresse(l : List, hauteur : int, largeur : int):Image +getLargeur():int +getHauteur():int +getColorAt(x:int,y:int): java::awt::Color +afficheImage(x0 : int, y0 : int)

On ne s'intéressera ni à la méthode `init`, ni à `getColorAt` que l'on supposera définie.
Les méthodes `compresse` et `decompresse` seront décrites aux questions 3 et 4.

Q 1 . Donnez le code du constructeur qui définit une image de `largeur` sur `hauteur` pixels noirs (objet `java.awt.Color.BLACK`).

Q 2 . Donnez le code des méthodes `getLargeur` et `getHauteur` de la classe `Image`.

Q 3 . On suppose qu'existe une classe `Ecran` disposant d'une méthode *statique* `affichePoints(int x, int y, java.awt.Color c)` qui affiche un point de couleur `c` aux coordonnées (`x` ,`y`) de l'écran (le point (0,0) est en haut à gauche de l'écran). Donnez le code de la méthode `afficheImage` dont les paramètres correspondent aux coordonnées du point situé en haut à gauche de l'image et qui affiche l'image à l'écran. On ne s'occupera pas de la sortie éventuelle de l'écran lors de l'affichage de l'image.

Q 4 . Compression. La méthode de compression appliquée est la suivante : on parcourt l'image ligne par ligne, de gauche à droite et de haut en bas, chaque suite de n points de la même couleur c (y compris sur plusieurs lignes) est remplacée par un couple contenant ces 2 informations : (n, c). On représente alors l'image par une liste de tels couples.

On remplace ainsi n objets couleur par un seul objet et un entier. Dans le cas d'images avec de grandes plages de pixels de même couleur on limite la répétition de cette couleur.

Par exemple, l'image ci-dessous, représentant comme chacun le remarquera un soleil **R**ouge sur fond de ciel **B**leu et au dessus d'une plage **J**aune) :

B	**B**	**R**	**R**	**R**	**B**	**B**
B	**R**	**R**	**R**	**R**	**R**	**B**
B	**R**	**R**	**R**	**R**	**R**	**B**
B	**B**	**R**	**R**	**R**	**B**	**B**
J	**J**	**J**	**J**	**J**	**J**	**J**
J	**J**	**J**	**J**	**J**	**J**	**J**
J	**J**	**J**	**J**	**J**	**J**	**J**
J	**J**	**J**	**J**	**J**	**J**	**J**

peut être remplacée par la suite de couples : ((2,**B**) (3,**R**), (3,**B**), (5,**R**), (2,**B**), (5,**R**), (3,**B**), (3,**R**), (2,**B**), (28,**J**)).

Le troisième couple (3,B) se justifie par les 2 points **B**leu finissent la première ligne et celui qui commence la seconde.

Q 4.1. Définissez une classe `Couple` adaptée à la gestion des couples.

Q 4.2. Définissez la méthode `compresse` qui renvoie la liste des couples correspondant à la version compressée de l'objet `Image` sur laquelle elle est invoquée.

Q 5 . On souhaite bien évidemment pouvoir effectuer l'opération de décompression inverse. Cette opération doit permettre de construire une image à partir d'une liste de couples, connaissant `largeur` et `hauteur`.

Cette opération se réalise par la méthode *statique* :

```
public static Image decompresse(List lesCouples, int largeur, int
                                hauteur)
```

qui a pour résultat l'objet `Image` ainsi obtenue.

Si la liste de couples `lesCouples` ne fournit pas suffisamment de points on laissera noirs les points restants. A l'inverse les points éventuellement en trop seront ignorés.

Donnez le code de la méthode `decompresse`.