Classes abstraites et Interfaces

**Exercice 1** : Garage d’automobiles

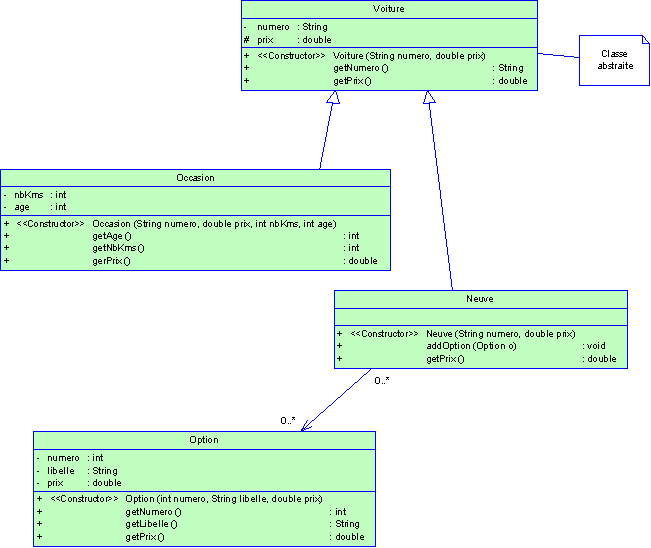
Un garage vend des voitures; chaque voiture possède un numéro d'immatriculation ainsi qu'un prix de vente de base. Deux types de voitures sont proposés:

* Des voitures d'occasion qui ont un certain kilométrage et une ancienneté en années. Le prix de base est modifié selon la règle suivante: le kilométrage normal est de 20000 km par an; le prix est majoré ou minoré à raison de 0.05 euros par km inférieur ou supérieur.
* Des voitures neuves qui peuvent être vendues avec des options - climatisation, airbag,... - chaque option a un numéro, un libellé et un prix. Le prix des options se rajoute au prix de base.

Le garage propose ainsi un parc d'automobiles à la vente, et des éventuelles options pour les voitures neuves.

L'objectif de l'application est de simuler des ventes de voitures.

Voici le diagramme de classes (les méthodes « toString » ne sont pas représentées):



1) Créer une classe « Option » avec son constructeur, ses 3 accesseurs et une méthode « toString »

*Exemple de résultat à obtenir :*

Numero:1 Libelle:Airbag Prix:800.0

Numero:2 Libelle:Toit ouvrant Prix:1300.0

2) Sachant qu’une voiture sera obligatoirement neuve ou d’occasion (on ne peut pas créer d’objet « Voiture », on ne peut créer que des objets « Voiture neuve » ou « Voiture d’occasion »), implémentez la classe abstraite « Voiture » avec son constructeur et son accesseur « getNumero ».

Si on essaye de créer une voiture. Que se passe-t-il ?

3) Ajoutez à votre classe « Voiture » une méthode « getPrix » qui retourne le prix du véhicule. Compte tenu de la différence de règles de calcul entre les voitures d’occasion et les voitures neuves, nous ne pouvons pas calculer le prix dans la classe « Voiture ». La méthode « getPrix » de la classe « Voiture » doit donc être une méthode abstraite.

4) Créer la classe « Occasion » qui hérite de voiture avec son constructeur et 2 accesseurs permettant de retourner le nombre de kilomètres et l’âge de la voiture en années.

5) Ajouter une méthode « getPrix » à votre classe « Occasion » en tenant compte de la règle énoncée au début.

6) Ajouter une méthode « toString » à votre classe « Voiture » qui retourne le numéro d’immatriculation

Exemple

Occasion o1= new Occasion("9874IY94",7000,50000,2);

System.out.println(o1.toString());

*Exemple de résultat à obtenir :*

9874IY94 Nb de km: 50000 anciennete:2 Prix de vente:7500.0

7) Créer la classe «Neuve » qui hérite de voiture avec son constructeur et une méthode qui permet d’ajouter des options

8) Ajoutez une méthode « getPrix » qui retourne le prix de la voiture en tenant compte de toutes ses options.

9) Ajoutez une méthode « toString » dans la classe « Neuve » selon le même principe que la méthode « toString » de la classe « Occasion » puis testez en créant une voiture neuve avec 2 options et en affichant son prix

*Exemple de résultat à obtenir :*

123AZE93 Prix avec options:12100.0

10) Créer une classe Test « Garage » ayant comme données un tableau d’options et un tableau de voitures, Alimenter votre garage en voitures et en options et Affichez la liste des véhicules du garage par ordre de prix

**Exercice 2 :** Editeur de Texte

On souhaite réaliser les éléments de base d’un éditeur de texte.

1. Proposer une classe Position qui représente une position dans un texte. Cette position est constituée du numéro de la ligne considérée (toujours strictement positif) et du numéro du caractère dans cette ligne numéroté à partir de 0).

2. Proposer une classe Curseur qui possède une Position modifiable.

3. On suppose donnée une classe Texte de squelette suivante :

public class Texte {

pubic char getChar(Position p) { }

/\*\*

\* Renvoie le caractère situé en position p.

\* @param p position étudiée

\* @return le caractère

\*/

public String getLigne(int n) { }

/\*\*

/\*\*

\* Renvoie la ligne numéro n sous forme d’une chaine de caractères

\* @param n numéro de la ligne

\* @return la ligne

\*/

public String getSubText(Position begin,Position end) {}

/\*\*

\* Renvoie sous forme d’une chaîne de caractères le texte compris entre la

\* position begin (incluse) et la position end (non incluse)

\* @param begin début du texte

\* @param end fin du texte

\* @return le texte

\*/

public String cutSubText(Position begin,Position end) {}

/\*\*

\* Renvoie sous forme d’une chaîne de caractères le texte compris entre la

\* position begin (incluse) et la position end (non incluse), et supprime ce

\* texte de celui stocké dans l’objet appelant

\* @param begin d´ebut du texte

\* @param end fin du texte

\* @return le texte

\*/

public void insertSubText(Position where,String text) {}

\* Insère à partir de la position where le texte contenu dans le paramètre

\* text. Le premier caractère de ce texte devient celui de position where.

\* @param where emplacement de l’insertion

\* @param text texte à insérer

}

On souhaite réaliser sur une instance de cette classe des modifications produites par un éditeur de texte. Chaque modification implantera l’interface suivante :

public interface Modification {

public void do(Position p,Curseur c,Texte t) ;

}

Le but de la méthode do est de modifier le texte courant et éventuellement la position du curseur. On programmera les modifications suivantes :

1. Avance : cette modification ne transforme pas le texte, elle se contente de faire passer le curseur sur le caractère suivant. Pour ce faire, elle augmente de 1 le numéro du caractère. Si ce numéro devient strictement supérieur au nombre de caractères présents dans la ligne courante, on passe `a la ligne suivante.
2. Recule : même opération, mais en faisant passer le curseur sur le caractère précédent.
3. Insere : cette modification hérite de Avance et prend comme paramètre de création un caractère à insérer dans le texte. Elle insère à la position du curseur le caractère en question, puis avance le curseur d’une position.
4. Backspace : cette modification hérite de recule. Elle fait reculer le curseur d’un cran, puis supprime le caractère situé à la nouvelle position.