Classes, classes abstraites, héritage, itérateurs

Exercice 1 - Plaque d'immatriculation

On veut modéliser des numéros de plaques d'immatriculation en séparant le code du département des autres informations.

1 Écrire une classe `Immatriculation` contenant 2 champs `codeDepartement` et `serie` représentant respectivement le code du département (penser aux codes 2A et 2B pour la Corse) et la séquence de chiffres et de lettres qui le précède.

2 Écrire le constructeur initialisant prenant en paramètres les valeurs des champs précédents, ainsi que les méthodes `toString()`, `getSerie()` et `getCodeDepartement()`.

3 Écrire la méthode `getDepartement()` qui retourne le nom du département en toutes lettres.

Exercice 2 - La classe Voiture

On souhaite maintenant représenter des voitures. Les informations qui nous intéressent sont le numéro d'immatriculation, la marque du véhicule et le nombre de fenêtres de celui-ci, qui nous servira à calculer le montant d'une taxe.

1 Écrire une classe `Voiture` contenant un champ `plaque` de type `Immatriculation`, ainsi que des champs `marque` et `nombreDeFenetres` repsectivement de type `String` et `int`.

2 Écrire les méthodes accesseurs de la classe puis une méthode `toString()` affichant la marque, l'immatriculation et la taxe.

3 Écrire la méthode `getMontantTaxe()` qui calcule le montant de la taxe Alif selon la formule 1000*`nombreDeFenetres`.

Exercice 3 - Généralisation à une classe Vehicule

On veut maintenant représenter 2 types de véhicules: les voitures et les motos. Les informations d'immatriculation et de marque sont utiles pour les deux types. En revanche, les motos n'ont pas de fenêtre et le montant de la taxe pour une moto est de `vitesseMaxi*vitesseMaxi` .

1 En tenant compte des indications précédentes, proposer une architecture de classes permettant de modéliser ces différents objets (penser à l'héritage).

2 Écrire les classes proposées. L'instruction `vehicule.getMontantTaxe()` doit retourner le montant de la taxe correctement calculée, que `vehicule` représente une moto ou bien une voiture.

Exercice 4 - Table dynamique générique

Le but de cet exercice est d'écrire une classe permettant d'ajouter des élements dans une table et de parcourir celle-ci.

1 Écrire la classe `Table`. Pour cela, écrire un constructeur prenant en argument un entier correspondant à la taille maximale de la table, une méthode `add()` permettant d'ajouter

un élement dans la table et une méthode `size()` retournant le nombre d'éléments contenus dans la table.

2. Écrire une méthode `iterator()` renvoyant un objet d'une classe implantant l'interface `java.util.Iterator` permettant d'obtenir les élements un par un. **Rappel :** Cette interface est composée de trois méthodes :
   1. `hasNext()` qui renvoie `true` s'il reste encore des éléments à parcourir ;
   2. `next()` qui renvoie l'élément parcouru et avance sur l'élément suivant. S'il n'y a plus d'élément à parcourir, la classe devra lever l'exception `NoSuchElementException` ;
   3. `remove()` qui supprime de la table l'élément que `next()` vient de renvoyer. Ici, nous n'implémenterons pas cette méthode, mais lèverons plutôt l'exception `UnsupportedOperationException`.

   La classe implantant l'interface `Iterator` sera déclarée dans un autre fichier ;
3. Transformer la classe implantant l'interface `Iterator` en classe interne de la classe `Table` ;
4. Transformer cette classe en classe interne de la méthode `iterator()` de Table ;
5. Enfin, transformer cette classe en classe anonyme.

Exercice 5 - Réutilisation

Le but de cet exercice est d'écrire une classe permettant de stocker des véhicules (cf. exercice 2) dans une table de véhicules en réutilisant la classe développée dans l'exercice précédent. La classe `TableVehicule` devra posséder les méthodes suivantes :

- `add(Vehicule)` qui permet d'ajouter un véhicule ;
- `iterator(String depCode)` qui itère sur les véhicules du département concerné.

1. Proposer une architecture pour implanter la classe `TableVehicule` en réutilisant la classe `Table` (PS: pas de copier/coller ni de modification de la classe `Table`) ;
2. Implanter l'architecture proposée.