DIC2
ESP
TRAVAUX DIRIGES/ PRATIQUES JAVA 4

NB : créer un projet du nom de **PROGRAMMESJAVA** *pour y mettre les différents codes sources en utilisant comme package* ***td4****.*

Exercice 1. Gestion Etudiant

Réaliser une classe **Etudiant** disposant des fonctionnalités suivantes : On suppose qu'il y a 2 sortes d'étudiants, les étudiants nationaux et les étudiants étrangers. Un étudiant du premier type est identifié par son **nom**, son **prénom** et son **âge**, son **niveau** et sa **filière**. Pour un étudiant du second type, on rajoute le **pays d'origine**.

On considère qu'on a une classe **Etudiant** et une autre classe **EtudiantEtranger** qui dérive de la première classe.

On prévoira donc :

- un constructeur qui permet d'initialiser correctement un étudiant
- un constructeur qui permet d'initialiser correctement un étudiant étranger (ce constructeur appellera le premier.
- une méthode **affiche** (pour chacune des 2 classes) qui imprime sur la console l'identité d'un étudiant.

On prévoit dans la classe **Etudiant** les méthodes **getNom** ( ), **getPrenom** ( ), **getAge** ( ), **getNiveau** ( ), **getFiliere** ( ) qui renvoient le nom, le prénom, l'âge, le niveau et la filière d'un étudiant. On redéfinira convenablement la méthode affiche (en faisant appel à **affiche** de **Etudiant**) dans **EtudiantEtranger.**

Maintenant, créer une classe **Classe** qui permet de lier un étudiant quelconque à une classe.

Une **Classe** est caractérisée par une liste d'étudiants (un tableau d'étudiants de taille maximum **maxEt)** et le nombre d'étudiants (variable statique ? : à déterminer) effectivement contenus dans cette liste d'étudiants.

On prévoit un constructeur qui prend en argument le nombre maximum d'étudiants et permettant ainsi de créer une liste initialement vide d'étudiants.

On prévoira une méthode **public void ajouterEtudiant (Etudiant e)** qui permet d'ajouter un étudiant à une **Classe** en fonction de son niveau et sa filière. Pour ajouter un étudiant, il faut s'assurer que vous n'avez pas dépassé la taille du tableau.

Créer une méthode **afficherListe** qui permet d'afficher tous les étudiants d'une Classe. Dans cette méthode, on fera appel à la méthode **affiche** (mise en œuvre du polymorphisme)**.**

On donne un programme de test :

```
public static void main (String args [ ])
{Classe L1maths = new Classe(4);
L1maths.ajouter (new Etudiant (" Sene " , " Pierre " ,12, " L1", " Maths" ));
L1maths.ajouter (new EtudiantEtranger (" Tall " , " Moussa " , 20 , " L1", " Maths"," Mauritanie "));
……
……
L1maths.afficherListe ( );} }
```

Exercice 2. Calcul des impôts locaux

Dans le cadre de l'informatisation d'une mairie, on veut automatiser le calcul des impôts locaux. On distingue deux catégories d'habitation : les habitations à usage professionnel et les maisons individuelles, l'impôt se calculant différemment selon le type d'habitation. Pour cela, on définit les classes HabitationProfessionnelle et HabitationIndividuelle et les caractéristiques communes à ces deux classes sont regroupées dans la classe Habitation. On a donc un schéma de classes où les classes HabitationProfessionnelle et HabitationIndividuelle héritent de la classe Habitation.

L'objet de cet exercice est d'implémenter ce schéma d'héritage et de mettre en œuvre le mécanisme de liaison dynamique.

Définition de la classe Habitation

Objectif : Définir une classe avec un constructeur et créer une instance de cette classe.

La classe Habitation comprend les attributs :

- propriétaire de type chaîne de caractères et qui correspond au nom du propriétaire,
- adresse de type chaîne de caractères et qui correspond à l'adresse de l'habitation,
- surface de type double et qui correspond à la surface de l'habitation et qui permet de calculer le montant de l'impôt.

Elle dispose aussi des méthodes suivantes :

- double impot() qui permet de calculer le montant de l'impôt que doit payer le propriétaire de l'habitation à raison de 2F par m^2.

- void affiche() qui permet d'afficher les trois attributs de la classe Habitation.
- et un constructeur à trois paramètres permettant d'initialiser une instance de la classe Habitation : Habitation (String proprietaire, String adresse, double surface);

Question 1 : Définissez la classe Habitation

Question 2 : Ecrire une classe TestHabitation pour tester le bon fonctionnement de la classe Habitation.

Définition des classes HabitationIndividuelle et HabitationProfessionnelle

Objectif : Utiliser l'héritage pour définir de nouvelles classes, redéfinir des méthodes dans les classes héritières.

Le calcul de l'impôt d'une maison individuelle est différent de celui d'une habitation, il se calcule en fonction de la surface habitable, du nombre de pièces et de la présence ou non d'une piscine. On compte 100F/pièce et 500F supplémentaire en cas de présence d'une piscine.

Question 3 : Définir la classe HabitationIndividuelle qui hérite de la classe Habitation. Elle dispose des attributs *nbPieces* de type entier et *piscine* de type booléen. Redéfinir les méthodes *impot* et *affiche*. La méthode *affiche* doit afficher les attributs *proprietaire*, *adresse* et *surface* de la classe Habitation, et les attributs *nbPieces* et *piscine* propres à la classe HabitationIndividuelle.

Question 4 : Ecrire une classe TestHabitationIndividuelle pour tester le bon fonctionnement de la classe HabitationIndividuelle.

Le calcul de l'impôt d'une habitation à usage professionnel est également différent de celui d'une habitation. Il se calcule en fonction de la surface occupée par le bâtiment et du nombre d'employés travaillant dans l'entreprise. On compte 1000F supplémentaire par tranche de 10 employés.

Question 5 : Définir la classe HabitationProfessionnelle qui hérite de la classe Habitation et qui dispose de l'attribut *nbEmployes* de type entier. Redéfinir les méthodes *impot* et *affiche*. La méthode *affiche* doit afficher, en plus des attributs *proprietaire*, *adresse* et *surface*, l'attribut *nbEmployes*.

Question 6 : Ecrire une classe TestHabitationProfessionnelle pour tester le bon fonctionnement de la classe HabitationProfessionnelle.

Gestion des habitations d'une commune

Objectif : Mettre en œuvre le mécanisme de liaison dynamique.

On désire à présent calculer l'impôt local des habitations (individuelles ou professionnelles) d'une commune. Pour cela, on utilise une collection d'objets représentée par un tableau *d'habitation* où chaque élément désigne une habitation individuelle ou professionnelle.

Question 7 : Ecrire une classe ImpotHabitation pour tester la mise en œuvre du mécanisme de liaison dynamique en affichant les informations de chaque habitation du tableau et le montant de l'impôt qui lui est associé.

Exercice 3. Interface

On donne l'interface ci-dessous qui permet de manipuler des valeurs de types primitifs :

```
interface Affichable
{
        void affiche ( );
}
```

A partir de cette interface, définissez trois classes **Entier, Flottant** et **Char** qui disposent toutes trois d'un constructeur pour créer un entier, respectivement un flottant et un char (chacune de ces classes aura donc un champ qu'on peut nommer **valeur**) et d'une méthode **affiche** pour afficher l'entier respectivement le flottant et le char.

Créer la classe **HertPoly** disposant des fonctionnalités suivantes : on dispose d'un champ de type tableau d'**Affichable** destiné à contenir une liste d'entiers et/ou de flottant et/ou de char, d'un champ static qui est le nombre actuel d'éléments dans cette liste, d'un constructeur qui permet d'initialiser le tableau

On dispose aussi d'une méthode **public void ajoutValeur (Affichable a)** qui permet d'ajouter un entier ou un flottant ou un char dans le tableau.

On dispose d'une méthode **public void affiche ( )** pour imprimer tous les éléments de la liste (les entiers et/ou les flottants et/ou char)

Voici une classe de test :

```
public class TestHer
{ public static void main (String [] args) {
HerPoly h = new HerPoly (3);
h.ajoutValeur (new Entier (25));
h.ajoutValeur (new Flottant (1.25f)) ;
h.ajoutValeur (new Entier (42)) ;
h.ajoutValeur (new Char ('A')) ;
h.affiche ( );}}
```

Exercice 4. Redéfinition/surdéfinition et polymorphisme

Soit une classe Point avec les membres suivants:

- 2 champs x et y de type int (coordonnées d'un point)
- 1 méthode identique() permettant de tester si 2 points sont de même coordonnées.

Soit la classe PointColore dérivée de Point

- Rajoute un champ c de type byte (la couleurd'un point)
- Définit une autre méthode identique() prenant en compte les coordonnées et la couleur

Point p1,p2; PointColore pc1,pc2;…

p1.identique(p2) renvoie true si p1 et p2 ont mêmes coordonnées, de même

pc1.identique(pc2) renvoie true si pc1 et pc2 ont mêmes coordonnées et couleur

Considérons maintenant

Point p1=new Pointcol (3,4, (byte) 5);

Point p2=new Pointcol (3,4, (byte) 6);

Que vaut p1.identique(p2)? Expliquer.

Peut-on faire en sorte d'avoir le résultat escompté ?