TD n°1

Premières classes

Exercice 1 Qu'affiche le programme suivant ?

```
public class Ex1 {
    int a = 0 ;
    public void f() {
        a++ ;
    }
    public void affiche() {
        System.out.println(a);
    }
    public static void main(String [] args) {
        Ex1 p = new Ex1();
        Ex1 q = new Ex1();
        p.affiche();
        q.affiche();
        p.f();
        p.affiche();
        q.affiche();
        p = q;
        p.f();
        p.affiche();
        q.affiche();
    }
}
```

Exercice 2 Qu'affiche le programme suivant ?

```
public class Ex2 {
    int a = 0 ;
    public static void f(Ex2 o) {
        o.a++ ;
        o.affiche();
    }
    public void affiche() {
        System.out.println(a);
    }
    public static void main(String[] args) {
        Ex2 obj = new Ex2();
        obj.affiche();
        f(obj) ;
        obj.affiche();
    }
}
```

Exercice 3 Qu'affiche le programme suivant ?

```
public class Ex3 {
    int a = 0 ;
    public void f(int a) {
        System.out.println(a) ;
    }
    public static void main(String[] args) {
        Ex3 p = new Ex3();
        p.f(12);
    }
}
```

Dans la fonction `f`, comment faire si on veut également afficher la valeur de l'attribut `a` ?

Exercice 4 Soit le programme suivant :

```
// Fichier Point.java
class Point {
    // deux attributs de type int
    int abscisse;
    int ordonnee;

    // constructeur
    Point(){
        abscisse = 0 ;
        ordonnee = 0 ;
    }

    // méthode permettant de changer les coordonnées d'un point
    void set( int u , int v ){
        abscisse = u ;
        ordonnee = v ;
    }

    // méthode permettant de translater un point
    void translate( int u , int v ){
        abscisse = abscisse + u ;
        ordonnee = ordonnee + v ;
    }
}
```

1. Ajouter à la classe `Point` la méthode `affiche`, de type de retour `void`, de sorte que `p.affiche()` affiche à l'ecran l'abscisse et l'ordonnée de `p`.
2. Ajouter à la classe `Point` la méthode `origine`, de type de retour `boolean` qui teste si les coordonnées du point sont nulles. Ajouter également une méthode `egale` telle que `p.egale(q)` renvoie `true` si et seulement si les abscisses et ordonnées des points `p` et `q` sont égaux.
3. Écrire un deuxième constructeur de la classe `Point`, dont le prototype est `Point( int u , int v )` qui permet d'initialiser l'abscisse et l'ordonnée avec `u` et `v`. Écrire une seconde méthode `set`, prenant en argument un objet de la classe `Point`, et qui recopie les attributs de ces arguments.
4. Ajouter à la classe `Point` une méthode `symetrie` telle que `p.symetrie()` renvoie un nouvel objet `Point` qui représente le symétrique du point `p`, dans une symétrie centrale par rapport à l'origine du repère.

5. On veut numéroter les points au fur et à mesure de leur création. On ajoute donc les variables suivantes :
   `static int nombre ;`
   `int numero ;`
   où l'attribut `numero` indique le numero du point et où la variable de classe `nombre` indique combien d'objets ont été créés.
   Réécrire les constructeurs `Point` en conséquence. Réécrire également la méthode `affiche` pour observer la valeur de ces nouveaux attributs.

Exercice 5 Ecrire une classe implémentant une *paire* d'entier :

1. Définir une classe `Paire` dont le constructeur initialise les attributs privés de la paire. Définir une méthode `affiche` et une fonction `main` pour tester cette classe.
2. Définir un deuxième constructeur, qui initialisera à 0 les composants de la paire.
3. Définir un troisième constructeur, qui initialisera une paire à l'aide d'une autre paire.
4. Définir des fonctions permettant d'accéder et de modifier chaque élément de la paire.

Exercice 6 Algorithmes de tri naïfs, sur des Paires d'entiers :

1. Enrichir la classe `Paire` d'une méthode définissant quand une paire est inférieure à une autre selon la règle lexicographique suivante :
   $(x1, y1) < (x2, y2)$ *ssi* $(x1 < x2)$ *ou* $(x1 = x2$ *et* $y1 < y2)$.
2. Le principe du tri bulle est de parcourir un tableau jusqu'à ce qu'il soit trié. A chaque parcours, on compare deux à deux les éléments consécutifs et on les permute s'ils ne sont pas dans l'ordre. Si lors d'un parcours on n'a permuté aucun élément, le tableau est trié. Créer une classe `Tri`, et une methode statique `Tribulle` qui trie un tableau de paires.
3. Le principe du tri par insertion consiste à considérer une à une les valeurs du tableau, et à les insérer au bon endroit dans le tableau constitué des valeurs précédemment considérées et triées. Le tableau est donc parcouru de droite à gauche, dans l'ordre décroissant. Les éléments sont donc décalés vers la droite tant que l'élément à insérer est plus petit qu'eux. Dès que l'élément à insérer est plus grand qu'un des éléments du tableau trié, il n'y a plus de décalage et on insère l'élément dans la case laissée vacante. Créer une méthode statique `TriInsertion` qui trie un tableau de paires.
4. Le principe du tri fusion suit la stratégie *diviser pour régner* : on divise le tableau à trier en deux parties, on trie (récursivement) chacune d'entre elle, et on fusionne le résultat. Créer une méthode statique `TriFusion` qui trie un tableau de paires.
5. Pour les trois algorithmes de tri que vous avez implémenté, et pour les deux tableaux suivants :
   – $\{(3,3),(2,2),(1,1),(0,0)\}$
   – $\{(0,0),(1,1),(2,2),(3,3)\}$
   Combien de comparaisons de paire ont été effectuées ? Combien de références vers des objets de la classe `Paire` ont été déclarées lors du tri ?