

TD/TP 2

L'**objectif** de ce TD est de comprendre et appliquer les notions vues au cours ; à savoir :

- La classe et ses caractéristiques (attributs/méthodes, encapsulation « private/public », static, constructeurs).
- Les caractéristiques héritées de la classe **Object** (méthodes **equals()** et **toString()**).
- Codage des relations qui existent entre classes (association simple, agrégation et composition).

I. Notion de classe

1.1 Classe, attributs, méthodes, instantiation

Créer une classe **Point** (coordonnées x,y) avec des méthodes **setX()**, **setY()**, **getX()**, **getY()** pour respectivement affecter une coordonnée (x ou y) et accéder à sa valeur (x ou y). Mettre cette classe dans un fichier **Point.java** et le compiler.

- 1) Faire un programme test qui crée un point p, lui affecte des coordonnées et imprime ensuite ses coordonnées. Mettre ce programme dans un fichier **TestPoint.java** dans le même répertoire.

```
1  public class TestPoint{  
    Run | Debug  
2      static public void main(String args[]){  
3          Point p = new Point();  
4          p.setX (p: 3);  
5          p.setY (p: 4);  
6          System.out.println( p.getX() );  
7          System.out.println( p.getY() );  
8      }  
9  };
```

- 2) Constaté qu'on ne peut appeler aucune méthode sur un objet **Point** sans avoir l'initialisé par **new Point()**; (constructeur déjà défini en Java).
- 3) Remarquer que dans les méthodes on peut aussi écrire **this.x** ou **this.y** au lieu de **x** ou **y** tout court.
- 4) Dans votre programme de test main, essayer d'accéder directement à x, y par la notation **p.x** et **p.y** où p est une instance de la classe **Point**. Conclusion.
- 5) Dans la classe, mettre x, y **public** au lieu de **private** cette fois-ci. Conclusion. (Remettre ensuite x, y privés.)

1.2 Autres méthodes

- 1) Rajouter à la classe **Point** d'autres méthodes de votre choix, par exemple :
 - a. Déplacer un point d'une longueur sur l'axe des x et des y,
 - b. Ramener un point à l'origine par une méthode **reset()**, etc.

- 2) Définir aussi une méthode **public double distance (Point b)** qui calcule la distance entre le point **this** et le point **b** en paramètre. Tester en calculant la distance entre les points (1,2) et (2,3).
- 3) Définir aussi une méthode (version-2) **public static double distance (Point a, Point b)** qui calcule la distance entre les points **a** et **b** en paramètres. Comment utiliser cette méthode ? Quelle est la différence avec le cas 2 précédent ? Quelle serait votre choix de conception d'une méthode distance, le cas 3. ou 2. ?

1.3 Egalité ou pas entre deux objets ?

- 1) La méthode **public boolean equals(Object o)**, héritée de la classe **Object**, permet de tester «l'égalité» entre deux objets. Usage : **p** et **q** étant deux instances de **Point**.

p.equals(q);

Permet de tester si les deux points **p** et **q** sont égaux ou pas.

- 2) Créer, toujours dans votre programme de test, deux points **p** et **q** et leur affecter les mêmes coordonnées. Vérifier le résultat de la méthode **equals** sur ces points. Conclusion ? (La réponse est false, voir ci-après)
- 3) Même question, mais cette fois-ci, le point **p** est initialisé normalement par **new** et **q** est initialisé par l'affectation

q = p;

Conclusion ?

Réponse : en 2. on compare en fait les références de deux points différents (même si ayant même valeurs).

En 3. on compare deux références égales, car référence à un même objet.

- 4) Dans la classe **Point**, définir la méthode **equals** comme suit :

```
public boolean equals(Point a){
    return (this.x==a.x && this.y==a.y);
}
```

où on compare deux points par leurs coordonnées. Refaire le test précédent. Conclusion ?

- 5) Maintenant la classe **Point** possède deux méthodes **equals** : une définie à 4) et une deuxième héritée de la classe **Object** (**public boolean equals(Object)**).

Dans la classe de test, déclarer les deux objets **p** et **q** comme suit :

Object p = new Point (2,5);

Object q = new Point (2,5);

Comparer à nouveau. Conclusion ?

Redéfinir la méthode **equals** héritée de la classe **Object** :

```
boolean equals(Object a){
    return (this.x == ((Point)a).x && this.y == ((Point)a).y );
}
```

Refaire le test précédent. Conclusion ?

II. Notion de constructeur

Reprendre la classe `Point` avec les deux nouvelles méthodes suivantes qui sont des constructeurs.

```
// Initialise un point à l'origine
public Point(){x = 0; y = 0;}

// Initialise un point à (a et b)
public Point(int a, int b){x = a; y = b;}
```

1) Dans les programmes, `main()`, utiliser

`Point p = new Point();` pour déclarer et initialiser un objet `Point` à (0,0) par défaut.

`Point q = new Point(5,2);` pour déclarer et initialiser un objet `Point` à (5,2).

A noter :

- Avec `new Point()` il sera fait appel au constructeur sans paramètre défini juste ci-dessus, au lieu de celui hérité comme dans 1.1.
 - **Important :** Initialiser un point par constructeur, e.g. `Point p = new Point (2,5)`, n'est pas la même chose que lui affecter des valeurs par les méthodes `p.setX(2)` et `p.setY(5)`, même si dans les deux cas l'objet a la même valeur. En effet, dans le premier cas, initialisation, l'objet `p` n'existe pas avant son initialisation, alors que dans le deuxième cas, affectation, `p` est déjà créé mais ne fait que changer de valeurs.
- 2) Vérifier que si on omet le constructeur `Point()` sans paramètre, c'est une erreur de compilation. En effet, à partir du moment qu'un constructeur est déclaré, l'appel `new Point()`, ne cherche pas le constructeur par défaut hérité précédemment, mais celui que l'utilisateur doit définir aussi.
- 3) Remplacer maintenant le code `{x = 0; y = 0;}` du constructeur par défaut, par `{this(0,0);}`. Vérifier le résultat. L'instruction `this(0,0)` est un appel de l'instance courante au constructeur `Point(int a, int b)` avec ici 0 et 0 comme paramètres.

C'est d'ailleurs la seule fois qu'un constructeur peut être appelé explicitement.

Exercice :

- Rajouter un constructeur avec un seul paramètre (initialisation de l'abscisse) qui affecte ce paramètre à `x`, et 0 à `y`.
- Ecrire le code de ce constructeur de deux façons différentes.
- Vérifier qu'on peut réécrire le constructeur (par défaut) par `{this (0);}` qui fait appel au constructeur `public Point(int a)` nouvellement ajouté.

III. Conversion vers texte `toString()`

Comme pour la méthode `equals(Point)` (§ 1.3), on peut redéfinir la méthode `toString()` héritée aussi de `Object`, pour convertir un objet `Point` vers une chaîne de caractères imprimable. Exemple :

```
public String toString(){
    return "(" + x + "," + y + ")";
}
```

Exemple d'usage :

```
Point pt = new Point (a: 2,b: 5);
System.out.println(pt.toString()); // imprime (2,5)
```

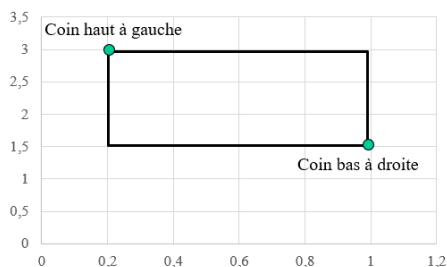
Vérifier en changeant le message d'affichage avec un autre texte dans la méthode `toString()`.

IV. Codage des associations

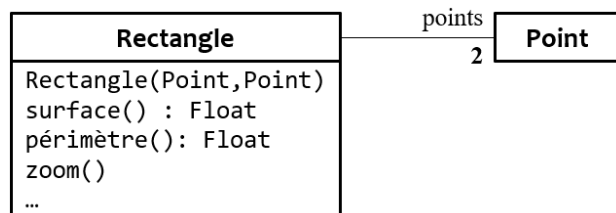
4.1 Codage d'une association simple

Un rectangle a quatre sommets qui sont des points. On peut construire un rectangle à partir des coordonnées de deux points (le point haut à gauche et le point bas à droite). Il est possible de calculer sa surface et son périmètre, ou encore le zoomer. Le diagramme de classes suivant illustre cette situation :

Représentation dans le plan cartésien



Représentation UML



Créer une classe `Rectangle` (un nouveau fichier, `Rectangle.java`), qui utilise la classe `Point` et implémente les méthodes décrites dans le diagramme de classes ci-dessus.

```
1  class Rectangle {           // Rectangle droit
2      private Point hg ;      // Le coin haut à gauche
3      private Point bd ;      // Le coin bas à droite
4
5      public Rectangle() {
6          // rectangle par défaut. Choisir son initialisation
7      }
8      public Rectangle(Point h, Point b) {
9          // initialisation des coins à partir des paramètres donnés
10     }
11     public void afficher() {
12         // Affiche les coordonnées des coins
13     }
14     public int surface() {
15         // calcul de la surface
16     }
17     public void zoom(int deltax, int deltay) {
18         // Dilatation des coordonnées. Delta donné.
19     }
20     // autres méthodes...
21 }
```

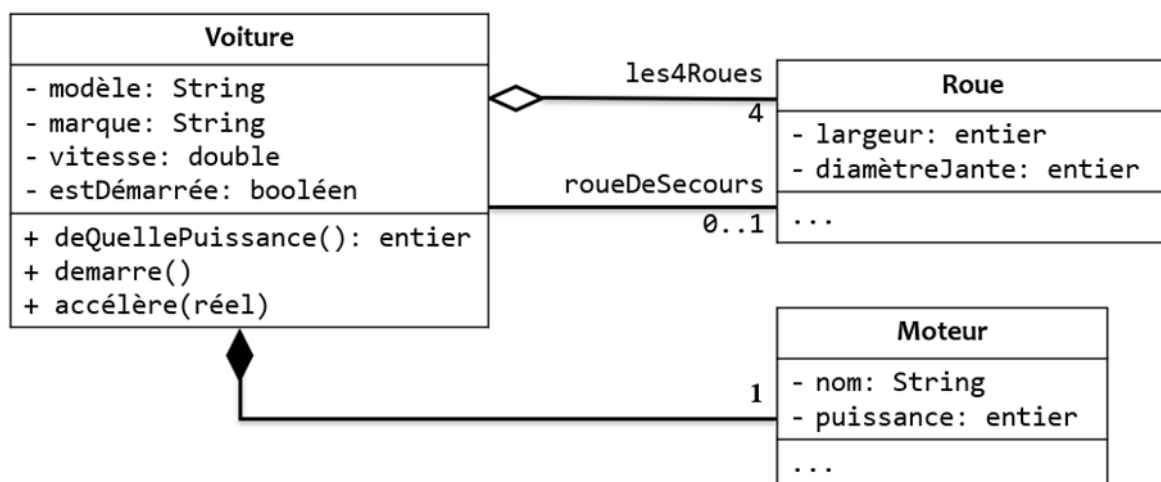
- 1) Programmer les méthodes et les tester. (Nouvelle classe `TestRectangle.java` avec la méthode `main()`).
- 2) Rajouter à la classe `Point` un constructeur copie
Point (Point p) {...}
 et l'utiliser dans le constructeur **Rectangle(Point h, Point b)** . Quel est l'intérêt par rapport à avant ?
- 3) Rajouter les méthodes appropriées **set** et **get** qui modifient et lisent les attributs d'un rectangle.

4.2 Codage d'une composition/agrégation

Reprendre l'exemple du cours qui modélise les voitures ; et prenant en considération les mises à jour suivantes :

- Une voiture a une marque, un modèle, une vitesse et un moteur.
- Le moteur a un nom et une puissance.
- La voiture a quatre roues et une roue de secours optionnelle.
- Le constructeur de la classe `Voiture` se chargera d'associer les 4/5 roues et le moteur à la voiture.

Considérant le diagramme de classes suivant :



- 1) Implémenter les différentes classes et ajouter les méthodes qui manquent. Créer une nouvelle classe `TestVoiture.java` et tester ces méthodes.
- 2) Rajouter les méthodes appropriées **set** et **get** qui modifient et lisent les attributs des différentes classes.
- 3) La méthode `deQuellePuissance()` doit retourner la puissance du moteur associé à la voiture en question.
- 4) On suppose que la durée de vie d'un Moteur est de 300 000km. Définir une méthode `void changerLeMoteur(Moteur)` permettant de remplacer un moteur endommagé par un autre nouvellement créé.
- 5) On veut permettre un échange de messages entre les classes `Voiture` et `Moteur` (navigation bidirectionnelle). Faire les modifications nécessaires (constructeurs des deux classes). Tester avec des exemples.