IN201 : 1 – Classes et objets

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :

Solution

Résumé

Le but de ce TP est de manipuler les notions de classe et d'objet au travers de la classe `Equation` vue en cours et d'un exemple simple, la classe `Point`.

1 Contenu

Ce corrigé succinct contient les réponses au TP numéro 1. Vous y trouverez en particulier les diagrammes UML de la partie conception. Le corrigé de la partie implantation en Java est disponible sur http://www.tofgarion.net/lectures/IN201.

2 Manipulation de la classe `Equation`

Le site http://www.tofgarion.net/lectures/IN201 servira tout au long du cours. Sur ce site vous trouverez les sujets de chaque séance et les corrigés des TPs.
Récupérer sur le site l'archive `.jar` contenant les fichiers source `Equation.java` et `ResolutionEquation.java`. Décompresser l'archive via la commande :

```
jar xvf TP1.jar
```

et placer les sources dans le répertoire `src` précédemment créé.
– compiler les deux classes en utilisant le compilateur `javac` ;
– générer la documentation HTML en utilisant `javadoc` (utiliser l'option –**private**) ;
– « exécuter » la classe `ResolutionEquation` en utilisant la machine virtuelle `java`.

Solution :
A priori, pas de problème particulier. Le but était bien sûr de manipuler au moins une fois les outils du JDK à travers la console. Vous trouverez une copie de la trace d'exécution de `ResolutionEquation` sur le site.

3 Conception et implantation d'une classe `Point`

On cherche à concevoir une classe `Point` qui représente un point dans le plan (au sens géométrique). On souhaite pouvoir réaliser le « programme » suivant :
– création d'un point p_1 de coordonnées cartésiennes $(1, 0)$;
– affichage des coordonnées du point p_1 ;
– translation du point p_1 en utilisant le vecteur $(5, 0)$. On pourra également translater le point en utilisant d'autres vecteurs ;
– affichage des coordonnées du point p_1 ;
– création d'un point p_2 de coordonnées cartésiennes $(-1, -5.5)$;
– calcul et affichage de la distance entre p_1 et p_2.

1. comment représenter l'état d'un point ? En déduire les attributs de la classe `Point` ;

   Solution :
   On cherche ici à concevoir une classe `Point` représentant un point au sens géométrique. Le « programme » de test nous permettait de trouver les attributs et les méthodes de la classe.
   Les attributs de la classe seront l'abscisse et l'ordonnée du point qui sont des réels. On aurait pu choisir d'utiliser des coordonnées polaires, mais le programme nous indiquait que l'on construisait des points avec des coordonnées cartésiennes. Nous noterons ces attributs `x` et `y`.

2. de quoi a-t-on besoin pour construire un point ? Quelle est la signature du ou des constructeurs de la classe ?

   Solution :
   Pour construire un point, nous avons besoin de ses deux cordonnées cartésiennes. La signature du constructeur de la classe `Point` va donc être la suivante : `Point(x_: double, y_: double)`.

3. les opérations (ou les méthodes) de la classe représentent les services qui peuvent être rendus par un objet de type `Point`. Utiliser le « programme » fourni pour déterminer les opérations de la classe `Point`. On se placera dans le cadre d'une conception d'une classe `Point` générique (pensez au méthodes : faut-il des paramètres, par exemple pour translater ?) ;

   Solution :
   Les méthodes nous sont données par les services demandés à un point :
   – une méthode `translater` qui ne renvoie rien et prend deux réels en paramètres (qui correspondent aux coordonnées du vecteur de translation) ;
   – une méthode `afficher` qui ne renvoie rien et ne prend pas de paramètres ;
   – une méthode `distance` qui renvoie un double et prend une référence vers un objet de type `Point` en paramètre ;
   – des accesseurs et des modifieurs pour les attributs privés de la classe. Ces accesseurs et modifieurs porteront les noms définis en cours [1].

4. quelle est la visibilité des attributs et des méthodes ?

   Solution :
   Nous allons appliquer le *principe d'encapsulation* : les attributs de la classe seront donc *privés*. Le constructeur de `Point` sera bien évidemment publique, pour pouvoir instancier la classe. Les méthodes que nous avons choisies seront également publiques.

5. écrire le diagramme UML de la classe `Point` ;

   Solution :
   Avant de donner la représentation UML de la classe `Point`, nous allons définir les méthodes facultatives qui sont demandées plus loin :
   – `lt` prend un point en paramètre et renvoie un booléen qui est **`true`** ssi le point en paramètre est strictement plus petit que le point sur lequel on appelle la méthode ;
   – `ge` prend un point en paramètre et renvoie un booléen qui est **`true`** ssi le point en paramètre est plus grand que le point sur lequel on appelle la méthode ;
   – `gt` prend un point en paramètre et renvoie un booléen qui est **`true`** ssi le point en paramètre est strictement plus grand que le point sur lequel on appelle la méthode ;
   – `le` prend un point en paramètre et renvoie un booléen qui est **`true`** ssi le point en paramètre est plus petit que le point sur lequel on appelle la méthode.
   L'ordre est défini comme un ordre lexicographique : on compare d'abord les abscisses puis les ordonnées.
   La représentation UML de la classe est donnée sur la figure 1.

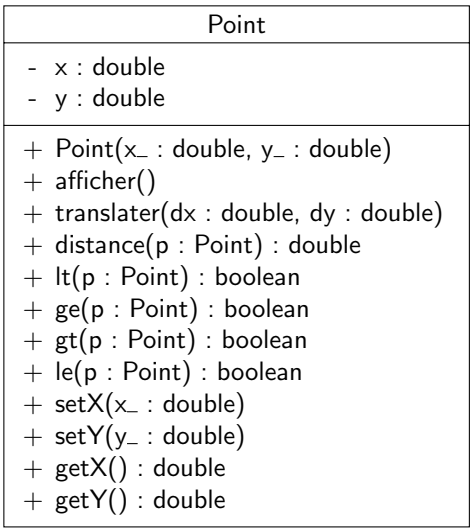

FIGURE 1 – Représentation UML de la classe `Point`

6. implanter la classe `Point` en Java. On « traduira » dans un premier temps le diagramme UML réalisé, puis on documentera la classe, ses attributs et ses méthodes et enfin, on écrira le code des méthodes. On testera les méthodes **au fur et à mesure de leur écriture**.

   Comme vous aurez à utiliser des méthodes mathématiques, comme par exemple le calcul d'une racine carrée ou l'élévation d'un nombre à une puissance, **vous consulterez la documentation javadoc de la classe `Math`** (cf. adresse de la documentation javadoc de l'API Java sur le site). Attention, les méthodes de la classe sont *statiques*, on les appelle donc en utilisant le nom de la classe `Math` et pas un objet instance de `Math`.

   Solution :

   L'implantation de la classe `Point` et de la classe `TestPoint` sont disponibles sur le site. Je voudrais attirer votre attention sur la méthode `distance`. Dans la présentation de l'application qui doit utiliser les points, on demande à pouvoir calculer la distance entre deux points et l'afficher. On pourrait alors être tenté d'inclure l'affichage de la distance à l'intérieur de la méthode `afficher`. Cependant, je ne vous le conseille pas :
   - l'utilisateur de votre classe `Point` peut très bien afficher la distance lui-même ;
   - si vous passez votre code en environnement de production (i.e. qu'il est livré à votre client), celui-ci va se retrouver avec des affichages sur la console alors qu'il ne le veut peut-être pas. Il faut savoir que les affichages console peuvent être pénalisants en termes de performance. Donc il ne vaut mieux pas inclure l'affichage dans la méthode `distance`. Cette remarque vaut également pour les affichages de « débogage » que vous pouvez inclure dans votre code : lorsque vous livrez votre logiciel, il faut les enlever, ce qui est fastidieux. Nous verrons en séance 2 que nous pouvons utiliser un *framework* de test simple, JUnit, qui permet de simplifier l'écriture de tests. On peut également utiliser des solutions avancées dites de *logging*, comme log4j [1].

   Enfin, vous remarquerez que j'ai développé une méthode particulière, `toString`, qui *renvoie*[2] une représentation du point sous forme de chaîne de caractères. `toString` est une méthode particulière : lorsqu'une classe possède une telle méthode (attention à sa signature), le compilateur l'utilise automatiquement lorsqu'il a besoin d'avoir une représentation d'un objet de la classe sous forme de chaîne de caractères. Par exemple, considérons la méthode `afficher` de `Point` :

   ```
   public void afficher() {
       System.out.println(this);
   }
   ```

   Dans `afficher`, `System.out.println` attend un objet de type `String` en paramètre. Or, je lui passe ici **`this`**, qui est une instance de `Point`. Le compilateur transforme en fait cet appel en `System.out.println(this.toString())` *automatiquement*.

   Notez que les classes qui ne définissent pas `toString` possèdent quand même une méthode `toString` « par défaut ». Nous reparlerons de tout cela lors du cours sur l'héritage.

7. écrire une classe `TestPoint` qui implante le programme décrit de façon informelle plus haut ;

   Solution :

   Rien de bien particulier, il faut implanter un *programme*, donc créer une classe `TestPoint` possédant une méthode `main`. Le code source est disponible sur le site.

8. (facultatif) : ajouter les opérations de comparaisons

   L'objectif de cette question est de définir des méthodes de comparaison sur les points.

   1. écrire une méthode `lt` qui indique si un point est strictement inférieur à un autre. Pour comparer deux points, on commencera par comparer les abscisses des points puis leurs ordonnées.
   2. en utilisant la méthode `lt` coder les méthodes `ge` (supérieur ou égal), `gt` (supérieur strict), `le` (inférieur ou égal).

   Solution :

   cf. questions précédentes.

Références

[1] *Apache log4j*. URL : http://logging.apache.org/log4j/.