Points et segments
Corrigé

Objectifs

— Comprendre les notions de classe, objet et poignée ;
— Comprendre et compléter des classes Java ;
— Utiliser le JDK d'Oracle pour compiler, exécuter et documenter.

L'objectif de ces exercices est de comprendre les concepts objets en exécutant en salle machine un exemple simple et en le complétant.

L'exemple que nous prenons consiste à implanter une version simplifiée d'un outil de dessin de schémas mathématiques. Un schéma est composé de points et segments. La figure 1 montre un exemple de schéma : un triangle (défini par trois segments) et son centre de gravité.

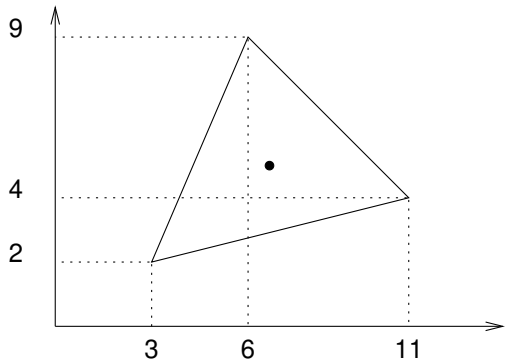

FIGURE 1 – Schéma mathématique composé de trois segments et de leur barycentre

Avertissement : Contrairement à ce que laisse supposer la figure 1, l'objectif n'est pas de faire du graphisme mais d'illustrer les concepts objets. Nous nous limiterons ainsi à une visualisation en mode texte.

Exercice 1 : Comprendre la classe Point

Le texte source de la classe Point correspond au diagramme d'analyse donné à la figure 2.

1.1. Expliquer comment on obtient le squelette d'une classe Java à partir de son diagramme d'analyse UML.

Solution : Le diagramme d'analyse UML donne :
— le nom de la classe qui devient le nom de la classe Java,
— les requêtes et les commandes, qui deviennent des méthodes (pour l'instant on n'a que leur signature et pas encore leur code),

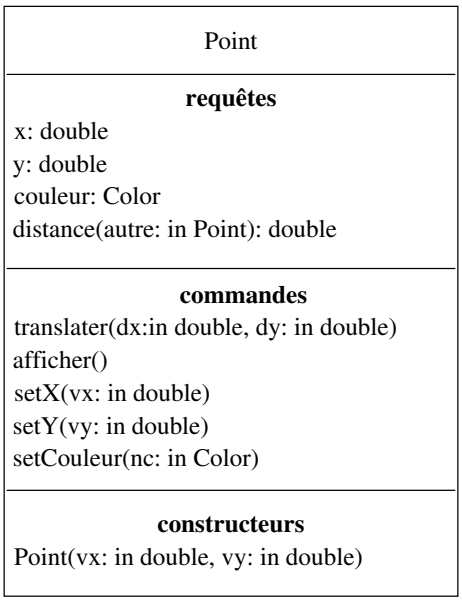

FIGURE 2 – Diagramme d'analyse de la classe Point

— les constructeurs qui deviennent des... constructeurs (dans que l'on ne connaissent encore leur code).

Ensuite, on pourra choisir les attributs (ici, l'abscisse, l'ordonnée et la couleur). On pourra alors ensuite écrire le code des méthodes et constructeurs, éventuellement ajouter d'autres éléments privés (méthodes locales par exemple).

1.2. Indiquer, s'il y en a, les entorses par rapport aux conventions de programmation Java.

Solution : Les conventions de programmation Java préconisent de mettre les attributs en début de classe et non à la fin ! L'ordre préconisé est :

1. les attributs
2. les constructeurs
3. les méthodes.

Il y a peut-être d'autres entorses... À vous de me les signaler !

Exercice 2 : Compiler et exécuter

Dans cet exercice, nous allons utiliser les outils du JDK (Java Development Kit) pour compiler et exécuter une application.

2.1. *Lire et comprendre le programme ExempleComprendre.* Lire le programme ExempleComprendre et dessiner l'évolution de l'état de la mémoire au cours de l'exécution du programme. Ceci doit être fait sur le listing, avant toute exécution du programme.

Solution : Il faut bien voir que les variables de type Point sont des poignées, donc des pointeurs sur des objets Point. Plusieurs poignées peuvent désigner le même objet. S'il est modifié au travers d'une de ses poignées, les modifications seront visibles des autres.

L'exécution avec Java Tutor (question 2.5) permettra de bien comprendre ce qui se passe.

2.2. *Compiler ExempleComprendre.* Le compilateur du JDK s'appelle `javac` (Java Compiler). Il transforme chaque classe (ou interface) contenue dans un fichier source Java (extension `.java`) en autant de fichiers contenant du byte code (extension `.class`). Remarquons que lorsqu'une classe est compilée, toutes les classes qu'elle utilise sont également automatiquement compilées. Par exemple, pour compiler la classe ExempleComprendre contenue dans le fichier ExempleComprendre.java, il suffit de taper la commande suivante :

```
javac ExempleComprendre.java
```

Cette commande compile `ExempleComprendre.java` et produit `ExempleComprendre.class` mais aussi `Point.java` pour produire `Point.class` puisque ExempleComprendre utilise la classe Point.

Attention : Pour que ceci fonctionne, il est nécessaire que le fichier porte exactement le même nom (y compris la casse) que l'unique classe publique qu'il contient.

Compiler le programme `ExempleComprendre` fourni.

Remarque : On pourra utiliser l'option `-verbose` du compilateur `javac` pour lui faire afficher les différentes opérations qu'il réalise.

Solution : RAS. Il faut juste faire ce qui est demandé...

2.3. *Exécution de ExempleComprendre.* Le byte code n'est pas directement exécutable par le système mais par la machine virtuelle de Java (Java Virtual Machine). La machine virtuelle fournie avec le JDK s'appelle `java`. Seules les classes qui contiennent la définition de la méthode principale peuvent être exécutées. La méthode principale est :

```
public static void main (String[] args)
```

Ainsi, la classe Point ne peut pas être exécutée et la classe ExempleComprendre peut l'être. Pour exécuter ExempleComprendre, il suffit de taper la commande :

```
java ExempleComprendre
```

Attention : Il ne faut pas mettre d'extension (ni `.class`, ni `.java`), mais seulement le nom de la classe (en respectant la casse !).

Exécuter le programme ExempleComprendre.

Remarque : Tous les appels suivants sont incorrects. Pourquoi ? Vérifier les réponses en exécutant les commandes et en lisant les messages affichés.

```
java ExempleComprendre.java
java exemplecomprendre
java Point
java PasDeClasse
```

Solution : L'appel `java exemplecomprendre` dépend de votre système. Sous Windows (qui ne fait pas différence entre majuscules et minuscules dans les noms de fichier), vous aurez :

```
Erreur : impossible de trouver ou de charger la classe principale exemplecomprendre
Causé par : java.lang.NoClassDefFoundError: ExempleComprendre (wrong name: exemplecomprendr
```

Sous Linux, vous aurez le message suivant :

```
Erreur : impossible de trouver ou de charger la classe principale exemplecomprendre
Causé par : java.lang.ClassNotFoundException: exemplecomprendre
```

La raison est qu'il le fichier exemplecomprendre.class n'a pas été trouvé contrairement à Windows. Pour reproduire le message de windows, il faudrait créer un lien symbolique (ou copier le fichier) :

```
ln -s ExempleComprendre.class exemplecomprendre.class
```

Notez bien la différence entre `ClassNotFoundException` et `NoClassDefFoundError`, en particulier le suffixe `Error` ou `Exception`, qui deviendra plus claire après le cours sur les exceptions.

2.4. *Vérifier les résultats.* Vérifier que l'exécution donne des résultats compatibles avec l'exécution à la main réalisée à la question 2.1.

2.5. *Exécuter avec Java Tutor.* Pour bien comprendre ce programme, on peut l'exécuter avec Java Tutor (`http://www.pythontutor.com/java.html`) qui affiche l'évolution de l'état de la mémoire lors de l'exécution pas à pas du programme. Il suffira de copier le contenu du fichier `ExempleComprendreTutor.java` et le coller dans le cadre destiné à écrire le programme Java. On peut ensuite lancer l'exécution : *Visualize Execution*.

2.6. *Corriger le programme ExempleErreur.* Compiler le programme ExempleErreur. Le compilateur refuse de créer le point à attacher à `p1`. Est-ce justifié ? Expliquer pourquoi. Quel est l'intérêt d'un tel comportement ? Corriger le programme.

Solution : Le compilateur dit qu'il ne trouve pas le constructeur sans paramètre : Point(). C'est effectivement le cas. Point définit un constructeur qui prend deux réels comme paramètres. Il ne possède pas de constructeur sans paramètre. Le compilateur refuse de créer `p1` car il ne sait pas l'initialiser, il ne trouve pas de constructeur correspondant.

L'intérêt est que le compilateur contrôle que tout objet créé est correctement initialisé grâce à l'exécution du constructeur adéquat, choisi en fonction des paramètres effectifs fournis. Tout objet créé sera initialisé.

Pour corriger le programme, il ne faut surtout pas définir le constructeur par défaut dans la classe Point, ni construire un point de coordonnées (0, 0) mais fournir les bons paramètres effectifs (et supprimer les deux lignes qui suivent).

2.7. *Durée de vie des objets.* Supprimer dans le fichier `Point.java` les commentaires devant l'affichage dans le constructeur (devant `System.out.println("CONSTRUCTEUR...");`) et les commentaires à la C (`/* */`) autour de la méthode `finalize` (vers la fin du fichier).

Compiler et exécuter le programme CycleVie. Est-ce que les points sont « détruits » ? Augmenter le nombre d'itérations (5000, 50000, 500000, etc) jusqu'à ce que les messages de « destruction » apparaissent lors de l'exécution du programme.

Solution : Avec 1000 itérations, on ne voit que des constructions et aucune destruction. Le ramasse-miettes ne s'est donc mis en route. Conséquence, il n'y a pas de garantie que le destructeur de tous les objets d'une application soient exécutés.

Quand on passe à 50000 (ou plus) on constate des séries de constructions et des séries de destruction. Les destructions ont lieu quand le ramasse-miette se met en route. Conséquence : un objet non accessible n'est pas immédiatement récupéré par le ramasse-miette.

Le comportement a évolué avec les versions de Java. Avant, il y avait de longue période de création puis de destruction, puis de création... Maintenant, les séries de destruction sont plus courtes et plus fréquentes.

Attention, tous les points ne sont pas nécessairement détruits !
Conclusion : La méthode `finalize` (le destructeur de Java) n'est pas un moyen fiable pour libérer une ressource.

Exercice 3 : Produire la documentation
Le JDK fournit un outil appelé javadoc qui permet d'extraire la documentation directement du texte des classes Java (l'option -d indique le dossier où la documentation sera engendrée).

```
javadoc -d doc *.java
```

Seule est engendrée la documentation des classes contenues dans les fichiers donnés en argument de la commande `javadoc`. Par défaut, seuls les éléments publics sont documentés. On peut utiliser l'option `-private` pour engendrer la documentation complète (mais utile seulement pour le concepteur des classes).

La documentation doit bien entendu être donnée par le programmeur de la classe. Ceci se fait grâce aux commentaires qui commencent par `/**` et se terminent par `*/` et qui sont placés juste devant l'élément décrit. Des balises HTML peuvent être utilisées pour préciser une mise en forme du texte.

Il existe des étiquettes prédéfinies telles que *@author, @version, @see, @since, @deprecated* ou *@param, @return, @exception* pour la définition d'une méthode.

3.1. *Produire la documentation des classes.* Utiliser la commande javadoc pour produire la documentation des classes de notre application. Consulter la documentation ainsi engendrée.

3.2. *Consulter la documentation de la classe Segment.* En consultant la documentation engendrée pour la classe Segment, indiquer le sens des paramètres de la méthode `translater` et le but de la méthode `afficher`.

3.3. *Consulter la documentation en ligne.* La documentation des outils et des API Java est disponible en ligne à partir de l'URL suivante :

```
http://docs.oracle.com/javase/7/docs/api/
```

Exercice 4 : Comprendre et compléter la classe Segment
Une version incomplète d'une classe Segment est fournie ainsi que le programme TestSegment.

4.1. Compléter le diagramme d'analyse UML en faisant apparaître la classe Segment.

Solution : Un segment possède deux requêtes, couleur et longueur, et trois commandes, afficher, translater et changerCouleur. Un segment est construit à partir de ses 2 points extrémités (qui initialiseront les deux attributs privés).

Segment
requêtes
longueur : double color : java.awt.Color
commandes
translater(dx, dy : double) afficher changerCouleur(nouvelle : java.awt.Color)
constructeurs
Segment(extremité1, extrémité2 : Point)

4.2. Compléter la classe Segment. On commencera par donner le code de la méthode `translater`, puis des méthodes `longueur` et `afficher`. Le code actuel n'est en effet pas le bon ! Le fichier `TestSegment` contient un programme de test.

Solution : Cf les classes constituant la solution finale.

On se pose la question « Comment faire pour translater un segment de dx et dy ? ». On devrait répondre : on translate ses points extrémités des mêmes dx et dy. On a alors la bonne réponse : on appliquer la méthode translater de Point sur extrémité1 et extrémité2 avec dx et dy en paramètre.

On peut prendre des chemins plus longs...

Par exemple, on pourrait commencer par faire :

```
this.extremite1.x += dx;
this.extremite1.y += dy;
...
```

Mais on ne peut pas puisque x est privé dans Point. Idem pour y.

On pourrait alors rectifier en faisant :

```
this.extremite1.setX(this.extremite1.getX() + dx);
this.extremite1.setY(this.extremite1.getY() + dy);
...
```

Ceci fonctionne mais à quoi correspondent les deux premières lignes : à translater le point extrémité 1. Il y a justement une méthode translater sur Point !

Normalement, les classes devraient fournir les opérations de haut niveau dont on a besoin. Et c'est le cas ici. Il faut donc s'en servir au lieu d'essayer de les réimplanter.

Exercice 5 : Définir un schéma particulier

Écrire un programme Java qui construit le schéma de la figure 1. Ce schéma est composé de quatre éléments :

— trois segments (s_{12}, s_{23} et s_{31}) construits à partir de trois points (p_1, p_2 et p_3) ;

— et le barycentre de ces trois points.

Solution : Le seul point notable est qu'avant de créer le barycentre, on va calculer ses coordonnées (de manière à le créer avec les bonnes informations). On a deux variables qui seront initialisées respectivement avec la moyenne des abscisses et la moyenne des ordonnées des trois points.