Travaux Dirigés de Java n°2

Cours de Java

—IG 2000 - IR 2—

Octobre 2003

Les classes définition, redéfinition, surcharge et délégation

Dans cette séance, nous allons écrire et utiliser nos propres classes. Les mécanismes de définition, redéfinition, surcharge et délégation vont être abordés. Pour garder de bonnes habitudes, vous écrirez toutes vos classes dans des paquetages dont le nom commence par `fr.umlv.nomdelogin.cequevousvoulez`. Par exemple, vous pouvez mettre les classes que vous écrirez durant ce td dans le paquetage `fr.umlv.nomdelogin.td2`. De plus, on rappelle qu'il est préférable en Java que chaque classe soit définie dans un fichier qui lui est propre. Le nom du fichier doit être le nom de la classe qu'il contient, auquel on ajoute le suffixe ".java". Les noms de classe doivent être constitués de mots acollés dont la première lettre est une majuscule. Enfin, chaque classe écrite doit nécessairement être accompagnée de sa documentation explicite au format Javadoc.
Nous allons nous intéresser dans cette feuille de td à la manipulation de composants géométriques localisés par des coordonnées cartésiennes dans un espace à deux dimensions, puis à trois dimensions.

▸ **Exercice 1.** *[Définition de classe] Écrire une classe* `Point` *représentant les points dans un espace à deux dimensions. La classe* `Point` *sera équipée d'une méthode* `translate(int dx, int dy)` *réalisant la translation du point. Attachez-vous à restreindre les visibilités des attributs, méthodes et constructeurs au strict nécessaire.*

▸ **Exercice 2.** *[membres statiques] Modifiez la classe* `Point` *pour qu'à chaque instance soit* automatiquement *associé un nom de la forme* `Point1`, `Point2`, *etc. Le nom est stocké à l'intérieur de chaque instance, sous la forme d'une chaîne de caractères.*

▸ **Exercice 3.** *[Redéfinition de méthodes, appel de méthode, surcharge de méthode] Créez une instance de la classe* `Point` *et appeler la méthode* `toString()` *dessus. Que constatez-vous ? Comment y remédier ? Écrivez une méthode* `boolean equals(Point)` *permettant la comparaison de deux instances de* `Point`. *Peut-on faire désigner une instance de la classe* `Point` *à une référence déclarée de type* `Object` *? Expliquez. Qu'affiche le code suivant (expliquez) :*


```
Point p1 = new Point(1,1);
Point p2 = new Point(1,1);
Object p3 = new Point(1,1);
System.out.println(p1.equals(p2));
System.out.println(p1.equals(p3));
```

Que faut-il faire pour que le résultat soit le résultat attendu ? Faites-le !

▸ **Exercice 4.** *[héritage] Écrivez une classe* `Point3D` *représentant les points dans un espace à trois dimensions. Pour éviter de réécrire le code de la classe* `Point`, *vous ferez hériter* `Point3D` *de* `Point`. *Pensez à redéfinir les méthodes qui ont besoin de l'être. Faut-il redéfinir la méthode* `translate()` *à deux paramètres ?*

▸ **Exercice 5.** *[membres statiques] Comme pour les points en deux dimensions, faites la création automatique de noms pour les points en trois dimensions*

Dans la suite, un disque est représenté par son centre (un point), et son rayon. Toutes les figures géométriques sont équipées d'une méthode qui en calcule la surface, d'une autre en réalisant la translation, d'une méthode de comparaison et d'une méthode `toString()` adaptée.

▸ **Exercice 6.** *Écrire la classe* `Disque`. *On l'équipera de deux constructeurs, un prenant en argument un point et un rayon, l'autre les coordonnées du centre et un rayon. Le code suivant est-il correct (au sens de Java) :*

```
Disque d = new Disque(new Point3D(1,2,3),4.0);
```

▸ **Exercice 7.** *On voit un anneau comme un disque équipé d'un "rayon interne". Implantez la classe* `Anneau`, *en la faisant hériter de* `Disque`. *Donnez une autre manière d'implanter la classe* `Anneau` *(inutile d'écrire la classe correspondante, donnez juste l'idée).*

▸ **Exercice 8.** *Si ça n'est pas déjà fait, contrôler que le rayon interne est plus petit que le rayon externe dans le constructeur de la classe* `Anneau`.

▸ **Exercice 9.** *Généralisez* `Disque` *et* `Anneau` *à trois dimensions. On considère que la surface d'un anneau en trois dimensions est sa surface extérieure (la surface visible quand on regarde l'anneau à trois dimensions).*

▸ **Exercice 10.** *Écrire une classe* `Carre`*, où un carré est représenté par son point supérieur gauche et la longueur de ses cotés, qu'on supposera parallèles aux axes du repère. Comme les autres figures géométriques, les carrés possèdent une méthode de translation et une méthode calculant sa surface.*

▸ **Exercice 11.** *[interface] Nous souhaitons maintenant ranger des objets ayant une surface, c'est-à-dire des instances des quatres classes précédement écrites, dans un tableau. Ensuite, nous souhaitons écrire une boucle calculant la somme des surfaces des éléments du tableau. Voici le code :*

```
tab[0] = new Disque(new Point(1,2),3.0);
tab[1] = new Sphere(new Point3D(5,6,7),8.0);
tab[2] = new Anneau(new Point(4,2),5.0,4.0);
tab[3] = new Carre(new Point(1,2),3.0);
double s;
for (int i=0; i<tab.length; ++i)
  s += tab[i].surface();
System.out.println(s);
```

Comment déclarer `tab` *? Quelles sont les modifications de code à effectuer sur les classes des figures géométriques précédement écrites ?*

▸ **Exercice 12.** *[ordres, comparateurs] On souhaite maintenant trier notre tableau de figures par ordre croissant de surface, en utilisant la méthode de tri de l'API* `java.util.Arrays.sort(Object[])`*. Cette méthode utilise l'*ordre naturel *sur les objets.* Cet ordre naturel *est défini par les objets implantant l'interface* `java.lang.Comparable`*, qui contient une méthode* `int compareTo(Object)` *qui compare l(objet appelant avec l'objet dont la référence est passée en paramètre. Elle retourne une valeur négative si le premier est inférieur au second, une valeur positive si le premier est supérieur au second et zéro si les deux sont identiques. Les méthodes* `compareTo(Object)` *et* `equals(Object)` *devraient être* cohérentes*, c'est-à-dire qu'elles devraient vérifier le contrat :*

```
a.equals(b)==true si et seulement si a.compareTo(b)==0
```

De plus, si la méthode `compareTo(Object o)` *est bien écrite, alors elle doit vérifier un contrat de bon sens, dont les détails sont dans sa documentation. Modifier toutes les classes des figures géométriques pour qu'elles implantent l'interface* `Comparable`*, en définissant l'ordre naturel (la méthode* `compareTo(Object)`*) comme l'ordre croissant des surfaces des figures. Créer un tableau de figures géométriques, le trier avec la méthode* `java.util.Arrays.sort(Object[])`*, et afficher le résultat.*

▸ **Exercice 13.** *Il existe une autre manière que l'ordre naturel pour définir des ordres sur des objets : les* objets comparateurs*. Ce sont des objets dont la spécialité est de comparer d'autres objets entre eux. Ils implantent l'interface* `java.util.Comparator` *qui contient :*

- *une méthode* `int compare(Object o1, Object o2)` *qui retourne une valeur négative si l'objet référencé par* `o1` *est plus petit que celui référencé par* `o2`*, une valeur positive si l'objet référencé par* `o1` *est plus grand que celui référencé par* `o2`*, zéro si les deux objets sont égaux ;*
- *une méthode* `boolean equals(Object)` *qui permet de comparer deux objets comparateurs entre eux.*

Définir une classe d'objets comparateurs de figures. En créer une instance, et l'utiliser pour trier un tableau de figures : vous chercherez pour celà dans la classe `java.util.Arrays` *une méthode* `sort` *qui vous semble adaptée.*

▸ **Exercice 14.** *Nous souhaitons maintenant écrire une classe dont les instances servent à stocker des figures géométriques. La classe disposera des méthodes suivantes :*

- `add(...)` *qui permet d'ajouter une figure à la suite des autres ;*
- `size()` *qui retourne le nombre de figures contenues ;*
- `get(int pos)` *qui retourne la posième figure insérée ;*
- `surface()` *qui calcule la somme des surfaces des figures contenues ;*

Les figures seront stockées dans un tableau attribut privé de la classe. Écrire la classe.

▸ **Exercice 15.** *Même exercice que le précédent, mais cette fois-ci les figures seront stockées dans un objet de type* `java.util.ArrayList` *attribut privé de la classe.*

▸ **Exercice 16.** *[classes abstraites] Qu'ont en commun les deux classes écrites dans les deux exercices précédents ? Comment factoriser le code commun aux deux classes ? Faire les modifications nécessaires.*

▸ **Exercice 17.** *[classes internes, itérateurs] Écrire, dans la classe abstraite, une classe interne implantant l'interface* `java.util.Iterator` *permettant de récupérer un par un les éléments appartenant au conteneur.*

▸ **Exercice 18.** *Améliorer votre itérateur pour qu'il soit* fail-fast.