3. L'héritage

♣ **Exercice 3.1.** *Bibliothèque.* Pour la gestion d'une bibliothèque on nous demande d'écrire une application manipulant des documents de nature diverse : des livres, des dictionnaires, etc.

Tous les documents ont un *numéro d'enregistrement* et un *titre*. Les livres ont, en plus, un *auteur* et un *nombre de pages*, les dictionnaires ont une *langue* et un *nombre d'articles*.

Ces divers objets doivent pouvoir être manipulés de façon homogène en tant que documents.

A. Définissez les classes `Document`, `Livre` et `Dictionnaire`. Donnez à chacune le constructeur qui prend autant arguments qu'il y a de variables d'instance à initialiser, et qui se limite à recopier les valeurs des arguments dans les variables correspondantes. Définissez également les accesseurs (`getNumero()`, `getTitre()`, etc.) nécessaires, ainsi que la méthode `toString()` habituelle.

B. Définissez une classe `Bibliothèque` réduite à une méthode `main` permettant de tester les classes et méthodes précédentes, ainsi que celles que vous écrirez pour la question suivante.

C. Définissez la classe `ListeDeDocuments` permettant de créer une liste (vide) et d'y ajouter des documents. La liste peut être représentée par un objet `java.util.Vector`. Dans cette classe, définissez les méthodes

- `void add(Document d)` qui ajoute un document à la liste,
- `void afficherTousLesDocuments()` qui affiche tous les documents de la liste,
- `void afficherTousLesLivres()` qui affiche tous les livres (uniquement) de la liste.

• • •

♣ **Exercice 3.2.** *Une interface, plusieurs implémentations.* Dans une précédente série d'exercices nous avons défini une classe pour manipuler les points du plan. Nous en avons réalisé une première version utilisant les coordonnées cartésiennes des points, puis nous avons constaté que le respect du principe d'encapsulation permettait de changer l'implémentation (par exemple, en remplaçant les coordonnées cartésiennes des points par leurs coordonnées polaires) tout en conservant les prototypes des méthodes publiques de la classe, ce qui garantissait qu'il n'y aurait rien à modifier dans les applications utilisatrices de la classe.

Le problème posé ici est le suivant : faire *cohabiter* deux classes distinctes `PointCart` (point représenté par ses coordonnées cartésiennes), `PointPol` (point représenté par ses coordonnées polaires) et une interface `Point` rendant les applications utilisatrices indépendantes de l'implémentation choisie dans chaque cas.

Faites en sorte qu'un point, quelle que soit sa représentation interne, puisse être construit aussi bien à partir de ses coordonnés cartésiennes qu'à partir de ses coordonnées polaires. Par exemple, on peut convenir que lorsque le constructeur d'un point a deux arguments, ce sont ses coordonnées cartésiennes, tandis que s'il a trois arguments, alors les deux premiers sont ses coordonnées polaires (le troisième est sans utilité).

• • •

♣ **Exercice 3.3.** *Héritage ou composition ?* Un journal est une collection d'*événements*. Un événement est fait de deux champs : une date et un texte. Un journal doit posséder les opérations suivantes :

- *unJournal*`.add(`*unTexte*`)` - ajout au journal d'un événement composé de la date courante (que cette méthode obtient automatiquement) et du texte indiqué ;
- *unJournal*`.toString()` - renvoie une chaîne de caractères contenant tous les événements du journal ;
- *unJournal*`.toString(`*uneChaine*`)` - renvoie une chaîne de caractères contenant tous les événements dont le texte contient la chaîne indiquée ;

A. Écrivez une classe `Evenement` et une classe `Journal`. La classe `Evenement` sera aussi simple que possible (les deux champs indiqués, un constructeur élémentaire et la méthode `toString` habituelle).

Dans cette question, la classe `Journal` doit être une sous-classe de `java.util.Vector`.

B. Écrivez une classe `TestJournal` avec une méthode `main` simple qui lit et exécute des commandes comme :

`+ texte`	ajout au journal de l'événement ayant le texte indiqué ;
`?`	listage de tous les événements du journal ;
`? chaîne`	listage des événements qui contiennent la chaîne indiquée ;
`*`	abandon du programme.

C. Réécrivez la classe `Journal` mais, au lieu d'en faire une sous-classe de `Vector` mettez-y un membre de type `Vector`. La classe `TestJournal` doit fonctionner sans changement.

D. Etre ou avoir ? Dans la question *A* vous avez lié les classes `Vector` et `Journal` par un lien *d'héritage* : un objet `Journal` « est » un objet `Vector` ; dans la question *C* vous les avez liés par un lien de *composition* : un objet Journal « a » un objet `Vector`. D'après vous, quels sont les mérites de l'une et l'autre manière de faire ?

Dans quel cas contrôlez-vous mieux le comportement d'un objet `Journal` ? Supposez q'on vous demande d'interdire les suppressions d'événements du journal ; est-il facile d'obtenir cela dans la solution *A* ?

Voyez-vous dans quelle situation l'héritage peut-il devenir préférable, voire nécessaire ?

• • •

♣ **Exercice 3.4.** *Objets fonctionnels.* On considère le problème suivant : comment déclarer une variable dont les valeurs sont des fonctions ? Comment faire qu'une fonction ait pour argument une autre fonction ?

Par exemple, la méthode `transformation` prend un tableau $(t_0, t_1, \ldots t_{n-1})$ et une fonction ϕ et renvoie le tableau $(\phi(t_0), \phi(t_1), \ldots \phi(t_{n-1}))$ obtenu en appliquant la fonction à chaque élément du tableau. Pour fixer les idées nous supposerons $t_i \in R$ et $\phi : R \to R$. Il s'agit donc d'écrire une fonction dont le prototype sera

```
double[] transformation(double[] t, Fonction phi)
```

A. Définissez l'interface `Fonction`, composée d'une seule méthode, nommée `appel`, qui prend un `double` et rend un `double`.

Écrivez alors la méthode `transformation`. Pour l'essayer, écrivez une méthode `main` qui construit le tableau [0, 1, 2, 3, 4, 5, 6] et le transforme par la fonction $x \mapsto x^2$, de sorte à obtenir le tableau [0, 1, 4, 9, 16, 25, 36].

B. Écrivez la méthode `double[] filtre(double[] t, Predicat condition)` qui prend un tableau t comme précédemment et un prédicat (c'est-à-dire une fonction rendant un booléen) représentant une condition et qui construit le tableau formé des éléments de t pour lesquels la condition est satisfaite.

Il y a un petit problème (à résoudre) liée au fait que le tableau produit a probablement moins d'éléments que le tableau donné.

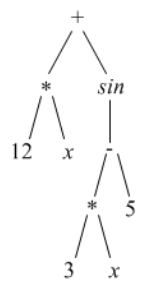
C. Écrivez la méthode `cumul(double[] t, Operation ope)` qui prend un tableau t comme précédemment et une opération binaire *ope* et qui renvoie le résultat de l'application de *ope* aux éléments du tableau. Par exemple, si $t = (t_0, t_1, \ldots t_{n-1})$ et *ope* représente l'addition, alors la fonction renvoie $t_0 + t_1 + \ldots + t_{n-1}$.

`Operation` est le nom d'une interface composée de deux méthodes : `elementNeutre`, qui renvoie la valeur de l'élément neutre pour l'opération considérée, et `operateur`, qui représente l'opération binaire elle-même.

• • •

♣ **Exercice 3.5.** *Représentation d'expressions arithmétiques.* Dans cet exercice on vous demande de définir un ensemble de classes pour représenter des fonctions d'une variable formées avec des constantes, des occurrences de la variable x, les quatre opérations arithmétiques $+$, $-$, $\times$, $/$, et des appels de quelques fonctions convenues comme *sin*, *cos exp*, *log*, etc. Par exemple : $12x + sin(3x - 5)$.

Dans un programme, une expression comme celle-là peut être efficacement représentée par une structure arborescente, organisée comme le montre la figure ci-contre, faite de feuilles (les constantes et les variables), de nœuds à deux « descendants » (les opérateurs binaires) et de nœuds à un descendant (les fonctions d'une variable).



Les classes qu'il faut définir sont destinées à représenter les nœuds d'un tel arbre. Il y en a donc de plusieurs sortes :

- un nœud représentant une constante porte un nombre, la valeur de la constante,
- un nœud représentant une occurrence de la variable x ne porte aucune autre information,
- un nœud représentant une addition, une soustraction, une multiplication ou une division porte deux informations : les expressions qui sont ses opérandes,

– un nœud représentant l'appel d'une fonction porte comme unique information l'expression qui en est l'argument.

Définissez les classes suivantes (la marge à gauche reflète la relation *implements* ou *extends*) :

`Expression` – interface représentant ce qu'ont en commun toutes les expressions arithmétiques (c'est-à-dire toutes les sortes de nœuds de notre structure arborescente). Elle se compose d'une seule méthode : `public double valeur(double x);` qui renvoie la valeur de l'expression pour la valeur de x donnée.
Bien entendu, toutes les classes concrètes de cette hiérarchie devront fournir une définition de cette méthode `valeur`. Elles fourniront aussi une redéfinition intéressante de la méthode `String toString()`.

`Constante` – classe concrète dont chaque instance représente une occurrence d'une constante. Cette classe a un membre : la valeur de la constante.

`Variable` – classe concrète dont chaque instance représente une occurrence de la variable x. Cette classe n'a besoin d'aucun membre.

`OperationBinaire` – classe abstraite rassemblant ce qu'ont en commun tous les opérateurs à deux opérandes. Elle a donc deux membres d'instance, de type `Expression`, représentant les deux opérandes, et le constructeur qui va avec.

`Addition`, `Soustraction`, `Multiplication`, `Division` – classes concrètes pour représenter les opérations binaires. C'est ici qu'on trouve une définition pertinente de la méthode `valeur` promise dans l'interface `Expression`.

`OperationUnaire` – classe abstraite rassemblant ce qu'ont en commun tous les opérateurs à un opérande. Elle doit avoir un membre d'instance, de type `Expression`, représentant l'opérande en question.

`Sin`, `Cos`, `Log`, `Exp`, etc. – classes concrètes pour représenter les fonctions standard. Ici on doit trouver une définition pertinente de la méthode `valeur` promise dans l'interface `Expression`.

On ne vous demande pas de résoudre le problème (difficile) de la « lecture » d'un tel arbre, c'est-à-dire de sa construction à partir d'un texte, par exemple lu à la console. En revanche, vous devez montrer que votre structure est bien adaptée au calcul de la valeur de l'expression pour une valeur donnée de la variable x. Pour cela, vous exécuterez un programme d'essai comme celui-ci :

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

        /* codage de la fonction f(x) = 2 * sin(x) + 3 * cos(x) */
    Expression f = new Addition(
            new Multiplication(
                    new Constante(2), new Sin(new Variable())),
            new Multiplication(
                    new Constante(3), new Cos(new Variable())));

        /* calcul de la valeur de f(x) pour quelques valeurs de x */
    double[] tab = { 0, 0.5, 1, 1.5, 2, 2.5 };

    for (int i = 0; i < tab.length; i++) {
        double x = tab[i];
        System.out.println("f(" + x + ") = " + f.valeur(x));
    }
}
...
```

L'exécution de ce programme produit l'affichage de :

```
f(0.0) = 3.0
f(0.5) = 3.5915987628795243
f(1.0) = 3.3038488872202123
f(1.5) = 2.2072015782112175
f(2.0) = 0.5701543440099361
f(2.5) = -1.2064865584328883
```

• • •

♣ **Exercice 3.6.** *L'héritage multiple en Java.* En Java l'héritage est simple : chaque classe a une et une seule super-classe (sauf `Object`, qui n'a pas de super-classe). Mais alors, comment faire lorsque les objets d'un certain type doivent être considérés comme appartenant à deux hiérarchies d'héritage, ou plus ?

> *Question annexe* : pourquoi serait-on obligé de déclarer une classe C comme héritant de deux autres classes A et B ? C'est-à-dire, si on veut que les membres de A et ceux de B soient membres de C, pourquoi ne suffit-il pas de mettre dans C une variable de type A et une variable de type B ?
> *Réponse.* Il est nécessaire que C soit sous-classe de A [resp. B] si on veut pouvoir mettre un objet C à un endroit où un objet A [resp. B] est prévu. Par exemple, si on dispose d'une méthode déclarée avec un argument formel de type A [resp. B] et qu'on veut l'appeler avec un argument effectif de type C alors il faut que la classe C soit sous-classe de A [resp. B].

Exemple (un peu tiré par les cheveux, à vrai dire, mais c'est un exemple...) :

- un `Personnel` est un fonctionnaire relevant du MENESR (Ministère de l'Éducation Nationale, de l'Enseignement Supérieur et de la Recherche) ; il est défini par un certain nombre de variables d'instances privées, chacune associée à une méthode publique de même nom qui permet d'en obtenir la valeur :
  - `nom` (de type `String`),
  - `numeroSS` (de type `long`),
  - `anneeNaissance` (de type `int`),
  - etc.
- un `Enseignant` est une entité pouvant dispenser des cours ; cela peut être un personnel de l'éducation nationale, mais pas nécessairement (un cours peut être donné par un intervenant extérieur, un magnétoscope, un âne parlant...) ; comme précédemment, cela se définit par un ensemble de variables d'instance privées avec des méthodes publiques d'accès
  - `matiere` (de type `String`),
  - `classeNiveau` (de type `int`),
  - etc.
- un `Chercheur` est une autre entité non nécessairement membre de l'éducation nationale, consacrant une partie de son temps à des recherches ; encore une fois, des variables d'instance privées et des méthodes publiques d'accès :
  - `domaineDeRecherches` (de type `String`),
  - `nombrePublications` (de type `int`)
  - etc.

Nous supposerons que `Personnel`, `Enseignant` et `Chercheur` sont trois classes concrètes, précédemment définies et parfaitement opérationnelles. En outre, nous supposerons que des méthodes sont disponibles par ailleurs, qui prennent pour argument des objets de chacune de ces trois classes :

```
static void gestionCarriere(Personnel unPersonnel);
static void emploiDuTemps(Enseignant unEnseignant);
static void rapportActivite(Chercheur unChercheur);
```

L'exercice est le suivant : définir une classe `PersonnelEnseignantChercheur` destinée à représenter des personnels de l'éducation nationale qui sont *en même temps* des enseignants et des chercheurs. Il faut utiliser les trois classes `Personnel`, `Enseignant` et `Chercheur`, et la nouvelle classe doit être définie de telle manière que ses instances puissent être données pour argument aux méthodes ci-dessus, éventuellement au prix d'un (très léger) changement de nom de certains types.

• • •