Programmation Orientée Objet

TP abstraction – Expressiez-vous !

---

1 Introduction

Ce TP a pour objectif de manipuler des expressions arithmétiques [1] fonction d'une ou plusieurs variables réelles : affichage, évaluation, etc.

La notation usuelle d'une expression est dite *infixée*. Cette notation impose des parenthèses en fonction des priorités des opérateurs utilisés. Voici quelques exemples en notation infixée :

$$E_{infixée}(y) = 3 * y \tag{1}$$

$$F_{infixée}(x) = (x + x) * 5 \tag{2}$$

$$G_{infixée}(a, b) = -3.5 * sin(a + b) \tag{3}$$

Une autre notation est possible, dite *préfixée* ou *polonaise*, dans laquelle un opérateur est toujours placé *devant* ses opérandes : *operateur operande*1 *operande*2. On donne ci-dessous les mêmes expressions `E` `F` et `G`, cette fois ci en notation préfixée. Notez qu'il n'est plus nécessaire d'utiliser des parenthèses.

$$E_{prefixée}(y) = * \ 3 \ y \tag{4}$$

$$F_{prefixée}(x) = * \ + \ x \ x \ 5 \tag{5}$$

$$G_{prefixée}(a, b) = * \quad -3.5 \quad sin \quad + \quad a \quad b \tag{6}$$

Pour représenter une expression arithmétique, on construira en mémoire un arbre *d'objets* Java. L'arbre est une traduction directe de la notation préfixée. Chaque noeud représente soit une valeur (*e.g.* -3.5), soit une constante (*e.g* `x`), soit un opérateur binaire (*e.g.* `+`), soit un opérateur unaire (*e.g.* `sin`). Les arbres représentant les trois exemples d'expressions sont donnés figure 1.

On ne considérera dans ce TP que quelques opérateurs (l'extension à d'autres opérateurs ne présentant aucune difficulté) :

- des constantes (des valeurs flottantes)
- des variables
- les opérateurs binaires `+` et `*`
- les opérateurs unaires `sin` et `cos`

2 Classes des noeuds d'un arbre d'expression

Pour représenter les expressions, on implantera les classes Java décrites à la figure 2.

1. TP 100% vegan cette semaine, fichons la paix aux bestioles.

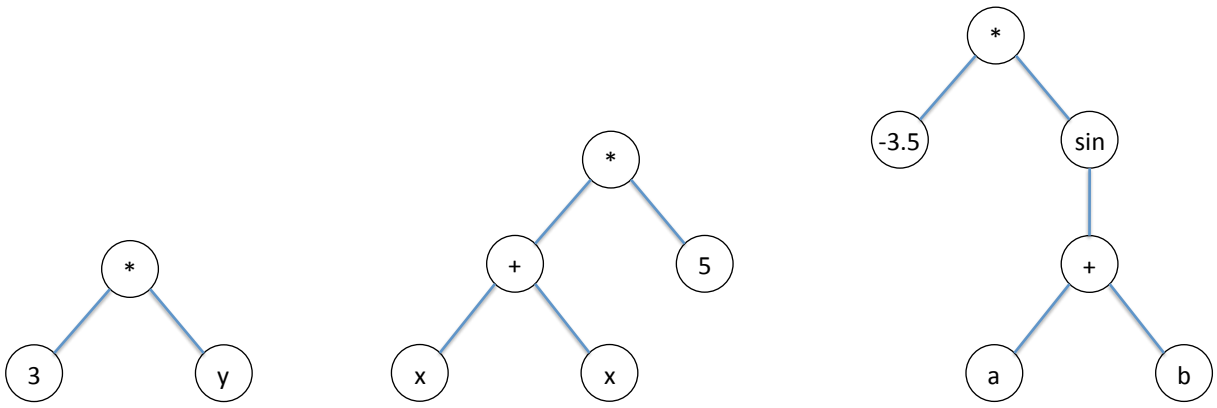

FIGURE 1 – Les arbres d'objets représentant E(x), F(x) et G(a,b)

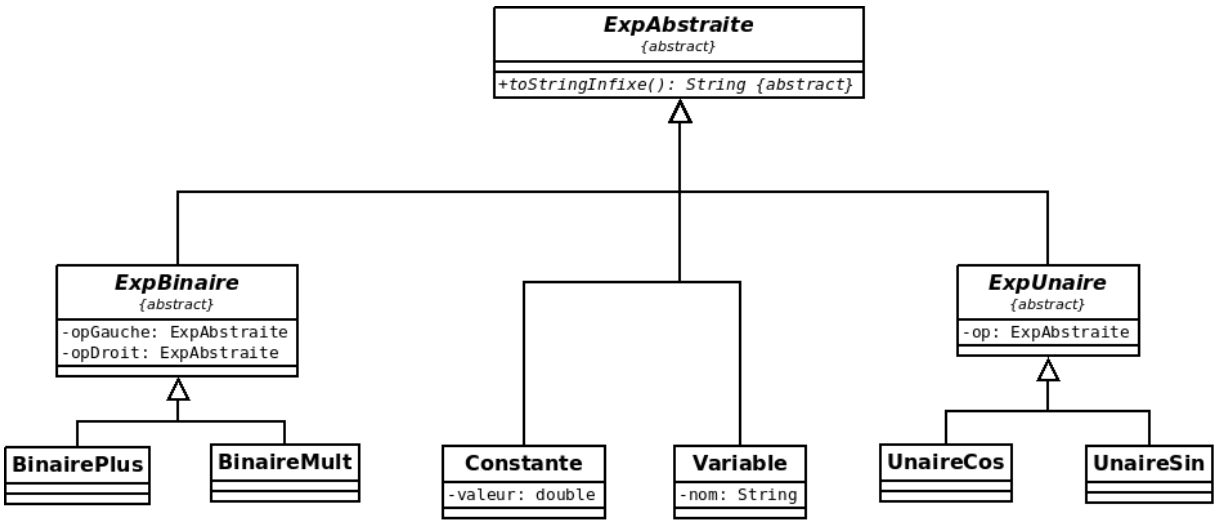

FIGURE 2 – Hiérarchie de classes pour les expressions.

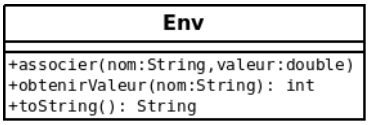

FIGURE 3 – Classe d'environnement pour stocker les associations entre nom de variable et valeur.

Dans ce diagramme, repérez les classes *abstraites* représentant les concepts génériques (expression, expression binaire, etc.) et les classe *concrètes* représentants les types réellement instanciés (constante, variable, opérateur plus, etc.).

La classe abstraite `ExpBinaire` déclare deux arguments pour les opérandes droit et gauche. Notez bien que ce sont des instances de `ExpAbstraite`, donc une expression quelconque. Le même raisonnement s'applique à la classe `ExpUnaire`, avec un seul opérande.

Enfin, remarquez que la classe `ExpAbstraite` déclare une méthode abstraite : `String toStringInfixe()` retournant une représentation textuelle de l'expression, en notation infixée (usuelle) totalement parenthésée. Bien entendu, cette méthode abstraite devra être définie dans les sous-classes. Pour simplifier, on utilisera systématiquement des parenthèses. Par exemple, la chaîne représentant l'expression `F` sera :

$$((x + x) * 5) \tag{7}$$

Les classes définies ci-dessus permettent de représenter une expression quelconque par un arbre *d'objets* Java. Dans cet arbre, les noeuds sont des instances de l'une des classes concrètes d'opérateurs, et les feuilles des instances des classes `Constante` ou `Variable`.

Le programme de test suivant (disponible sur le site du cours) illustre la création d'une constante simple puis des trois arbres E(x), F(x) et G(a,b), et leur affichage infixé.

```
1  public class TestAffichageInfixe {
2      public static void main(String[] args) {
3          ExpAbstraite exp;
4
5          // teste l'expression préfixée 42
6          exp = new Constante(42) ;
7          System.out.println(exp.toStringInfixe());
8
9          // teste l'expression préfixée *  y  3
10         exp = new BinaireMult(new Variable("y"), new Constante(3)) ;
11         System.out.println(exp.toStringInfixe());
12
13         // teste l'expression préfixée * + x x 5
14         exp = new BinaireMult(
15                 new BinairePlus(
16                     new Variable("x"),
17                     new Variable("x")
18                     ),
19                 new Constante(5)
20                 );
21         System.out.println(exp); // et pas toStringInfixe; comme ca!
22
23         // teste l'expression préfixée * -3.5 sin + a b
24         exp = new BinaireMult(
25                 new Constante(-3.5),
26                 new UnaireSin(
27                     new BinairePlus(
28                         new Variable("a"),
29                         new Variable("b")
30                         )
31                     )
32                 );
33         System.out.println(exp);
34     }
35 }
```

Question 1 Pour commencer, implantez uniquement les classes `ExpAbstraite` et `Constante` puis testez avec le programme fourni. Prenez garde à la visibilité des attributs et définissez uniquement les constructeurs et méthodes nécessaires.

Question 2 Redéfinissez la méthode `String toString()` dans la classe `ExpAbstraite` (et uniquement dans cette classe !) de telle sorte qu'elle retourne une chaine conforme à `"Je suis une expression et me voila en notation infixée : <ici, la notation infixée>"`.

Question 3 Implantez maintenant les opérateurs binaires, et testez (vous décommenterez les tests au fur et à mesure des questions).

Note : les opérateurs unaires suivent la même logique. Il n'est pas nécessaire de les implémenter tout de suite, autant le faire en fin de TP après avoir traité l'évaluation des expressions.

3 Encore plus de factorisation ?

Il existe encore de la redondance dans les classes concrètes d'expressions (que vous verriez encore plus en ajoutant d'autres opérateurs : division, tan, carré, etc.). Par exemple pour l'affichage d'une expression binaire : tout est dans les filles, alors que le shéma *afficher l'opérande gauche, puis l'opérateur, puis l'opérande droit* est toujours identique.

Question 4 Comment, en ajoutant quelques méthodes ou attributs, factoriser le plus de code possible au niveau des classes abstraites intermédiaires `ExpBinaire` (et `ExpUnaire`), et ne garder dans les filles que ce qui leur est réellement spécifique ? Y a-t-il un intérêt par rapport à la visibilité des attributs ?

4 Evaluation d'une expression

Pour évaluer numériquement une expression, il faut substituer chaque variable par sa valeur réelle. ll faut donc associer une valeur à chacune des variables.

Pour cela, on va d'abord écrire une classe d'environnement `Env` dont le rôle est de stocker les associations entre les noms de variables et les valeurs de ces variables. Le diagramme de cette classe est donné figure 3. Suivant le principe d'encapsulation, seule l'API publique est définie ; le reste ne regarde que vous.

Question 5 Implantez la classe `Env` associant noms de variables et valeurs.

Le diagramme fourni ne précise pas le/les attribut(s) de la classe ; à vous donc de les définir. Le plus simple est certainement d'avoir recours à un dictionnaire, ou "table associative", par exemple de type `HashMap<String, Double>`. Cette structure est présentée dans la fiche Introduction aux Collections, disponible sur le site du cours. Consultez également la Javadoc de `HashMap` : `http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html`

Question 6 Méthode d'évaluation

Déclarez dans la classe `ExpAbstraite` une nouvelle méthode abstraite `double evaluer(Env env)`, dont le rôle est de calculer la valeur de l'expression en fonction des valeurs des variables stockées dans l'environnement `env` passé en paramètre. Définissez ensuite cette méthode dans les classes filles appropriées, en factorisant toujours au maximum.

Notez qu'il n'est pas possible d'évaluer une expression si une variable de l'expression n'existe pas dans l'environnement. Une gestion d'exceptions, dans la méthode `obtenirValeur` de l'environnement et dans la hiérarchie des expressions semble donc appropriée...

Question 7 Testez l'évaluation au moyen de la classe dans `TestEval.java` fournie. C'est le même programme que précédemment avec en plus l'évaluation des expressions, par exemple :

```
1 System.out.println("Valeur de l'expression : " + exp.evaluer(env));
```

Note : un dernier programme `TestParser.java` est fourni, qui parse des expressions saisies par l'utilisateur au clavier. *A n'utiliser que si vous en avez le temps !!*

5 Extension possible : fonction dérivée

En supposant qu'on ne s'intéresse qu'aux expressions d'une seule variable, `x` par exemple, ajouter à la hiérarchie une méthode qui calcule la dérivée d'une expression donnée. Cette méthode `ExpAbstraite calculerDerivee()` retourne une nouvelle expression fonction de `x` correspondant à la fonction dérivée.

Variante : passer à cette méthode le nom de la variable par rapport à laquelle on veut calculer la dérivée.