FIGURE 1 – ISIMA

FIGURE 2 – Université de Clermont-Ferrand

TP 5 POO :

Réalisation d'équations à l'aide de classe

Exercice 1 (Expressions arithmétiques) On se propose ici de créer une structure de donnée pour représenter et évaluer une expression arithmétique.

Une expression arithmétique sera représentée sous la forme d'une structure composite de type arbre, dont les nœuds pourront être de trois types :

— une constante, qui contient une valeur numérique (`double`),
— une variable, qui possède un nom,
— une opération binaire, qui possède deux fils qui sont eux même des expressions.

CLIQUOT Théo
Professeur : Yannick Loiseau
Travail à rendre pour le : 1/05/2021

Table des matières

1 Division du problème et choix fait **2**
1.1 Différentes classe . . . 2
1.2 Choix fait . . . 2

2 Explication du code **2**
2.1 Nœud . . . 2
2.2 NoeudConstante . . . 3
2.3 NoeudVariable . . . 3
2.4 NoeudOperation . . . 3
2.4.1 NoeudAddition . . . 3
2.4.2 NoeudSoustraction . . . 3
2.4.3 NoeudDivision . . . 3
2.4.4 NoeudProduit . . . 3

3 Main (test) **4**

4 Conclusion **4**

Table des figures

1 ISIMA . . . 1
2 Université de Clermont-Ferrand . . . 1

1 Division du problème et choix fait

1.1 Différentes classe

Le travail principal demandé est de créer un arbre représentant une équation mathématique, on va donc dans un premier temps devoir créer une classe `Nœud` qui va accueillir notre valeur à ce nœud mais aussi des fonction que nous verrons plus tard. Cependant chacun de ces nœuds peuvent être divisé en 3 sous nœud différents les uns des autres. Une façon assez intuitive serait donc de créer 3 sous-classe qui vont hériter de `Nœud` qui seront `NoeudOperation`, `NoeudConstante` et `NoeudVariable` respectivement pour représenter une opération, une constante et une variable. De plus vu qu'on ne va pas instancier d'objet avec la classe `Nœud`, on peut la rendre *abstract*, ce qui permettra en plus de définir des méthodes *abstract*. Il n'y aura pas à proprement parlé d'une classe `Arbre` puisque c'est seulement un `NoeudOperation` si notre arbre n'est pas composé que d'une racine ou l'un des deux autres sinon. En ce qui concerne le test et l'utilisation du code tout sera dans un fichier à part nommé `testArbre`.

1.2 Choix fait

En ce qui concerne les choix faits, j'ai préféré regroupé le plus de fonction possible dans le `Nœud` afin de simplifier le plus l'utilisation pour l'utilisateur. En effet il n'a pas à se soucier de savoir si son Nœud est une opération, une constante ou une variable lorsqu'il l'utilise, il à sa classe `Nœud` et c'est tout. Cependant ce n'est pas la classe `Nœud` qui va contenir l'attribut correspondant à l'élément dans le Nœud car celui-ci peut changer de type en fonction de si il est une constante (Double) ou autre (String) (rien ne nous aurait empêcher de stocker la valeur dans un String pour la constante ou encore de stocker la variable dans `Nœud` avec comme Type englobant `Object` mais ça me semblait plus logique dans ce sens). La classe `Nœud` peut tout de même avoir accès à cette variable avec la méthode `getValue()`.

On aurait aussi pu créer une implémentation `Noeud` et on aurait alors définie chaque Noeud comme l'implémentant mais la première idée qui m'est venue était la classe abstraite englobante.

2 Explication du code

2.1 Nœud

La classe `Nœud` sera donc abstract et sans attributs. On a aussi surcharger la classe toString qui va juste renvoyer la valeur contenu dans le Nœud en String (évite de le spécifier dans `NoeudConstante` et `NoeudVariable`). On aura aussi en méthodes abstraite `eval()` et `operation()` qui nous serviront pour la question 3 et enfin une méthode statique `fromPrefixe()` qui va nous renvoyer l'Arbre (ou le Nœud) correspondant à une expression préfixe. Je l'ai mise dans le Nœud pour 2 raisons :

- Cela permet de rentrer n'importe quel préfixe et de le transformer en expression, que ce soit une préfixe composé d'opérations comme d'une seule constante. Cependant cela force à regarder la string et en fonction de ce que l'on a d'appeler la méthode `fromPrefixe()` de la bonne Classe `Nœud`, mais c'est un choix que j'ai fait comme expliqué là (voir **??**).
- On à une séparation de ce problème entre les classes, ce qui rends ce code plus lisible à mon sens avec un effet de yo-yo entre la classe `Noeud` et la classe `NoeudOperation`. Cependant si on veut rajouter un nouveau type de nœud on va être obligé de modifier cette méthode. ce qui empêche un peu le côté extensible, mais je ne vois pas d'autre moyen élégant pour rajouter d'autre élément nœud sans perturber la classe `Nœud`.

Enfin, on a 2 méthodes privées `isNumeric()` et `findIndexParentheseFermante()` qui servent pour notre méthode `fromPrefixe`. En effet lorsqu'on à le cas d'une opération (c'est à dire un "("), on peut se dire qu'on prends le tout et on rappelle la fonction `fromPrefixe()` récursivement, ce qui va nous faire avancer sans même connaître dans le cas d'une opération en englobant d'autres quand est ce qu'elle finit. Cependant connaître la fin d'une opération permet de diviser de façon plus esthétique et propre le problème à mon sens.

`isNumeric` va nous permettre de différencier les constantes des variables afin d'appeler la bonne classe.

2.2 NoeudConstante

Il n'y a rien de très dur à comprendre avec cette classe, pour `eval()` tout comme `getValue()` on retourne le Double stocké dans notre attribut `val`, si jamais on essaie d'utiliser la méthode `operation()` on renvoie une erreur car elle ne la supporte pas. Sinon pour le constructeur on va juste appeler le constructeur de la classe mère et initialiser `val` avec la valeur donné et enfin la méthode statique `fromPrefixe()` va juste instancier un `NoeudConstante()` avec comme valeur `val` celle donné.

2.3 NoeudVariable

Très similaire au `NoeudConstante` (voir 2.2) avec comme seule différence que la valeur stocké va être du type String et que notre méthode `eval()` va se servir de la HashMap donnée en argument pour faire correspondre un Double à notre nom de variable. On fait aussi attention que notre nom de variable ne soit pas utilisé pour représenté une opération mais ce n'est pas non plus interdit.

2.4 NoeudOperation

Le plus gros du travail se fait ici. `NoeudOperation` n'est qu'une classe abstraite qui va servie comme père pour chaque opération (classe) ; soit :

- `NoeudAddition` (voir 2.4.1)
- `NoeudSoustraction` (voir 2.4.2)
- `NoeudDivision` (voir 2.4.3)
- `NoeudProduit` (voir 2.4.4)

En plus de l'attribut `val` qui va représenter quelle opération on à dans ce nœud, on aura aussi 2 autres attributs `Nœud` représentant ce qu'il y a à gauche et à droite de l'opérateur. On va instancier un Nœud operation en se servant du constructeur de `Nœud` et en initialisant chacun de ses nœuds en attributs à null (rien). Sa méthode `eval()` va appliquer l'opération a son sous arbre gauche et son sous arbre droit. On aura donc la valeur de l'opération de tous ses nœud enfants et il nous restera plus qu'à faire de même jusqu'à la racine. La méthode `fromPrefixe()` va continuer sur la même idée mais en sens inverse : On va mettre la valeur de l'opération dans l'attribut `val` puis on va appeler `fromPrefixe()` pour obtenir les 2 sous arbres. La méthode `toString` à été surchargé pour pouvoir afficher non seulement le contenu de `val` mais aussi celle de ses sous-arbres. Enfin `creerNoeudOperation()` va seulement appeler la bonne sous classe en fonction de l'opération.

2.4.1 NoeudAddition

Les 4 classe qui suivent se ressemblent beaucoup, la seule différence est comment va être gérés la méthode `opération()` et la valeur stocké dans l'attribut `val` de `NoeudOperation`, pour celle ci se sera l'addition et "+" ;

2.4.2 NoeudSoustraction

La méthode `operation()` va renvoyer la soustraction de l'élément gauche avec le droit et on va stocker "-" dans `val`.

2.4.3 NoeudDivision

La méthode `operation()` va renvoyer la division de l'élément gauche avec le droit et on va stocker "/" dans `val`.

2.4.4 NoeudProduit

La méthode `operation()` va renvoyer le produit de l'élément gauche avec le droit et on va stocker "*" dans `val`.

3 Main (test)

C'est juste un main pour appeler et tester les différentes fonctions, une fois lancé on à un exemple et ensuite on peut le tester nous même en rentrant l'expression en préfixe ainsi que l'ensemble de nos variables (à noter qu'on transforme les ',' en '.' est ce pour prendre en compte la notation française ou anglaise de la virgule).

```
1  mot = mot.replace(',','.');
```

4 Conclusion

Ce compte rendu explique brièvement le code et se concentre surtout sur mes choix pour résoudre ce problème car il n'existe pas une seule et unique façon d'implémenter un arbre. Une Javadoc et des commentaires sont présent dans les programmes pour expliquer ce que fait chaque méthodes.