Réalité virtuelle et imagerie

CHASSAGNOL Rémi

April 29, 2021

1 Organisation des classes

Une expression binaire est un arbre qui contient la décomposition de l'expression, où chaque nœud contient une sous expression et chaque feuille contient une variable ou une constante. Pour modéliser cet arbre à l'aide de Java, on utilisera une classe **Expression** qui représentera l'expression et contiendra la racine de l'arbre de l'expression. L'arbre sera composé de **node** qui est une interface qui sera implémentée par des classes représentants des littéraux (variables ou constantes) ou des opérations (addition, soustraction, multiplication, division). Voici le diagramme UML des classes ci-dessous:

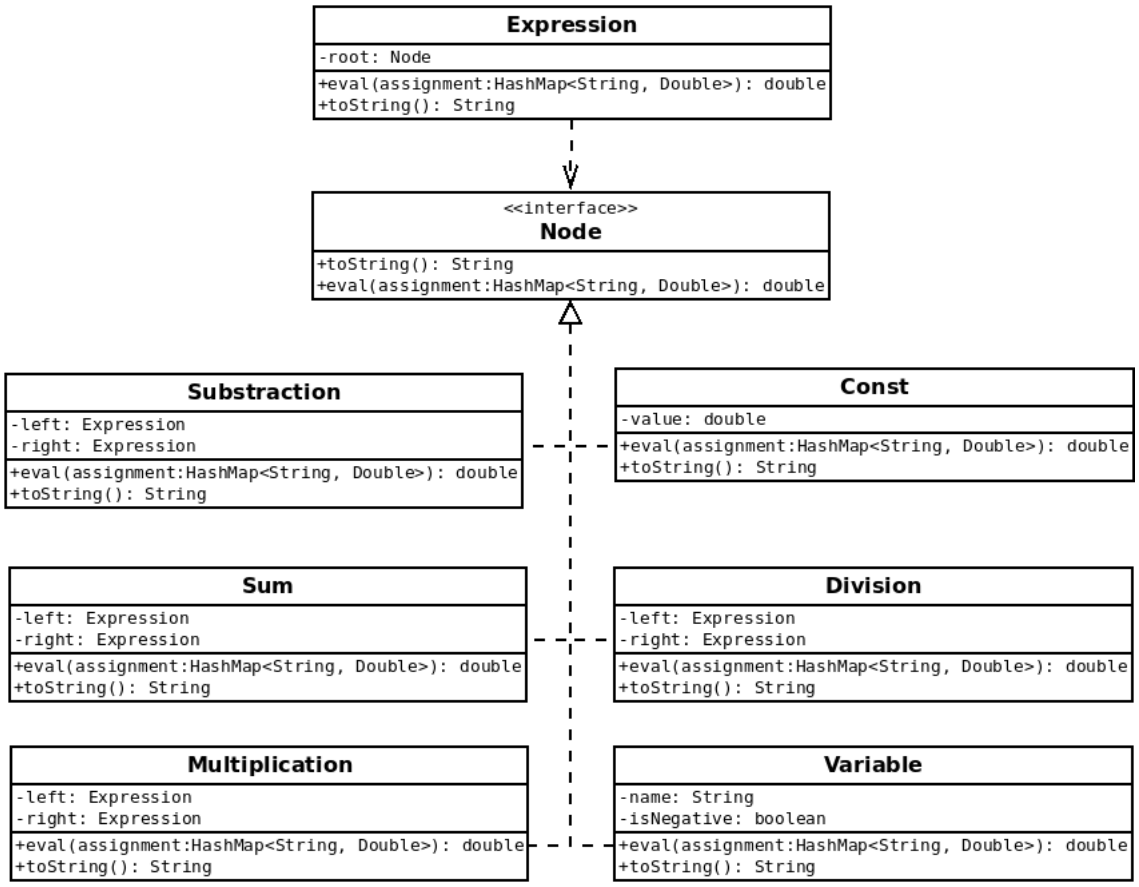

Figure 1: Diagramme des classes

1.1 Expression

La classe **Expression** possède un attribut privé **root** qui est la racine de l'arbre. Cette classe possède deux constructeurs, le premier permet de générer une expression depuis un nœud, et le second permet de générer une expression depuis une chaîne de caractères contenant une expression binaire écrite en utilisant la convention **préfixe** (ex: `(+ 2 2)`). Enfin, la classe possède deux méthodes publiques, **toString** qui permet d'afficher la formule (avec la convention infixe: `(2 + 2)`) et **eval** qui prend pour paramètre

une `HashMap<String, double>` contenant les couples **[nomVariable : valeurVariable]** (les valeurs affectées aux variables de la formules) et qui retourne le résultat de la formule sous forme de `double`.

A noter que cette classe possède aussi des méthodes privées utilisées pour extraire une expression des expressions de la chaîne de caractères donnée en paramètre au second constructeur.

1.2 L'interface nœud

Chaque nœud de l'arbre doit posséder deux méthodes: **eval** et **toString** permettant l'évaluation et l'affichage de la formule. Cependant, bien que nous les nœuds possèdent ces méthodes, les expressions binaires (addition, ...), les variables et les constantes ne possèdent pas les mêmes attributs ni les mêmes implémentations de ces méthodes, il est donc logique de préférer l'utilisation d'une interface à l'utilisation d'une classe abstraite. De ce fait, une **Expression** peut manipuler une addition et une variable de la même façon.

1.3 Les implémentations de nœud

1.3.1 Variables et constantes

La classe **variable** possède deux attributs privés, **name** qui contient le nom de la variable et **isNegative** qui est un booléen qui contient `vrai` si la variable est négative dans la formule (ce qui n'a rien à voir avec sa valeur, une variable **var** peut contenir les valeurs 5 ou -12 mais être utilisé en tant que **-var** dans la formule). La méthode **eval** retourne la valeur correspondant à **name** dans la `HashMap` **assignment** (et la modifie en fonction de **isNegative**). La méthode **toString** retourne `name` ou `"-" + name` en fonction de `isNegative`.

La classe **constante** possède un attribut privé double qui contient sa valeur et implémente les méthodes **eval** qui retourne la valeur et **toString** qui retourne la valeur sous la forme d'une chaîne de caractères.

1.3.2 Expressions binaires

Les quatre expressions binaires sont très similaires, elles possèdent toutes deux attributs privés **left** et **right** qui contiennent les sous expressions de gauche et de droite de l'expression (ex: `left + right`). La seule différence se trouve dans les méthodes **eval** et **toString**, **eval** va effectuer l'addition de **left** et **right** dans le cas de **Sum**, la soustraction dans le cas de **Substraction**, ... et **toString** va retourner `"left c right"` en remplaçant **c** par le caractère correspondant à l'opération.

2 Récupération d'une formule

2.1 Le constructeur de Expression

Pour rappel, la classe **Expression** possède deux constructeurs, le premier génère une expression binaire à partir de deux sous-expressions et le second prend en paramètre une chaîne de caractères qui contient une expression mathématique écrite selon la convention préfixe (de plus, on suppose que l'expression donnée est une expression binaire et non pas une constante ou une variable). Avant de commencer l'extraction de l'expression, on commence par remplacer tous les potentiels "bloques d'espaces" (plusieurs espaces à la suite) et tabulation par un espace simple, ce qui rend l'utilisation des indices dans la chaîne possible (par exemple, on devient sûr que le quatrième caractère de la chaîne est soit le début d'une opération binaire **'('**, soit le premier caractère d'un littéral) et permet de corriger de potentielles erreurs de l'utilisateur. Ensuite, on distingue quatre cas qui correspondent à quatre formes possibles de l'expression:

- **cas 1: (. x y)**, opération entre deux littéraux x et y,
- **cas 2: (. (...) x)**, opération entre une sous opération binaire suivie d'un littéral x,
- **cas 3: (. x (...))**, une opération entre un littéral x suivi d'une sous opération binaire,
- **cas 4: (. (...) (...))**, une opération entre deux sous-opérations binaires.

De ce fait, on sait exactement ce que l'on a à extraire, et donc quelle méthode appeler pour construire les parties gauche et droite de l'expression.

2.2 Les méthodes d'extractions

Ces méthodes servent uniquement au second constructeur de la classe **Expression** et ne doivent pas pouvoir être appelées par l'utilisateur de l'objet, elle sont donc définies comme étant privées pour respecter l'encapsulation de l'objet Expression.

2.2.1 Récupération d'une expression depuis une String

Cette méthode est une procédure qui va modifier (par effet de bord) le tableau **expressionParts** (qui doit contenir les partie gauche `[0]` et droite `[1]` de l'expression). Si une des parties de l'expression est un littéral, la case de **expressionParts** correspondant contiendra une chaîne de caractères vide à la fin de l'exécution ce qui permet de traiter les quatre cas vus précédemment dans le second constructeur de **Expression**.

Cette méthode se divise en deux parties, dans un premier temps, on récupère une première opération binaire en se servant de la variable **formulaLevel** pour compter les parenthèses (pour récupérer la bonne chaîne si il y a des parenthèses dans les parenthèses) et d'une boucle **do while**. A noter qu'à la fin de cette première partie, la première case de **expressionParts** contient la première opération binaire trouvée dans la chaîne, si la première partie de l'expression est un littéral (et la seconde, une opération binaire), on vient de récupérer la seconde partie.

Ensuite, on teste si il reste une opération binaire à extraire (on regarde l'indice de la prochaine **'('** et si c'est -1, c'est qu'il n'y en a pas), si c'est le cas, on récupère la fin de la chaîne dans **expressionParts[1]** et on se retrouve dans le **cas 4**. Dans le cas contraire, on regarde si le quatrième caractère est une **'('** (i.e. que la première partie de l'expression est une opération binaire), si c'est le cas, on ne fait rien et on se retrouve dans le **cas 2**, sinon, on échange les valeurs dans les cases de **expressionParts** pour se retrouver dans le **cas 3**.

2.2.2 Extraction d'un littéral

Cette méthode est appelée dans le **cas 1** et elle permet de récupérer les deux littéraux gauche et droit de l'expression.

2.2.3 Création d'un littéral

La méthode **createExpressionFromLitteral** permet de créer une **Expression** qui est une **constante** ou une **variable** à partir d'une chaîne de caractères. Cette méthode fait appel à la méthode **isNumeric** qui retourne **vrai** si la chaîne de caractères passée en paramètre contient un double. Pour cela, cette méthode fait un **try** pour tester si la méthode **Double.parseDouble** lève une exception **NumberFormatException**, si c'est le cas, la chaîne ne contient pas un double et donc c'est un nom de variable, sinon, elle contient la valeur d'une constante.

2.2.4 Création d'une expression

Une fois que l'on a les parties gauche et droite d'une expression, on peut faire appel à la fonction **createOperation** qui crée un nœud (qui sera affecté à l'attribut **root** de **Expression**) en utilisant les sous-expressions et le signe de l'expression préalablement récupéré.