

Introduction

L'objectif de ce TP est de vous faire développer (à terme) une petite application JavaFX pour gérer des expressions arithmétiques simples telles que $r = 2 - 3 / (a * b) + 1$.

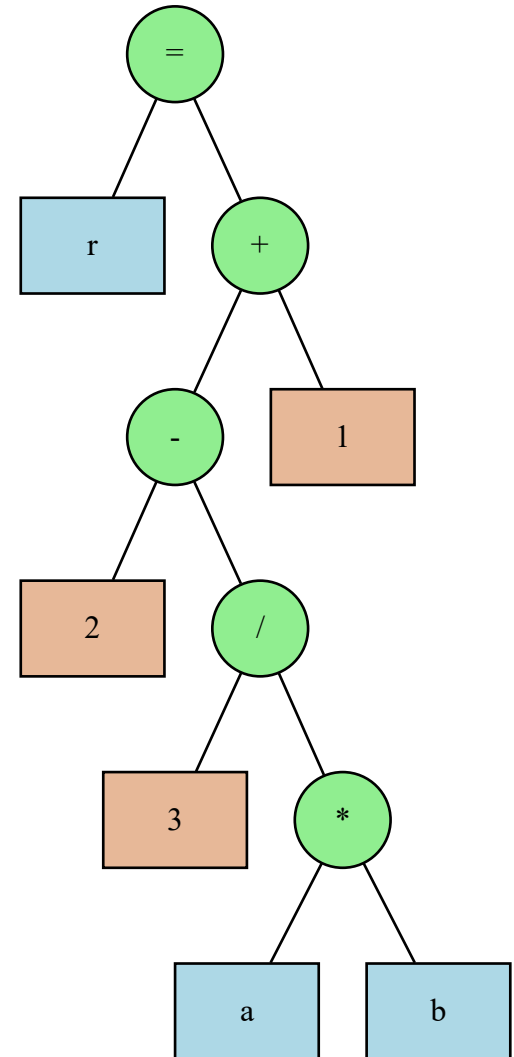
Une telle expression peut se traduire par un arbre (aussi appelé "Binary Expression Tree") :





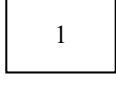

- Chaque nœud représente une opération : =, +, -, *, /
- Les branches gauche et droite représentent les opérandes de ces opérations.
- Les opérandes peuvent être
 - Des expressions dites terminales : r , 1, 2, 3, a , b .
 - Ou bien d'autres expressions : $3 / (a * b)$, etc.

Expressions

Chaque expression possède les caractéristiques suivantes regroupées dans l'interface `Expression<E>`: Le paramètre de type `E` sera ici particulier dans la mesure où il devra descendre de la classe `Number` qui est parente des classes `Integer`, `Float` et `Double`: `Expression<E extends Number>`. La classe `Number` est parente des classes `Integer`, `Float` et `Double` qui représentent les équivalents "objet" des types simple `int`, `float` et `double` respectivement.

- Une expression peut avoir une valeur (ou pas) : Cette notion est représentée par l'interface `ValueHolder<E>`
 - La méthode boolean `hasValue()` indique si l'expression a une valeur
 - La méthode `E value()` throws `IllegalStateException` renvoie cette valeur (de type `E`) si et seulement si celle-ci existe ou bien lève une `IllegalStateException` si la valeur de cette expression ne peut pas être déterminée.
- Une expression peut avoir un parent (une autre expression dans laquelle elle est contenue) ou pas : Cette Notion est représentée par l'interface `ParentHolder<T>`. `Expression<E>` sera donc un `ParentHolder<Expression<E>>`.
 - La méthode `Expression<E> getParent()` permet d'accéder à l'expression parente de cette expression. Celle-ci peut être `null` si l'expression n'a pas de parent.
 - La méthode `setParent(Expression<E> parent)` permet de mettre en place une expression parente de celle-ci.



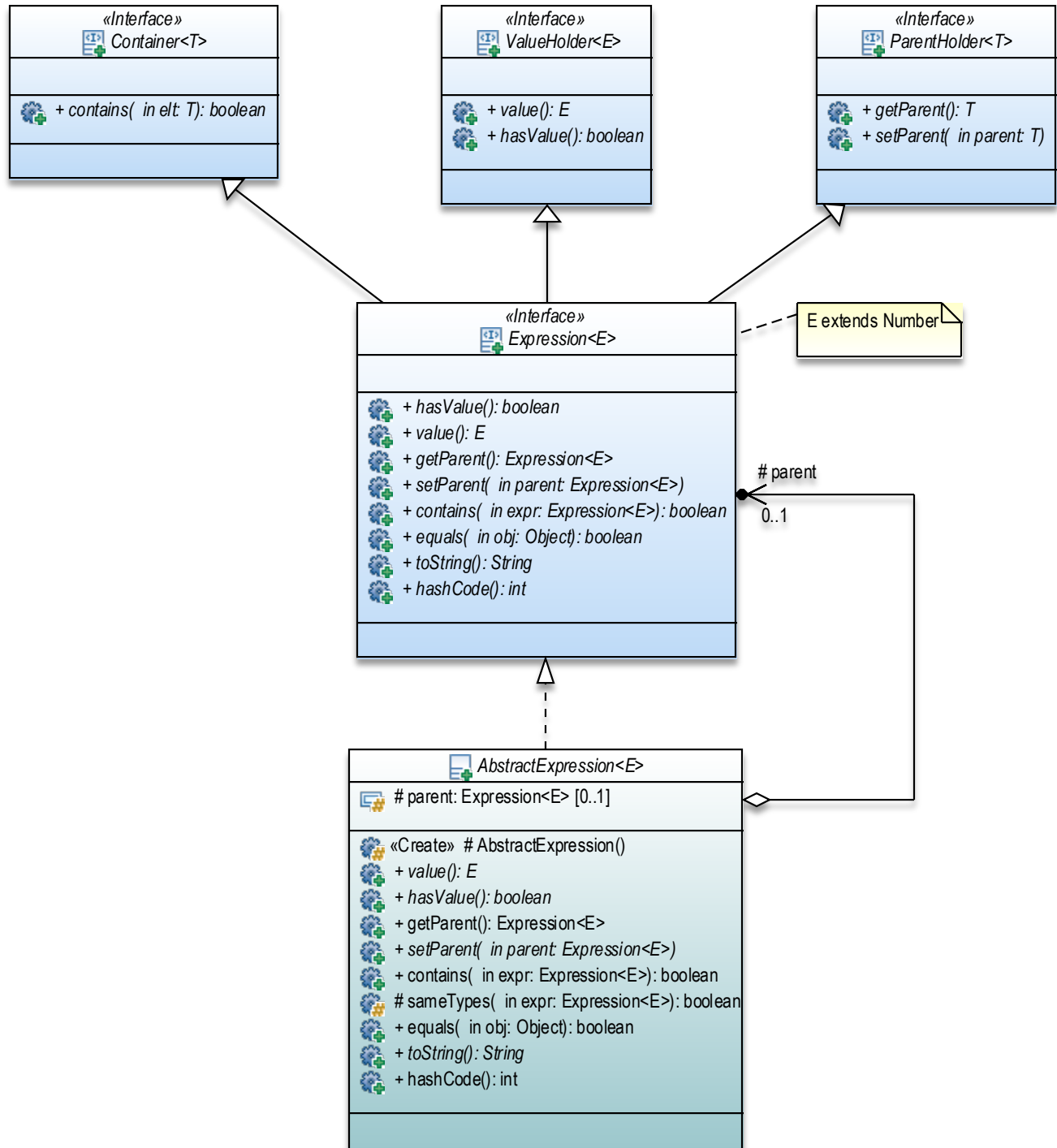
- Typiquement les expressions terminales (constantes et variables) auront pour parentes des expressions non terminales (comme les opérateurs). Les expressions terminales ne pourront pas être des parentes dans un arbre d'expressions.
- En revanche, les expressions non terminales comme les opérateurs binaires arithmétiques (+, -, *, /, ...) pourront être des parentes dans un arbre d'expressions.
- Une expression peut contenir une autre expression : Cette notion est représentée par l'interface `Container<T>`. `Expression<E>` sera donc un `Container<Expression<E>>`.
 - La méthode boolean `contains(Expression<E> expr)` permet d'indiquer si l'expression `expr` est **contenue** dans l'expression.
 - Pour une expression terminale la notion de "contenu" se limitera à l'égalité (de référence) entre l'objet courant (`this`) et l'objet fourni en argument `expr`.
 - Pour les expressions non terminales il faudra ajouter à la notion précédente le fait que l'expression recherchée (`expr`) puisse être l'une des expressions filles, ou bien même contenue à l'intérieur d'une des expressions filles. Exemples :
 - Dans l'arbre de la figure 1, l'expression contenant la valeur  est contenue dans l'expression de l'opérateur  puisqu'elle est contenue dans les opérandes de l'opérateur  qui fait partie lui-même des opérandes de l'opérateur .
 - Alors que l'expression contenant la valeur  n'est pas contenue dans l'expression de l'opérateur .

La Figure 1 page 3 présente l'interface `Expression<E>` qui implémente les 3 notions présentée ci-dessus, ainsi que la classe abstraite `AbstractExpression<E>`.

La classe abstraite `AbstractExpression<E>` contient une implémentation partielle des `Expressions<E>` :

- Elle contient donc une `Expression<E>` `parent` qui doit être initialisée à `null` dans un constructeur.
- Un accès en lecture à l'expression parente. Mais pas encore d'accès en écriture.
- Une implémentation de la méthode boolean `contains(Expression<E> expr)` qui implémente le comportement par défaut, à savoir, renvoie `true` lorsque l'expression en argument est la même (au sens des références) que l'expression courante.
- Une implémentation de la méthode boolean `equals(Object obj)` qui obéira aux règles suivantes:
 - On renverra `false` si `obj` est `null` (habituel).
 - On renverra `true` si `obj` et `this` sont égaux au sens des références (classique).
 - Et enfin, on renverra `true` si
 - `obj` est aussi une expression.
 - Les types (Type d'expression et type de nombres contenus) sont les mêmes. Ce résultat est fourni par la méthode protégée boolean `sameTypes(Expression<? extends Number> expr)`.
 - Les deux expressions s'écrivent de la même manière (au sens de la méthode `String toString()` même si celle-ci est encore abstraite).
- La méthode boolean `sameTypes(Expression<? extends Number> expr)` renvoie :
 - `false`
 - Si l'expression fournie est `null`.
 - Si la classe de l'expression fournie n'est pas la même que la classe de l'expression courante.

- Si les deux expressions ont une valeur et que les classes de ces deux valeurs sont différentes.
- true
 - Dans les autres cas.
- Pour respecter l'implémentation proposée dans la méthode boolean equals(Object obj) l'implémentation de la méthode int hashCode() devra elle aussi se baser sur la méthode String toString().



• Figure 1 : Classes de base des expressions

Expressions terminales

On parle ici d'expressions terminales car elles représentent les feuilles d'un arbre d'expressions.

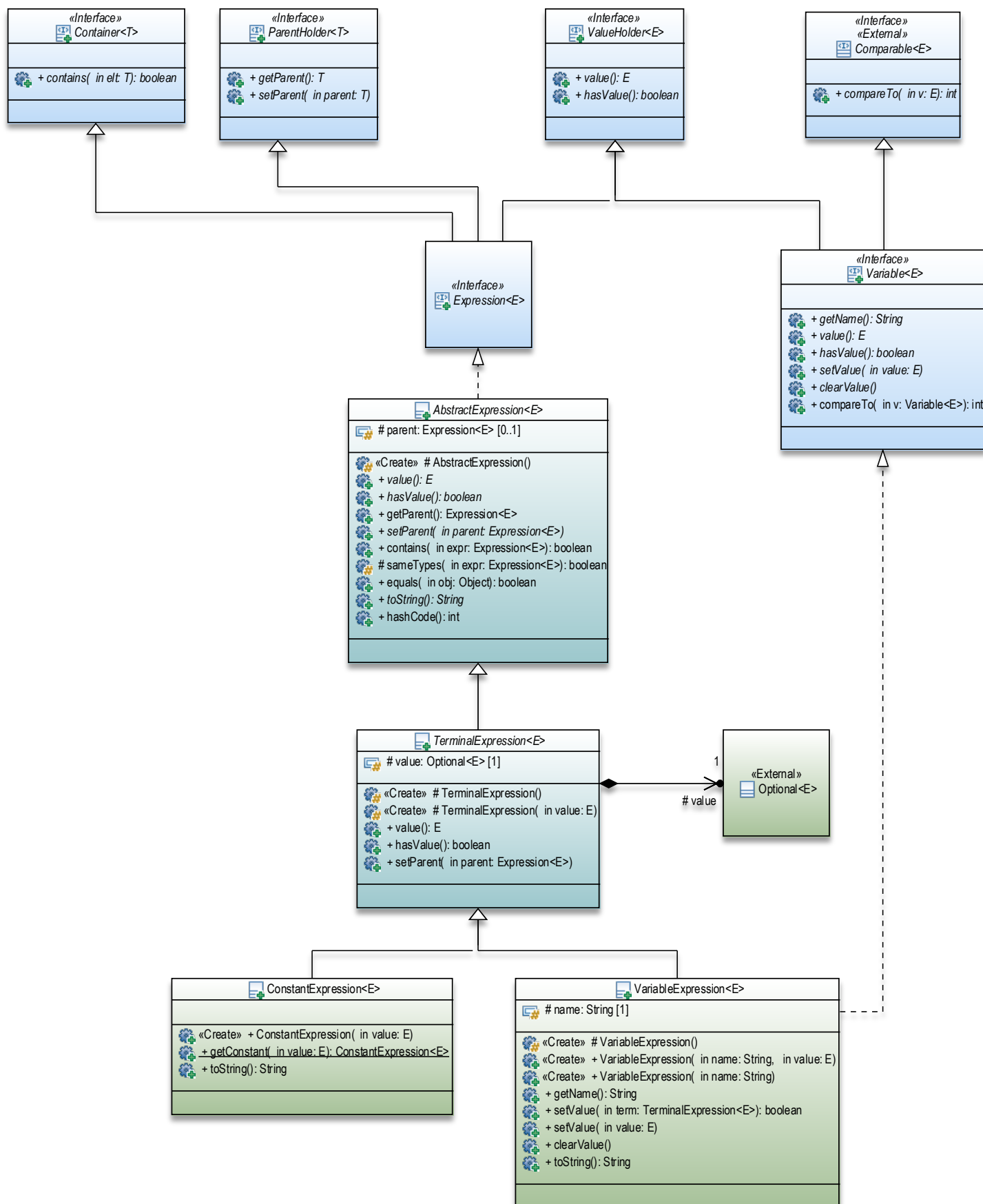


Figure 2 : Expressions terminales

Expression Terminale : TerminalExpression<E>

Les expressions terminales sont celles qui contiennent une valeur. On trouve donc dans la classe TerminalExpression<E> qui hérite de AbstractExpression<E> :

- Une valeur (**optionnelle**) Optional<E> value;
- Lorsqu'une valeur est présente dans value :
 - la méthode hasValue() renvoie true.
 - la méthode value() renvoie la valeur contenue **dans** value.
- A l'inverse, lorsqu'aucune valeur n'est présente dans value:
 - la méthode hasValue() renvoie false.
 - la méthode value() lève une IllegalStateException.
- La méthode setParent(Expression<E> parent) permet de mettre en place un parent qui ne soit pas une TerminalExpression sans quoi une IllegalStateException devra être levée.

Constantes : ConstantExpression<E>

La classe ConstantExpression<E> contient :

- Un constructeur permettant de créer une "constante" à partir d'une valeur numérique. On considérera qu'une constante ne peut *pas* ne pas avoir de valeur, on lèvera donc une NullPointerException si la valeur fournie est null.
- Une factory method <E extends Number> ConstantExpression<E> getConstant(E value) permettant de créer une constante à la valeur souhaitée.
- Une implémentation de la méthode String toString() qui affichera uniquement la valeur de la constante (si elle existe).

Variables : VariableExpression<E>

La classe VariableExpression<E> contient :

- Un attribut contenant le nom de la variable : String name;
- Des constructeurs pour initialiser
 - Le nom de la variable qui peut éventuellement être null.
 - La valeur de la variable: si une valeur nulle est fournie on initialisera value à Optional.empty();

La classe VariableExpression<E> implémente l'interface Variable<E> qui est un raffinement de l'interface ValueHolder<E>. A ce titre elle doit implémenter :

- Un accesseur en lecture pour le nom de la variable : String getName(). On considérera que l'on ne peut pas changer le nom d'une variable au cours de son cycle de vie.
- Un mutateur pour changer la valeur de la variable à partir d'une valeur : setValue(E value) qui lèvera une NullPointerException si la valeur fournie est nulle.
- une méthode clearValue() pour effacer la valeur actuelle et revenir à un état "sans valeur".
- une méthode de comparaison (3way) avec une autre variable int compareTo(Variable<E> v) basée **uniquement** sur le nom de la variable et pas sur la valeur.
- La classe VariableExpression<E> contient aussi :

- Un mutateur pour changer la valeur de la variable à partir d'une autre `TerminalExpression<E>` : `boolean setValue(TerminalExpression<E> term)` qui renverra vrai si la valeur contenue dans `term` (si elle existe) a été transférée dans la variable courante.
- Une implémentation de la méthode `String toString()` qui affichera uniquement le nom de la variable.

Note : Nous verrons lors d'un prochain TP comment "partager" des variables entre plusieurs expressions en créant une sous-classe `SharedVariable<E>`.

Expressions non terminales

On parle ici d'expressions non terminales car elles représentent les nœuds d'un arbre d'expressions

On se limitera dans un premier temps aux expressions représentant les opérations arithmétiques, qui sont donc des opérateurs binaires à deux opérandes : +, -, *, /et ^.

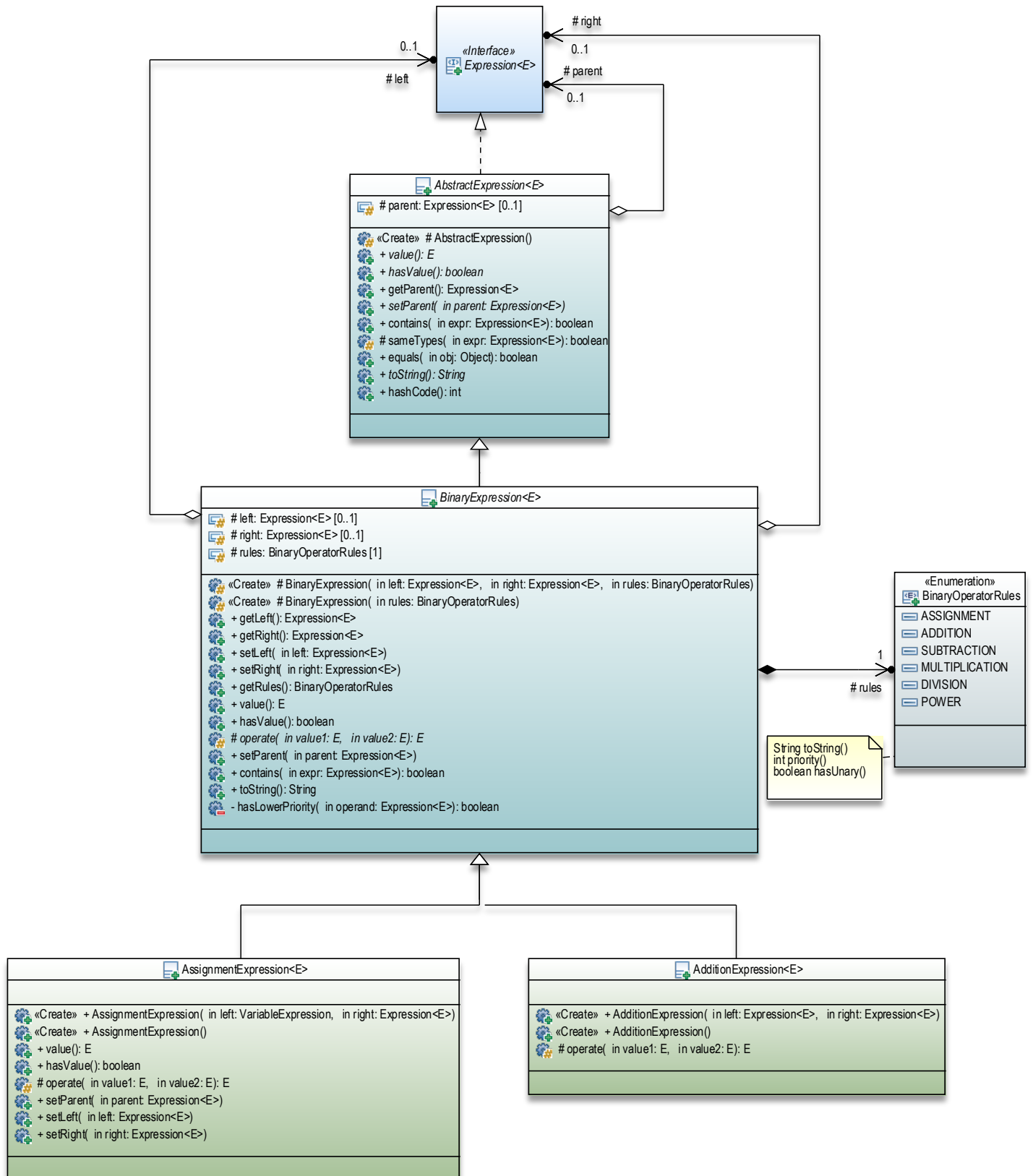


Figure 3 : Expressions binaires

Opérateurs binaires : BinaryExpression<E>

La classe BinaryExpression<E> est la class mère de tous les opérateurs arithmétiques, elle hérite de AbstractExpression<E> et contient :

- L'opérande de gauche : Expression<E> left.
- L'opérande de droite : Expression<E> right.
- Les règles s'appliquant à cet opérateur : BinaryOperatorRules qui est un enum contenant les constantes symboliques pour tous les opérateurs (ASSIGNMENT, ADDITION, MULTIPLICATION, ...) ainsi que des méthodes pour obtenir :
 - Une représentation sous forme de chaîne de caractère de cet opérateur : String toString()
 - La priorité de l'opérateur : int priority(). On considérera que les priorités des opérateurs se répartissent ainsi :
 - L'affectation = a une priorité de 0.
 - L'addition + et la soustraction – ont une priorité de 1.
 - La multiplication * et la division / ont une priorité de 2.
 - La puissance ^ a une priorité de 3 (optionnel).
 - Le fait que l'opérateur autorise (ou pas) les opérations unaires : boolean hasUnary(). On considérera que seules l'addition et la soustraction autorisent les opérations unaires telles que - 2 ou +3 car on pourra toujours les remplacer par leur équivalent binaire : 0 – 2 et 0 + 3 respectivement.
- Des accesseurs en lecture pour les opérandes gauche et droite : Expression<E> getLeft() & Expression<E> getRight()
- Des accesseurs en écriture pour les opérandes gauche et droite : setLeft(Expression<E>) & setRight(Expression<E>).
 - On veillera en mettant en place de nouveaux opérandes à mettre en place l'opérateur courant comme parent de ces opérandes.
- E value() & boolean hasValue() : On considérera qu'une expression binaire a une valeur lorsque chacun de ses deux opérandes possède une valeur. La valeur de l'opérateur (si celle-ci existe) sera le résultat de l'opération interne réalisée par la méthode E operate(E, E).
- La méthode abstraite E operate(E, E) devra être implémentée dans chacune des classes filles pour réaliser l'opération arithmétique spécifique à chaque opérateur. Si on ne peut pas fournir un résultat (si le type de nombre utilisé dans les expressions ne fait pas partie des types Double, Float ou Integer par exemple) on pourra lever une UnsupportedOperationException dans l'implémentation de cette méthode.
- setParent(Expression<E>) : On considérera qu'un parent valide ne peut pas être une expression terminale et qu'une expression ne peut pas être son propre parent.
- boolean contains(Expression<E> expr) : une expression binaire contient l'expression expr si
 - expr est cette expression binaire,
 - expr est une des deux opérandes de cette expression binaire.
- String toString() : La représentation d'une expression binaire consiste en la représentation infix (avec l'opérateur entre les opérandes) sous forme de chaîne de caractère. Elle est donc constituée :
 - De la représentation sous forme de chaîne de caractères de l'opérande de gauche
 - De la représentation sous forme de chaîne de caractères de l'opérateur (voir la méthode toString() de l'enum BinaryOperatorRules)
 - De la représentation sous forme de chaîne de caractères de l'opérande de droite.
 - On notera que si l'opérande de droite contient d'autres opérateurs de priorité inférieure ou égale à la priorité de l'opérateur courant il faudra encadrer l'opérande de droite par des parenthèses.

Opérateurs arithmétiques : Exemple AdditionExpression<E>

La classe AdditionExpression<E> qui hérite de BinaryExpression<E> est une implémentation concrète réalisant les opérations d'addition des opérandes : $a + b$.

Il vous reviendra de développer les autres opérations :

- SubtractionExpression<E> pour réaliser la soustraction des opérandes : $a - b$.
- MultiplicationExpression<E> pour réaliser la multiplication des opérandes : $a \times b$.
- DivisionExpression<E> pour réaliser la division des opérandes : $a \div b$.
- PowerExpression<E> pour réaliser l'exponentiation du premier opérande par le second opérande : a^b .

La classe AdditionExpression<E> contient très peu de choses :

- Un constructeur valué.
- Un constructeur par défaut.
- l'implémentation concrète de l'opération arithmétique : `E operate(E value1, E value2) throws UnsupportedOperationException`.

Opérateur d'affectation : AssignmentExpression<E>

La classe AssignmentExpression<E> qui hérite de BinaryExpression<E> ne représente pas à proprement parler un opérateur arithmétique mais peut être utilisée pour donner une valeur à une variable par exemple : $a = b + 2$. A ce titre l'opération réalisée par cet opérateur consiste à transférer la valeur de l'opérande de droite ($b + 2$) dans l'opérande de gauche (a), si cela est possible.

Pour simplifier le fonctionnement de nos expressions on considérera que :

- L'opérande de gauche d'une affectation doit obligatoirement être une variable.
- Une affectation ne peut pas avoir de parent. Elle peut en revanche être à la racine d'un arbre d'expressions. Auquel cas, elle doit être la seule affectation de cet arbre d'expressions.

La classe AssignmentExpression<E> contient donc :

- Un constructeur valué
- Un constructeur par défaut
- `boolean hasValue()` : renvoie vrai si l'opérande de droite a une valeur.
- `E value()` : transfère la valeur de l'opérande de droite dans la variable de gauche et renvoie cette valeur.
- `E operate(E value1, E value2)` : renvoie l'une ou l'autre des valeurs des opérandes puisqu'elles sont censées être égales.
- `setParent(Expression<E> expr)` : On pourra réutiliser le comportement de la méthode `setParent(...)` de la classe mère auquel il faudra ajouter le fait qu'une BinaryExpression<E> ne peut **pas** être un parent valide pour une affectation : Auquel cas on lèvera une `IllegalArgumentException`. Néanmoins, on n'interdira pas toutes les expressions car nous aurons besoin plus tard de pouvoir grouper plusieurs expressions dans notre futur modèle de données.
- `setLeft(Expression<E> expr)` : veillera au fait que `expr` **doit** être une VariableExpression<E>. On lèvera une `IllegalArgumentException` si ce n'est pas le cas.
- `setLeft(Expression<E> expr)` : veillera au fait que si `expr` est null ou si `expr` ne contient aucune valeur la variable de gauche doit elle aussi ne contenir aucune valeur. En revanche, si `expr` contient une valeur celle-ci doit être copiée dans la variable de gauche.

Parsing

Le parsing d'une expression infixe (avec l'opérateur au milieu, exemple "b + 3") requiert la lecture d'une première opérande "b" puis la lecture de l'opérateur "+" puis la lecture de la deuxième opérande "3".

Le parsing implique donc l'utilisation de deux piles : l'une pour les opérandes, l'autre pour les opérateurs.

- La pile des opérandes stocke les opérandes ou plus généralement les expressions déjà parsées.
- La pile des opérateurs stocke les opérateurs apparus lors de la lecture. On considère que l'expression à parser est une chaîne de caractères que l'on va parcourir un à un.

L'algorithme de parsing d'une chaîne de caractères représentant le contexte à analyser est donc le suivant (en ignorant pour l'instant les cas d'erreur):

Pour chaque caractère c du contexte à interpréter

Si c'est un espace ==> Skip

Si c'est un chiffre ==> Parser une constante et la mettre sur la pile des opérandes

Si c'est une lettre ==> Parser une variable et la mettre sur la pile des opérandes

Si c'est un opérateur ==>

Créer un opérateur (vide pour l'instant)

Tant que la pile des opérateurs contient des opérateurs avec une priorité >=

Dépiler 1 opérateur de la pile des opérateurs

Dépiler 2 opérandes en haut de la pile des opérandes pour les affecter à

l'opérateur

Empiler l'opérateur ainsi composé sur la pile des opérandes

Empiler l'opérateur créé sur la pile des opérateurs

Si c'est une parenthèse ouvrante ==> Parser le sous contexte commençant après la parenthèse (appel récursif)

Tant que la pile des opérateurs n'est pas vide

Dépiler 1 opérateur de la pile des opérateurs

Dépiler 2 opérandes de la pile des opérandes pour les affecter à l'opérateur

Empiler l'opérateur sur la pile des opérandes

lorsque la pile des opérateurs est vide renvoyer le résultat qui est l'élément en haut de la pile des opérandes.

Vous avez sans doute remarqué que toutes les opérations arithmétiques possédaient des constructeurs par défaut permettant de créer les expressions des opérations arithmétiques sans leur affecter d'opérandes.

Vous comprenez maintenant pourquoi : Les opérateurs sont créés puis empilés sur la pile des opérateurs avant que l'on puisse déterminer leurs opérandes.

Travail à réaliser

Vous trouverez un squelette de projet dans l'archive /pub/LAOB12/TP4/TP-Expressions.zip que vous pourrez importer dans votre IDE préféré. Vous aurez éventuellement besoin d'ajouter la librairie JUnit5 à votre projet (et plus tard la librairie JavaFX).

Complétez ou développez les classes requises.

Vous avez à votre disposition les classes de tests du package tests.

Vous avez aussi un exemple de programme principal dans la classe application/Main. Vous pourrez la lancer avec les arguments suivants pour éprouver votre parser : --type float c=1; r = 2 - 3 / (a * b) + 1; a = 5; a + b^3; 1 + (2 - 3 / (a * b)); (c + 2 - 3) / (a * b); b=2

Les prochains TPs

Lors des prochains TP nous allons :

- Construire une application GUI avec JavaFX qui intégrera nos expressions arithmétiques :
 - Parsing
 - Affichage et édition des expressions sous la forme
 - D'arbre
 - De tableau ou de liste
 - Affichage des variables de nos expressions et modifications de leurs valeurs.
- Créer un modèle de données (au sens du MVC) contenant nos expressions et compatible avec les affichages sus mentionnés (Arbre, Tableau, Liste).