eFrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

Structure de données : Rapport sur postfixe

Rédigé et présenté par :
- Cédric Alonso
- Jade Hatoum

Année académique 2023 - 2024

SOMMAIRE

Introduction

Structure Générale : Les expressions mathématiques que nous rencontrons au quotidien sont de nature infixée. Cependant, dans ce rapport nous allons explorer uniquement l'une de ses 2 dérivées : les expressions posfixées.

En effet, il s'agit d'un format suivant la structure d'une pile, permettant à l'ordinateur d'ordonner l'exécution des divers sections de l'expression fournie. Parmi les charactères de l'expression, on reconnaît les opérateurs (+, -, /, *), les chiffres et finalement les parenthèses de toutes natures (*( ), [ ] ,* {}).

Nous avons décidé d'effectuer ce TP en Java. De plus, nous avons fait en sorte que seul les chiffres [0 ; 9] seront pris en compte dans les expressions infixées, afin d'éviter une extraction et interprétation incorrecte des valeurs dans l'expression postfixée finale.

Ainsi, dans ce rapport, nous aborderons d'abord la vérification des parenthèses, puis l'intégration des expressions infixées et postfixées dans une calculatrice à l'aide de notre implémentation de la pile dynamique explicité au TP précédent.

1. Les utilitaires utilisés

```java
package TP3.utils;

19 usages  mrredcoding
public enum BracketType {
    3 usages
    OPEN_PARENTHESIS( symbol: '('),
    3 usages
    CLOSE_PARENTHESIS( symbol: ')'),
    3 usages
    OPEN_SQUARE_BRACKET( symbol: '['),
    3 usages
    CLOSE_SQUARE_BRACKET( symbol: ']'),
    3 usages
    OPEN_CURLY_BRACE( symbol: '{'),
    3 usages
    CLOSE_CURLY_BRACE( symbol: '}');

    2 usages
    private final Character symbol;

    12 usages  mrredcoding
    BracketType(Character symbol) {
        this.symbol = symbol;
    }

    18 usages  mrredcoding
    public Character getSymbol() {
        return symbol;
    }
}
```

```java
package TP3.utils;

8 usages  mrredcoding
public enum OperatorType {

    2 usages
    ADD('+'),
    2 usages
    SUBTRACT('-'),
    2 usages
    MULTIPLY('*'),
    2 usages
    DIVIDE('/');

    2 usages
    private final Character operator;

    8 usages  mrredcoding
    OperatorType(Character operator) {
        this.operator = operator;
    }

    8 usages  mrredcoding
    public Character getOperator() {
        return operator;
    }
}
```

I. Vérification des parenthèses

1. Algorithme explicité

La méthode areBracketsNested est utilisée vérifier si les parenthèses d'une expression infixée sont correctement imbriquées.

L'algorithme commence par créer une pile vide, appelée *stack*, qui servira à stocker temporairement les parenthèses ouvrantes rencontrées. Ensuite, il parcourt chaque caractère de l'expression donnée.

Pendant le parcours, l'algorithme traite chaque caractère comme suit :

- Si le caractère est un espace, il est ignoré.
- Si le caractère est une parenthèse ouvrante *(*, *[* ou {, il est ajouté à la pile.
- Si le caractère est une parenthèse fermante *)*, *]* ou }, l'algorithme vérifie si la pile est vide. Si c'est le cas, cela signifie qu'il n'y a pas de parenthèse ouvrante correspondante, et l'algorithme retourne *false*. Sinon, il dépile la dernière parenthèse ouvrante de la pile et vérifie si elle correspond à la parenthèse fermante actuelle. Si les parenthèses ne correspondent pas, l'algorithme retourne également *false*.

Une fois le parcours de l'expression terminé, l'algorithme vérifie si la pile est vide. Si elle est vide, cela signifie que toutes les parenthèses ouvrantes ont été correctement fermées, et l'algorithme retourne *true*. Sinon, s'il reste des parenthèses ouvrantes sans correspondance, l'algorithme retourne *false*.

L'utilisation d'une pile garantit que les parenthèses sont correctement imbriquées, car chaque parenthèse ouvrante doit être fermée par la parenthèse correspondante dans l'ordre d'apparition.

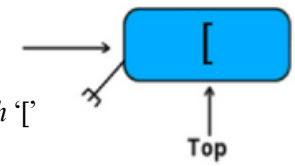
Voici son fonctionnement schématisé :

Un exemple avec : *x * [(a + b) * (c + d)]*

x * [(a + b) * (c + d)]

Dès l' apparition d'un *'open bracket'*, on le *push* dans la pile

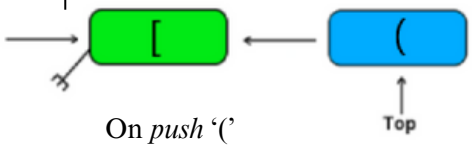
1



On *push* '['

x * [(a + b) * (c + d)]

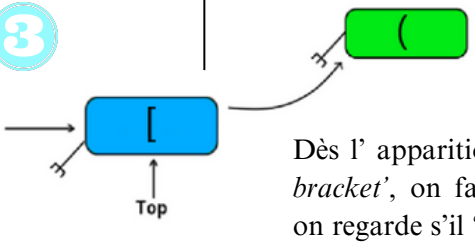
2



On *push* '('

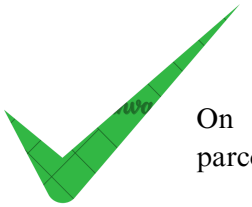
x * [(a + b) * (c + d)]

3



Dès l' apparition d'un *'closed bracket'*, on fait un *pop* puis on regarde s'il 'match' avec le caractère actuel.

Si c'est le cas



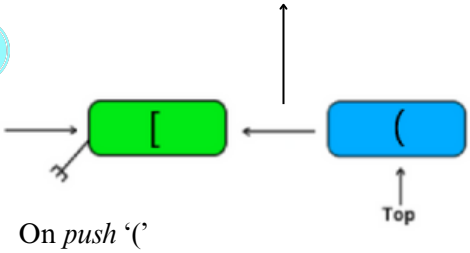
On continue de parcourir l'expression

Si ce n'est pas le cas

On arrête l'algorithme : l'expression est incorrecte

x * [(a + b) * (c + d)]

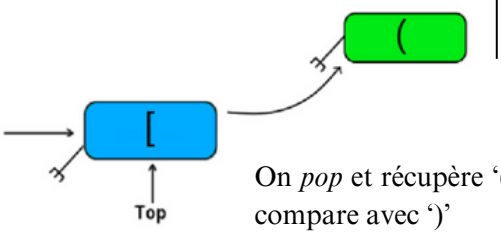
4



On *push* '('

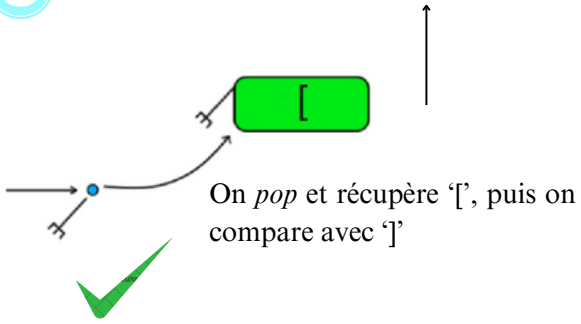
5

x * [(a + b) * (c + d)]



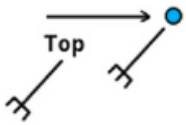
On *pop* et récupère '(', puis on compare avec ')'

6 x * [(a + b) * (c + d)]



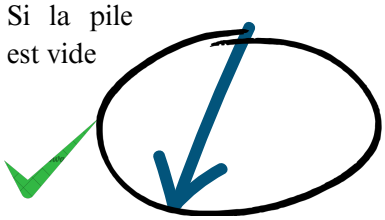
On *pop* et récupère '[', puis on compare avec ']'

7



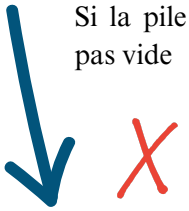
On vérifie l'état de la pile à la fin de l'algorithme

Si la pile est vide



Le nombre de caractères 'ouvrants' et 'fermants' est le même

Si la pile n'est pas vide



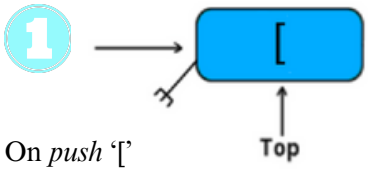
Le nombre de caractères 'ouvrants' et 'fermants' n'est pas le même

Un exemple avec : *[(a - b])*

[(a - b])

Dès l' apparition d'un caractère 'ouvrant', on le *push* dans la pile

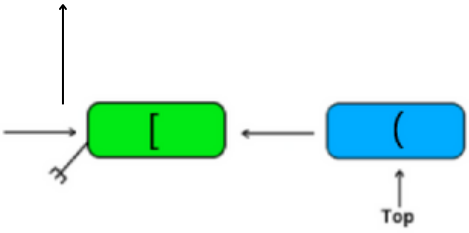
1



On *push* '['

[(a - b])

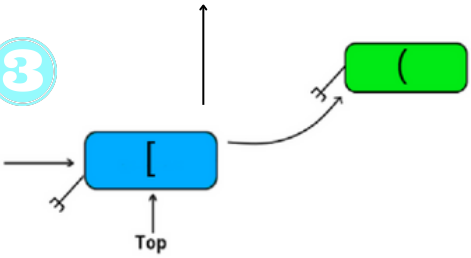
2



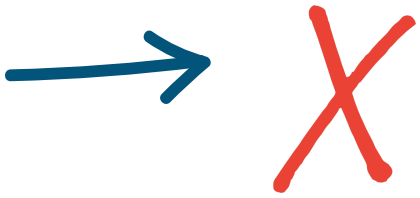
On *push* '('

[(a - b])

3



On *pop* et récupère '(', puis on compare avec ']'



La comparaison échoue

On arrête l'algorithme : l'expression est incorrecte

2. Le code réalisé

```java
package TP3.parenthesesChecker;
import TP2.dynamicStack.DynamicStack;
import TP2.stack.EmptyStackExceptions;
import TP3.utils.BracketType;

/**
 * La classe BracketChecker est utilisée pour vérifier si les parenthèses dans une expression donnée
 * sont correctement imbriquées.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
2 usages   mrredcoding *
public class BracketChecker {

    /**
     * L'expression à vérifier pour les parenthèses correctement imbriquées.
     */
    2 usages
    private final String expression;

    /**
     * Construit un BracketChecker avec l'expression spécifiée.
     *
     * @param expression L'expression à vérifier.
     */
    1 usage   mrredcoding
    public BracketChecker(String expression){
        this.expression = expression;
    }

    /**
     * Vérifie si les parenthèses dans l'expression sont correctement imbriquées.
     *
     * @return true si les parenthèses sont correctement imbriquées, false sinon.
     * @throws EmptyStackExceptions si une tentative est faite pour dépiler un élément d'une pile vide.
     */
    1 usage   mrredcoding *
    public boolean areBracketsNested() throws EmptyStackExceptions {
        DynamicStack<Character> stack = new DynamicStack<>();

        for (Character currentChar : this.expression.toCharArray()) {
            if (currentChar == BracketType.OPEN_PARENTHESIS.getSymbol()
                    || currentChar == BracketType.OPEN_SQUARE_BRACKET.getSymbol()
                    || currentChar == BracketType.OPEN_CURLY_BRACE.getSymbol()) {
                stack.push(currentChar);
            } else if (currentChar == BracketType.CLOSE_PARENTHESIS.getSymbol()
                    || currentChar == BracketType.CLOSE_SQUARE_BRACKET.getSymbol()
                    || currentChar == BracketType.CLOSE_CURLY_BRACE.getSymbol()) {
                if (stack.isEmpty())
                    return false;

                Character poppedChar = stack.pop();

                if ((currentChar == BracketType.CLOSE_PARENTHESIS.getSymbol()
                            && poppedChar != BracketType.OPEN_PARENTHESIS.getSymbol())
                        || (currentChar == BracketType.CLOSE_SQUARE_BRACKET.getSymbol()
                            && poppedChar != BracketType.OPEN_SQUARE_BRACKET.getSymbol())
                        || (currentChar == BracketType.CLOSE_CURLY_BRACE.getSymbol()
                            && poppedChar != BracketType.OPEN_CURLY_BRACE.getSymbol())) {
                    return false;
                }
            }
        }
        return stack.isEmpty();
    }
}
```

3. Les tests et résultats de l'implémentation

```java
package TP3.parenthesesChecker;

import TP2.stack.EmptyStackExceptions;

/**
 * La classe Main contient la méthode principale pour exécuter le programme de vérification des parenthèses
 * en utilisant la classe BracketChecker.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
 mrredcoding
public class Main {

    /**
     * La méthode principale du programme.
     *
     * @param args Les arguments de ligne de commande.
     */
     mrredcoding
    public static void main(String[] args) {
        // Expression contenant des parenthèses à vérifier
        String expression = "7 - { [ x * [ ( x + y) / (j - 3) ] + y ] / (4 - 2.5) }";

        // Vérification des parenthèses dans l'expression
        BracketChecker checker = new BracketChecker(expression);

        try {
            if (checker.areBracketsNested()) {
                System.out.println("Les parenthèses sont correctement imbriquées.");
            } else {
                System.out.println("Les parenthèses ne sont pas correctement imbriquées.");
            }
        } catch (EmptyStackExceptions e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```
Les parentheses sont correctement imbriquees.

Process finished with exit code 0
```

II. Calculatrice

1. Algorithme explicité

Il nous est d'abord demandé de transformer une expression infixée en expression postfixée. Encore une fois, nous supposons que les valeurs ne sont que des chiffres, et non des nombres.

Pour ce fait, nous faisons recours à la méthode *toPostfix( )* dans *InfixToPostfix.java*. Ce dernier nous fournira en sortie l'expression postefixée de l'attribut String *infixExpression*.

D'abord une pile dynamique *operatorStack* est initialisée, pour stocker les opérateurs durant l'itération. Un *StringBuilder postfix* permettera de retenir les caractères pertinents. La valeur *String* de ce dernier déterminera l'expression postfixe de *infixExpression.*

En itérant sur chaque caractère du String *infixExpression,* nous poserons *currentChar* en tant que le caractère évalué actuellement. Pour chaque itération, nous constatons les étapes suivantes :

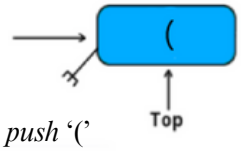
- L'itération actuelle est ignorée dès lors qu'il s'agit d'un espace.
- Nous vérifions d'abord si *currentChar* est une lettre ou un chiffre puis si ce dernier est un *'open bracket'* en faisant recours à la méthode *isOpenBracket( )*), dans quel cas le charactère sera *push* temporairement dans *operatorStack*.
- En outre, s'il s'agit d'un *'closing bracket'* (soit *),]* ou }), les valeurs de *operatorStack* seront *pop* dans *postfix,* tant que *operatorStack* n'est pas vide et tant que la valeur *top* de la pile ne soit pas un *'open bracket'* aussi. Finalement, la valeur de la pile sera rejeté via un *pop* dans le vide.
  Ceci signifie que l'opérateur stocké temporairement dans *operatorStack* sera reintégré dans *postfix,* dès lors qu'un *'closing bracket*' complétant l' *'opening bracket'* précedent, est retrouvé durant l'itération.
- Finalement, si aucune de ces conditions sont vérifiées, les valeurs de *operatorStack* seront *pop* dans *postfix*, tant que *operatorStack* n'est pas vide et tant que la valeur top de la pile soit aussi prioritaire ou plus que *currChar.* Finalement, *currentChar* sera *push* dans *operatorStack,* afin qu'il soit évalué lors de la prochaine itération.

  Ces deux dernières boucles while permettent de respecter les règles de priorité des opérateurs et des parenthèses en les empilant ou désempilant selon leur ordre de priorité.
- Pour terminer, *postfix* récupère l'ensemble des opérateurs dans *operatorStack,* sachant que toutes les manœuvres de priorité ont été gérées au préalable.

Un exemple avec : *(2 + 3) * (6 - 2)*

1

(2 + 3) * (6 - 2)



On *push* '('

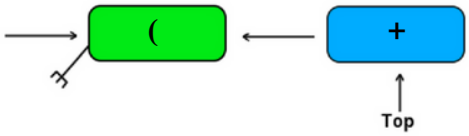
2

(2 + 3) * (6 - 2)

Expression postfixée : '2'

3

(2 + 3) * (6 - 2)



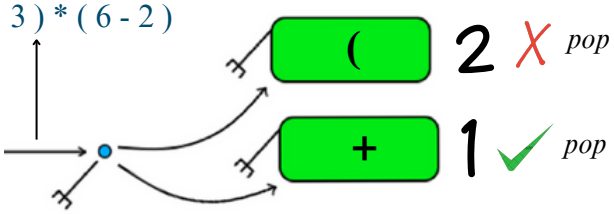
4

(2 + 3) * (6 - 2)

Expression postfixée : '23'

5

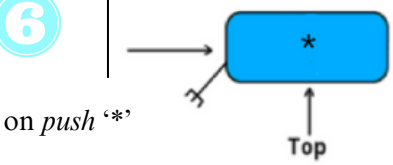
(2 + 3) * (6 - 2)



Expression postfixée : '23+'

6

(2 + 3) * (6 - 2)



on *push* '*'

7

(2 + 3) * (6 - 2)

on *push* '('

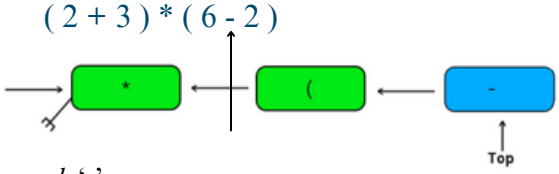
8

(2 + 3) * (6 - 2)

Expression postfixée : '23+6'

9

(2 + 3) * (6 - 2)



on *push* '-'

10

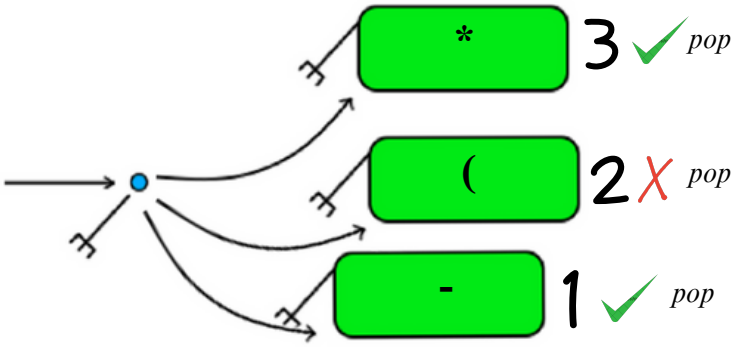
(2 + 3) * (6 - 2)

Expression postfixée : '23+62'

11

(2 + 3) * (6 - 2)

Expression postfixée : '23+62-*'



Il nous est ensuite demandé d'évaluer l'expression postfixée, afin qu'elle puisse être traitée par une calculatrice.

Nous ferons recours à la méthode *evaluate( )* dans *PostfixEvaluator.java* afin d'obtenir le résultat de l'opération en question. En attribut, cette classe possède l'expression postfixée à être évaluée, nommée *postfixExpression.* De plus, nous initialiserons une pile dynamique *operandStack,* qui portera l'ensemble des valeurs *operands* (lettres et chiffres) calculées au fur et à mesure.

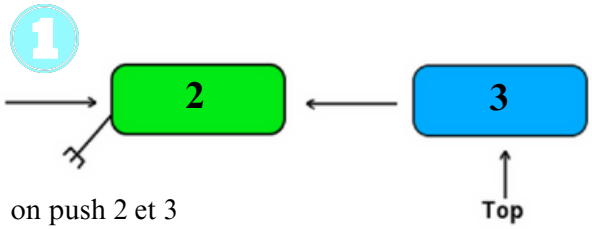
Chaque caractère *currentChar* de ce dernier *String* sera itéré. L'algorithme se déroulera de la manière suivante :

- L'itération actuelle est ignorée dès lors qu'il s'agit d'un espace.
- Si *currentChar* s'agit d'un chiffre, on effectuera un *push* de ce dernier dans *operandStack.* Ceci nous permettera de reaccéder à l'ensemble des valeurs numériques dès lorsqu'on rencontre un opérateur, dans les prochaines itérations.
- Sinon, il s'agit d'un opérateur ...
    - Nous affecterons le retour de 2 *pop* de *operandStack* dans les valeurs *double operand2* et *operand1,* afin de récupérer ces mêmes valeurs numériques mentionnées auparavant.
    - Puis, selon la nature de l'opérateur, nous effectuerons un *push* du résultat de l'opération réspective entre *operand1* et *opreand2*, dans *operandStack.*

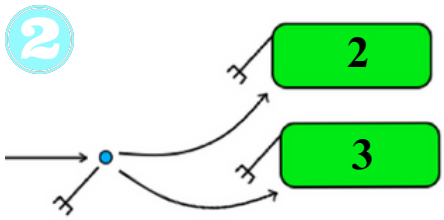
Au moment où l'ensemble des valeurs *double* sont traitées dans *operandStack,* il nous ne restera qu'une seule valeur *double* dans cette dernière pile ; précisemment celle de résultat final à retourner.

Voici son fonctionnement schématisé :

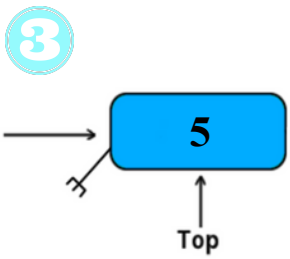
Un exemple avec : *23+62-**



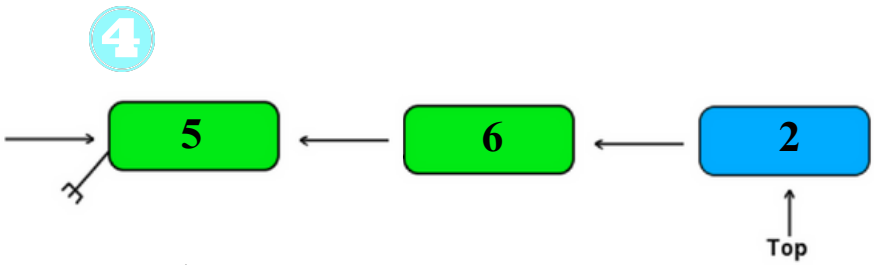
on push 2 et 3



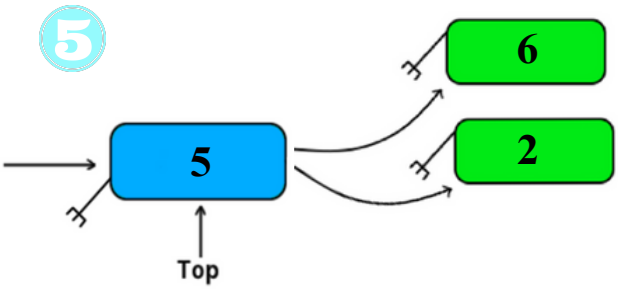
on pop 2 et 3 dans *operand2* et *operand1*



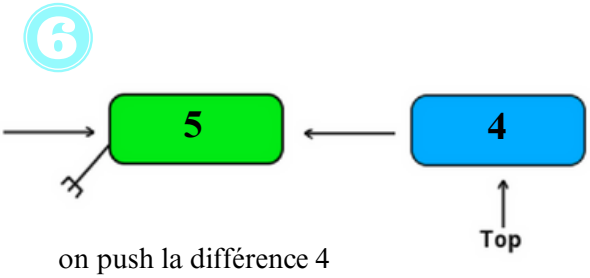
on push la somme 5



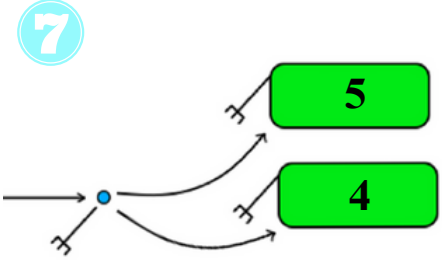
on push 6 et 2



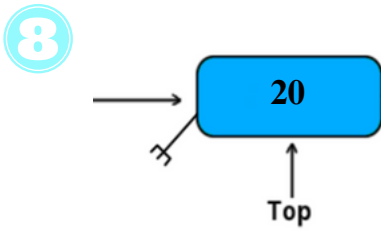
on pop 6 et 2 dans *operand2* et *operand1*



on push la différence 4



on pop 5 et 4 dans *operand2* et *operand1*



on push le produit 20

2. Le code réalisé

```java
package TP3.calculator;

import TP2.dynamicStack.DynamicStack;
import TP2.stack.EmptyStackExceptions;
import TP3.utils.*;

/**
 * La classe InfixToPostfix est utilisée pour convertir une expression infixée en notation postfixée.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
2 usages  ± mrredcoding
public class InfixToPostfix {

    /**
     * L'expression infixée à convertir en notation postfixée.
     */
    2 usages
    private final String infixExpression;

    /**
     * Construit un objet InfixToPostfix avec l'expression infixée spécifiée.
     *
     * @param infixExpression L'expression infixée à convertir.
     */
    1 usage  ± mrredcoding
    public InfixToPostfix(String infixExpression) {
        this.infixExpression = infixExpression;
    }

    /**
     * Retourne la priorité d'un opérateur donné.
     *
     * @param operator L'opérateur dont on veut connaître la priorité.
     * @return La priorité de l'opérateur.
     */
    2 usages  ± mrredcoding
    private int priority(Character operator) {
        if (operator == OperatorType.MULTIPLY.getOperator() || operator == OperatorType.DIVIDE.getOperator())
            return 2;
        else if (operator == OperatorType.ADD.getOperator() || operator == OperatorType.SUBTRACT.getOperator())
            return 1;
        else
            return 0;
    }

    /**
     * Vérifie si un caractère donné est un opérande (chiffre ou lettre).
     *
     * @param character Le caractère à vérifier.
     * @return true si le caractère est un opérande, false sinon.
     */
    1 usage  ± mrredcoding
    private boolean isOperand(Character character) {
        return Character.isLetterOrDigit(character);
    }

    /**
     * Vérifie si un caractère donné est une parenthèse ouvrante.
     *
     * @param character Le caractère à vérifier.
     * @return true si le caractère est une parenthèse ouvrante, false sinon.
     */
    2 usages  ± mrredcoding
    private boolean isOpenBracket(Character character) {
        return character == BracketType.OPEN_PARENTHESIS.getSymbol()
                || character == BracketType.OPEN_SQUARE_BRACKET.getSymbol()
                || character == BracketType.OPEN_CURLY_BRACE.getSymbol();
    }
```

```java
    /**
     * Vérifie si un caractère donné est une parenthèse fermante.
     *
     * @param character Le caractère à vérifier.
     * @return true si le caractère est une parenthèse fermante, false sinon.
     */
    1 usage  mrredcoding
    private boolean isCloseBracket(Character character) {
        return character == BracketType.CLOSE_PARENTHESIS.getSymbol()
                || character == BracketType.CLOSE_SQUARE_BRACKET.getSymbol()
                || character == BracketType.CLOSE_CURLY_BRACE.getSymbol();
    }

    /**
     * Convertit l'expression infixée en notation postfixée.
     *
     * @return L'expression en notation postfixée.
     * @throws EmptyStackExceptions si une tentative est faite pour dépiler un élément d'une pile vide.
     */
    1 usage  mrredcoding
    public String toPostfix() throws EmptyStackExceptions {
        StringBuilder postfix = new StringBuilder();
        DynamicStack<Character> operatorStack = new DynamicStack<>();

        for (char currentChar : infixExpression.toCharArray()) {
            if (currentChar == ' ')
                continue;

            if (isOperand(currentChar)) {
                postfix.append(currentChar);
            } else if (isOpenBracket(currentChar)) {
                operatorStack.push(currentChar);
            } else if (isCloseBracket(currentChar)) {
                while (!operatorStack.isEmpty() && !isOpenBracket(operatorStack.top())) {
                    postfix.append(operatorStack.pop());
                }
                operatorStack.pop();
            } else {
                while (!operatorStack.isEmpty() && priority(currentChar) <= priority(operatorStack.top())) {
                    postfix.append(operatorStack.pop());
                }
                operatorStack.push(currentChar);
            }
        }

        while (!operatorStack.isEmpty()) {
            postfix.append(operatorStack.pop());
        }

        return postfix.toString();
    }
}
```

```java
package TP3.calculator;

import TP2.dynamicStack.DynamicStack;
import TP2.stack.EmptyStackExceptions;
import TP3.utils.*;

/**
 * La classe PostfixEvaluator est utilisée pour évaluer une expression en notation postfixée.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
2 usages  ± mrredcoding
public class PostfixEvaluator {

    /**
     * L'expression en notation postfixée à évaluer.
     */
    2 usages
    private final String postfixExpression;

    /**
     * Construit un objet PostfixEvaluator avec l'expression postfixée spécifiée.
     *
     * @param postfixExpression L'expression en notation postfixée à évaluer.
     */
    1 usage  ± mrredcoding
    public PostfixEvaluator(String postfixExpression){
        this.postfixExpression = postfixExpression;
    }

    /**
     * Évalue l'expression en notation postfixée et retourne le résultat.
     *
     * @return Le résultat de l'évaluation de l'expression postfixée.
     * @throws EmptyStackExceptions si une tentative est faite pour dépiler un élément d'une pile vide.
     */
    public double evaluate() throws EmptyStackExceptions {
        DynamicStack<Double> operandStack = new DynamicStack<>();

        for (char currentChar : postfixExpression.toCharArray()) {
            if (currentChar == ' ')
                continue;

            if (Character.isDigit(currentChar)) {
                operandStack.push((double) Character.getNumericValue(currentChar));
            } else {
                double operand2 = operandStack.pop();
                double operand1 = operandStack.pop();

                if (currentChar == OperatorType.ADD.getOperator())
                    operandStack.push( value: operand1 + operand2);

                else if (currentChar == OperatorType.SUBTRACT.getOperator())
                    operandStack.push( value: operand1 - operand2);

                else if (currentChar == OperatorType.MULTIPLY.getOperator())
                    operandStack.push( value: operand1 * operand2);

                else if (currentChar == OperatorType.DIVIDE.getOperator())
                    operandStack.push( value: operand1 / operand2);
            }
        }

        return operandStack.pop();
    }
}
```

```java
package TP3.calculator;

import TP2.stack.EmptyStackExceptions;
import TP3.parenthesesChecker.BracketChecker;

/**
 * La classe {@code Calculator} fournit des méthodes pour évaluer des expressions mathématiques.
 * Elle utilise d'autres classes telles que {@link TP3.parenthesesChecker.BracketChecker},
 * {@link TP3.calculator.InfixToPostfix}, et {@link TP3.calculator.PostfixEvaluator}
 * pour effectuer les étapes nécessaires dans le processus d'évaluation.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
1 usage  new *
public class Calculator {

    /**
     * Évalue une expression mathématique et renvoie le résultat.
     * Cette méthode effectue les étapes suivantes :
     * 1. Vérifie si les parenthèses dans l'expression sont correctement imbriquées à l'aide
     * de {@link TP3.parenthesesChecker.BracketChecker}.
     * 2. Convertit l'expression infixée en postfixée à l'aide de {@link TP3.calculator.InfixToPostfix}.
     * 3. Évalue l'expression postfixée à l'aide de {@link TP3.calculator.PostfixEvaluator}.
     *
     * @param expression L'expression mathématique à évaluer.
     * @return Le résultat de l'évaluation de l'expression.
     */
    1 usage  new *
    public static double calculate(String expression) {
        BracketChecker bracketChecker = new BracketChecker(expression);

        try {
            // Étape 1 : Vérifier l'imbrication des parenthèses
            if (!bracketChecker.areBracketsNested())
                throw new EmptyStackExceptions("L'expression : " + expression + " n'est pas correcte.");
        } catch (EmptyStackExceptions e) {
            System.out.println(e.getMessage());
        }

        InfixToPostfix infixToPostfix = new InfixToPostfix(expression);

        String postfixExpression = null;
        try {
            // Étape 2 : Convertir l'infixe en postfixe
            postfixExpression = infixToPostfix.toPostfix();
        } catch (EmptyStackExceptions e) {
            System.out.println(e.getMessage());
        }

        PostfixEvaluator postfixEvaluator = new PostfixEvaluator(postfixExpression);

        try {
            // Étape 3 : Évaluer l'expression postfixe
            return postfixEvaluator.evaluate();
        } catch (EmptyStackExceptions e) {
            System.out.println(e.getMessage());
        }

        return 0.0;
    }
}
```

3. Les tests et résultats de l'implémentation

```java
package TP3.calculator;

import TP2.stack.EmptyStackExceptions;

/**
 * La classe Main contient la méthode principale pour exécuter le programme de calculatrice en utilisant les classes
 * InfixToPostfix et PostfixEvaluator.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
mrredcoding
public class Main {

    /**
     * La méthode principale du programme.
     *
     * @param args Les arguments de ligne de commande.
     */
    mrredcoding
    public static void main(String[] args) {
        // Expression infixée à évaluer
        String infixExpression = "( 2 + 3 ) * ( 6 - ( 7 - 8 ) ) / ( 6 + 1 )";
        InfixToPostfix infixToPostfix = new InfixToPostfix(infixExpression);

        // Conversion de l'expression infixée en notation postfixée
        String postfixExpression = null;
        try {
            postfixExpression = infixToPostfix.toPostfix();
        } catch (EmptyStackExceptions e) {
            System.out.println(e.getMessage());
        }

        // Affichage des expressions
        System.out.println("Expression infixée : " + infixExpression);
        System.out.println("Expression postfixée : " + postfixExpression);

        // Évaluation de l'expression postfixée
        PostfixEvaluator postfixEvaluator = new PostfixEvaluator(postfixExpression);

        double result = 0;
        try {
            result = postfixEvaluator.evaluate();
        } catch (EmptyStackExceptions e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Résultat: " + result);

        System.out.println("Résultat calculé avec le calculator : " + Calculator.calculate(infixExpression));
    }
}
```

```
Expression infixée : ( 2 + 3 ) * ( 6 - ( 7 - 8 ) ) / ( 6 + 1 )
Expression postfixée : 23+678--*61+/
Résultat: 5.0
Résultat calculé avec le calculator : 5.0

Process finished with exit code 0
```

efrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

Année académique 2023 - 2024