**Comprehensive Practice**

**Section Two: Expression Evaluation**

**Group ID: 1**

**Group Leader: Imraul Kayes Emmaka (2017239060027)**

**Group Members:**

**Mustansir Nawaz (2017239060007)**

**Mohammad Kemal Lale (2017239060006)**

**Mugisha Seif (2017239060021)**

**SM Shamim Al Mamun (2017239060018)**

**Didarul Islam (2017239060032)**

**Major: Computer Science**

**Supervisor: Shan He**

**School of Computer Science**

**Southwest Petroleum University**

**Problem Statement** :

Write a program that evaluates fully parenthesized arithmetic expressions using the standard operators for addition, subtraction, multiplication, and division, and a special operator for exponentiation ("^").

e.g.: (18.4-((2.3\*8.5)/(19.5+(2.7^4.9))))

**Our Approach to The Problem:**

If we decompose the problem, we find that there’s two critical steps to be taken care of in this problem.

First, we look at the steps in general to solve the problem.

1. Take an input (mathematical expression)
2. Tokenize the input- recognize which characters together makes up one token
3. Evaluate the expression – determine which part/sub-expression to be evaluated first.

From above we can see, the two critical parts are tokenization and evaluation.

For tokenization, we create a supporting class called StringSplitter. To handle the evaluation part we use famous Shunting-Yard algorithm by Edsger Dijkstra.

**Psrocedure in Brief OR main() Method from Client Program:**

Take input using Scanner and put it into a String variable

Take an instance of StringSplitter to tokenize the input

//create to Stacks in order to implement the Shunting-Yard algorithm.

//number stack for num values, symbmol stack for symbols

NumberStack = new Stack(), symbolStack =new Stack();

 Evaluate the expression

 If no error encountered, print result

 Else print Illegal Expression

**StringSplitter Class:**

* Fields:
  + Private Queue<Character> characters;
  + Private String token;
  + Constant String SPECIAL\_CHARS=”()+-\*/^”;
* Constructor:
  + Parameter: String line
  + Initialize character
  + Put each char of line into characters queue
* Methods:
  + hasNext()
    - if there’s no more token returns false, otherwise true
* next()
  + Returns the next token
* findNextToken()

**FindNextToken() or Tokenization:**

* Few Ground Rules:
  + No white space
  + If it belongs to any of our special characters, it’s a token
  + Numbers together – one token

**Pseudocode – Tokenization:**

* Peek if the first elem of the queue is a white space
  + If so - remove()

//the queue could be empty at this point

* If the queue is empty : token = null
* Else
  + Ch = first element of the queue

//we’ll set token as ch, but before that we need check if ch is legal according to our problem statement

* + If ch is a digit or a dot or a SPECIAL\_CHARACTER
    - Initialize token as ch;
  + Else
    - print – ch is not legal!!
    - throw IllegalArgumentException
  + If (token is not a SPECIAL\_CHAR)//then it’s a num
    - Boolean done = false; //we’re done one done when we took all the contiguous digits as a token
    - while(queue is not empty and not done)
      * Ch2 = queue.peek();
      * If ch2 is a whitespace or special\_char: done = true
      * Else if ch2 is a digit or a dot: token = token+queue.remove()
      * Else
        + print - ch2 is not legal!!
        + throw IllegalArumentException
* return token;

**Evaluation() from client class:**

**Overview of the method:**

* Does two things:
  + Evaluates the expression and returns false if no error encountered
  + Returns true if any error encountered
* Formal parameters:
  + StringSplitter splitter, Stack numStack, Stack symbolStack
* Returns boolean err\_flag

**Pseudo-code of Evaluate() Method:**

* Boolean err\_flag = false;
* While no error and there’s more token:
  + Str = next token
  + If str isNumereic: s=numStack.push(str);
* Else
  + If str is “(” or an operator: symbolStack.push(str)
  + Else if str is “)”: //go on and evaluate the sub-expr
    - Num2=numStack.pop(), num1 =numStack.pop();
    - Operator = symbolStack,pop();
    - if operator is not any of “+-\*/”:
      * print – operator(var)+”is in the place of an operator!!”
      * print – Error in parenthesis \n Check ur Input!!
      * return true;
    - Result = calculate(num1, num2, operator)
    - NumStack.push(result)
    - If symbolStack not empty: symbolStack.pop()
* Else err\_flag = true;
* Return err\_flag

**Other Methods in Client Class**

* IsNumeric(String str):
  + Return str.matches("-?\\d+(\\.\\d+)?");
* Calculate(double num1, double num2, String operator):
  + Result = num1 “operator” num2;//for details – plz refer to the source code

**Source Code:**

**StringSplitter Class:**

**import** java.util.LinkedList;

**import** java.util.Queue;

/\*\*

\* StringSplitter class is a supporting class which helps us to tokenize a

\* String the way we want.

\*

\* **@author** emmaka

\* **@since**

\*/

**public** **class** StringSplitter {

// ----------------Fields-----------------------------

**private** Queue<Character> characters;

**private** String token;

**public** **static** **final** String ***SPECIAL\_CHARACTERS*** = "()+-\*/^";

// -------------------Constructor--------------------------

/\*\*

\* This is the constrcor of StringSplitter class. It takes a {@code String} from

\* the class that uses this class and tokenizes the {@code String}. First It

\* puts each {@code char} of the String into a Queue.

\*

\* **@param** line A {@code String}

\*/

**public** StringSplitter(String line) {

characters = **new** LinkedList<Character>();

// put all each char into a queue

**for** (**int** i = 0; i < line.length(); i++) {

characters.add(line.charAt(i));

}

}

/\*\*

\* Checks if there's any more token i.e. if the Queue is empty.

\*

\* **@return** If no more token, returns false. Otherwise true.

\*/

**public** **boolean** hasNext() {

**if** (characters.isEmpty()) {

**return** **false**;

}

**return** **true**;

}

/\*\*

\* This method returns the next token.

\*

\* **@return** findNextToken() A method.

\*/

**public** String next() {

**return** findNextToken();

}

/\*\*

\* This method creates the next token.

\*

\* **@return** token A {@code String}

\*/

**private** String findNextToken() {

// peek ahead and see if the next elem is a white space

// If it's, remove it.

**while** (!characters.isEmpty() && Character.*isWhitespace*(characters.peek())) {

characters.remove();

}

**if** (characters.isEmpty()) {

token = **null**;

} **else** {

**char** ch = characters.remove();

// if ch is either a digit or a dot or a SPECIAL\_CHARACTER

// initialize token as ch

**if** (Character.*isDigit*(ch) || ch == '.' || ***SPECIAL\_CHARACTERS***.indexOf(ch) >= 0)

token = "" + ch;

// else throw IllegalArgumentException

**else** {

System.***out***.println(ch + " is not legal!!");

**throw** **new** IllegalArgumentException("Invalid Input!!");

}

**if** (!***SPECIAL\_CHARACTERS***.contains(token)) {

**boolean** done = **false**;

**while** (!characters.isEmpty() && !done) {

**char** ch2 = characters.peek();

**if** (Character.*isWhitespace*(ch2) || ***SPECIAL\_CHARACTERS***.indexOf(ch2) >= 0) {

done = **true**;

} **else** **if** (Character.*isDigit*(ch2) || ch2 == '.') {

token = token + characters.remove();

} **else** {

System.***out***.println(ch2 + " is not legal!!\n");

// done = true;

**throw** **new** IllegalArgumentException("Invalid Input!!");

}

}

}

}

**return** token;

}

}

**Client Class:**

**import** java.util.Scanner;

**import** java.util.Stack;

/\*\*

\* This program takes in a mathematical expression as input from the keyboard.

\* It uses StringSplitter class to tokenize the input. Then evaluates it. For

\* this program to evaluate the expression, the expression needs to be fully

\* parenthesized. The mathematical operations that it can calculate are

\* +,-,\*,/,^ . With invalid input, it might throw

\* {@code IllegalArgumentexception} or simply print invalid expression. This

\* program implements a variation of the Shunting-Yard algorithm by Edsgar

\* Dijkstra.

\*

\* **@author** emmaka

\* **@since** 8-July-2019

\*

\*/

**public** **class** Task2 {

// -----------main method------------

**public** **static** **void** main(String[] args) {

@SuppressWarnings("resource")

// take the input from keyboard using scanner

// put it into a Strng var.

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Enter Your Input:");

String input = sc.nextLine();

// declare a StringSpliter varible to tokenize the input

StringSplitter splitter = **new** StringSplitter(input);

// declare two stacks- numStack to put the numeric tokens

// and symbolstack to put the symbolic tokens.

Stack<Double> numStack = **new** Stack<Double>();

Stack<String> symbolStack = **new** Stack<String>();

// declare a boolean var error

// the method evaluates the tokens

// if it encounters any error it returns true, else false.

**boolean** error = *evaluate*(splitter, numStack, symbolStack);

**if** (error || numStack.size() != 1 || !symbolStack.isEmpty()) {

System.***out***.println("Illegal Expression!!");

} **else** {

System.***out***.println("Result = " + numStack.pop());

}

// System.out.println(numStack);

// System.out.println(symbolStack);

}

**public** **static** **boolean** isNumeric(String strNum) {

**return** strNum.matches("-?\\d+(\\.\\d+)?");

}

/\*\*

\* This method implements the Shunting-yard algorithm We keep taking tokens from

\* the splitter as long as there's token to process. We use two stacks- numStack

\* to store the numeric tokens and symboleStack to store symbolic tokens. When

\* we find a right parenthesis(')'), that means we have all the information for

\* that sub-expression, thus we evaluate it.

\*

\* We assume the input has no error. We use a boolean err\_flag to recognize it.

\* If an error if an error is encountered, we set it true.

\*

\* **@param** splitter A {@code StringSplitter} to get tokens.

\* **@param** numStack A {@code Stack} to store the numeric tokens.

\* **@param** symbolStack A {@code Stack} to store the symbols.

\* **@return** err\_flag A {@code boolean}, true if there's an error.

\*/

**static** **boolean** evaluate(StringSplitter splitter, Stack<Double> numStack, Stack<String> symbolStack) {

// a boolean flag to recognize any error

**boolean** err\_flag = **false**;

// As long as there's token and no error:

**while** (!err\_flag && splitter.hasNext()) {

// take the next token into a String var str.

String str = splitter.next();

// System.out.println("TOken: " + str);

/\*\*

\* Check if it's numeric if so,turn it into a double and put it into numStack,

\* else check if it's left parenthesis or any of the legal operator else if it's

\* a right parenthesis, which means we can evaluate this sub-expression.

\*/

**if** (*isNumeric*(str)) {

numStack.push(Double.*parseDouble*(str));

} **else** {

**if** (str.equals("(") || "+-\*/^".contains(str)) {

symbolStack.push(str);

} **else** **if** (str.equals(")")) {

// To evaluate this sub-expression, we pop last two numbers

// pass it as formal parameters of the calculate method

**double** num2 = numStack.pop();

**double** num1 = numStack.pop();

String operator = symbolStack.pop();

**if** (!"+-\*/^".contains(operator)) {

System.***out***.println("'" + operator + "' in the " + "place of an operator!!");

System.***out***.println("Error in parenthesis!" + "!\nCheck Your Input!!");

**return** **true**;

}

// take the calculated value from the last sub-expr into

// a double var result.

**double** result = *calculate*(num1, num2, operator);

// push result into numStack

numStack.push(result);

// if symbolStack not empty

// pop from symbolStack

**if** (!symbolStack.isEmpty())

symbolStack.pop();

} **else** {

System.***out***.println("The number '" + str + "' does not seem right!!");

err\_flag = **true**;

**break**;

}

}

// System.out.println("Num Stack: " + numStack);

// System.out.println("Symbol Stack: " + symbolStack);

// System.out.println("err\_flag = " + err\_flag);

}

**return** err\_flag;

}

/\*\*

\* This method carries out some certain mathematical operation for given

\* numerics.

\*

\* **@param** num1 A {@code double}

\* **@param** num2 A {@code double}

\* **@param** operator A {@code String}

\* **@param** err\_flag A {@code boolean}

\* **@return** result A {@code double}

\*/

**static** **double** calculate(**double** num1, **double** num2, String operator) {

**double** result = 0;

**if** (operator.equals("+")) {

result = num1 + num2;

} **else** **if** (operator.equals("-")) {

result = num1 - num2;

} **else** **if** (operator.equals("\*")) {

result = num1 \* num2;

} **else** **if** (operator.equals("^")) {

result = Math.*pow*(num1, num2);

} **else** **if** (operator.equals("/")) {

result = num1 / num2;

}

**return** result;

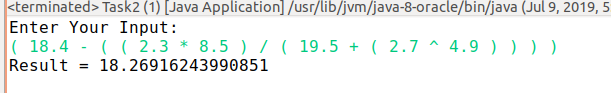
}

}

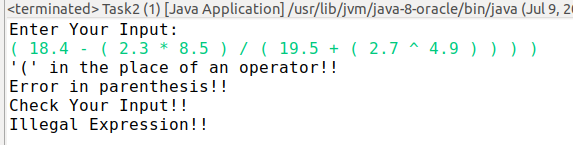
**Sample Input and Output:**

**Sample Input 1: ( 18.4 - ( ( 2.3 \* 8.5 ) / ( 19.5 + ( 2.7 ^ 4.9 ) ) ) )**

**Output:**

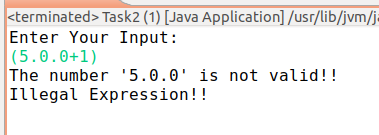
**Sample Input 2: ( 18.4 - ( 2.3 \* 8.5 ) / ( 19.5 + ( 2.7 ^ 4.9 ) ) ) ) //wothout the second left parenthesis**

**Output:**



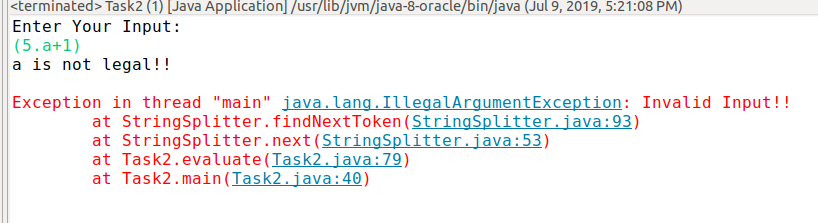
**Sample Input 3: (5.0.0+1)**

**Output:**



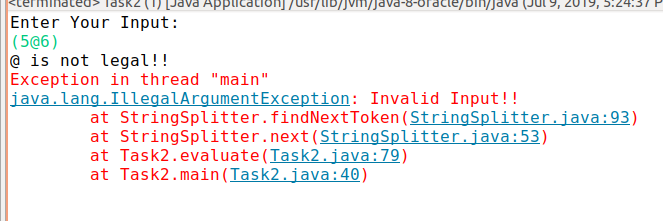
**Sample Input 4: (5.a+1)**

**Output:**



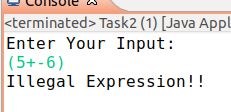
**Sample Input 5: (**[**5@6**](mailto:5@6)**)**

**Output:**



**Sample Input 6: (5+-6)**

**Output:**



**Summary:** While doing this task, the first problem I would say I faced was understanding the problem. Then understanding how the StringSplitter class works or its purpose. Because If it were splitting up the input, we can also do that using String.split() method.

Another part was difficult to understand that why can’t we use delimiter in this case. Also as Java has a pretty rich library , there’s a StringTokenizer class in Java.

Figuring out the right way was pretty insightful in terms of programming and problem solving.