Assignment 2 - CT2109 Object Oriented Programming: Data Structures and Algorithms

Daniel Hannon (19484286)

March 2021

1 Problem Analysis

1.1 Overview

Abstract: In this assignment, you will write a program which reads in a numerical infix expression from the user and convert it to a postfix expression. Once converted, you will then solve the postfix expression and print the result for the user.

In order to complete this assignment I must use *ArrayStack.java* and *Stack.java* which were provided as part of the brief
The *ArrayStack* class is an implementation of a Stack in Java and as a result has four main methods to use:

- *ArrayStack.**top()***: - Checks the top item in the Stack without taking it out of the Stack.
- *ArrayStack.**push(Object n)*** - Inserts *Object n* at the top of the Stack.
- *ArrayStack.**pop()*** - Removes the Object from the top of the stack.
- *ArrayStack.**isEmpty()*** - Checks if the Stack is Empty

Then after this there are two main challenges

- Parsing the input to postfix
- Calculating the result

1.2 Parsing the input to postfix

In order to take a stack based approach to a calculator it is easiest to calculate it using postfix but people do not particularly like using postfix so we have to convert it first.
In order to do this we need to verify input on a character by character basis The first check performed is if the character is a digit or a full stop, if this requirement is met, the string is iterated until it reaches a non numeric value, this index is stored in the inital iterator for the string, then a substring is invoked and the number is added to a postfix string which is then separated by a space (This will be explained later on.)
If that is not satisfied then a quick check is performed to see if the value is a valid operator or a whitespace character. If it is not, the calculation is terminated, a warning is provided and you are returned to the input section. But if it is satisfied it then checks the precidence for the operator and then orders them as such. This is done by using a temporary stack to store all operators which are less than or equally significant than the operator currently present until it reaches the bottom of the stack or an operator of greater significance, then it is inserted in the stack and the temporary stack is transferred back into the main stack. Unlike invalid characters, the program simply ignores mismatched brackets and assumes everything to the right of a mismatched opening bracket to be encapsulated by said bracket, where mismatched end brackets are simply ignored.
Then when the input has been handled in its entirety, the contents of the stack are emptied on to the end of the output string each separated by a space.

1.3 Calculating the value from the postfix

The postfix made from the sum is then passed into the *evaluate(**String** s)* Method.
The reason the spaces were present previously was that this was going to use the **StingTokenizer** class to iterate through the postfix string and deal with everything on a token by token basis. a token is determined wheter or not it is potentially an operator by checking the value at position 0 of the token string. if it is between the ascii values for 0änd 9ör it's equal to .then it's treated as a number and passed to the **Double.*parseDouble(String s)*** method. This is of course in a try catch to deal with invalid numbers and instantly ends the calculation if it's not a valid number. whereas if it's an operator it is passed through a switch statement where the appropriate calculations are performed. This is then returned and presented to the user. This is ultimately followed by the option to start again.

1.4 Results and Known errors

Overall it is working mostly, there is an issue involving equaltions in the form

$$a * (b)^c - d$$

it works relatively well otherwise.

2 Code

```
import javax.swing.JOptionPane;
import java.util.StringTokenizer;

public class Main {
  private ArrayStack stack;
  private String input;
  private String output;
  private boolean looping;
  private boolean valid;
  private int bracketsDepth = 0;
  private static char[] precidence = {'^','*','/','-','+'};

  public Main() {
    stack = new ArrayStack();
    input = "";
    looping = true;
    output = "";
  }

  public int getPriority(char value) {
    int val = -3;
    for(int i = 0; i < precidence.length; i++) {
      if(precidence[i] == value) {
        val = i;
        break;
      }
    }
    return (val + 1)/2;
  }

  public boolean validate(char value) {
    if(value == ')' || value =='(' ) {
      return true;
    }
    for(int i = 0; i < precidence.length; i++) {
      if (precidence[i] == value) {
        return true;
      }
    }
    return false;
  }

  public double evaluate(String inputString) {
    double val1 = 0;
    double val2 = 0;
    boolean prevsum = false;
    while(!stack.isEmpty()) {
      stack.pop();
    }
    StringTokenizer st = new StringTokenizer(inputString);
```

```
    while(st.hasMoreTokens()) {
      String temp = st.nextToken();
      //Check if it's an expression or a number
      if(!validate(temp.charAt(0))) {
        try {
          stack.push(Double.parseDouble(temp));
        } catch(Exception e) {
          //This executes If value is something like "." or "..." etc.
          System.out.println("Not a Number!");
          JOptionPane.showMessageDialog(null,"Not a Valid Number!\nPlease Check inputs.");
          System.out.println(e);
          //Returns so the program doesn't crash
          return Double.MAX_VALUE;
        }
      } else {
        try {
          val2 = (double)stack.pop();
          val1 = (double)stack.pop();
        } catch (Exception e) {
          System.out.println("Too many operators!");
          return Double.MAX_VALUE;
        }
        double eval = 0;
        switch(temp.charAt(0)) {
          case '+':
            eval = val1 + val2;
            break;
          case '-':
            eval = val1 - val2;
            break;
          case '/':
            eval = val1 / val2;
            break;
          case '*':
            eval = val1 * val2;
            break;
          case '^':
            eval = Math.pow(val1,val2);
            break;
        }
        prevsum = true;
        System.out.printf("%f %c %f = %f\n",val1,temp.charAt(0),val2,eval);
        stack.push(eval);
      }
    }
    return (double)stack.pop();
  }

  public void loop() {
    while(this.looping) {
      valid = true;
      output = "";
      input = JOptionPane.showInputDialog("Please input an infix expression between 3 and
      ↪  20 Characters");
      System.out.println(input);
      if(input == null) {
        looping=false;
        break;
      }
      if(input.length() < 3 || input.length() > 20) {
```

```
110         JOptionPane.showMessageDialog(null,"Expression is not of valid length please enter
            ↪  a value between 3 and 20 characters","Alert",JOptionPane.ERROR_MESSAGE);
111         continue;
112       }
113       for(int i = 0; i < input.length(); i++) {
114         char val = input.charAt(i);
115         if ((val >= '0' && val <= '9') || val == '.') {
116           int start = i;
117           while((input.charAt(i) >= '0' && input.charAt(i) <= '9') || input.charAt(i) ==
              ↪  '.') {
118             i++;
119             if (i >= input.length()) break;
120           }
121           output += input.substring(start,i);
122           output += " ";
123           i--;
124         } else {
125           if (stack.isEmpty()) {
126             if (this.validate(val)) {
127               //Ignore bracket errors
128               if (val != ')') {
129                 stack.push(val);
130               }
131               /*Ignore Spaces*/
132               if (val == '(') {
133                 bracketsDepth++;
134               }
135             } else if(val != ' ') {
136               JOptionPane.showMessageDialog(null,"Only the following characters are valid:
                  ↪  (,),*,+,/,-,^ and numbers 0-9","Invalid
                  ↪  Character!",JOptionPane.ERROR_MESSAGE);
137               valid=false;
138               break;
139             }
140           } else if (!stack.isFull()) {
141             /*Ignore spaces*/
142             if (val == ' ') {
143               continue;
144             }
145             if (!this.validate(val)) {
146               JOptionPane.showMessageDialog(null,"Only the following characters are valid:
                  ↪  (,),*,+,/,-,^ and numbers 0-9","Invalid
                  ↪  Character!",JOptionPane.ERROR_MESSAGE);
147               valid=false;
148               break;
149             } else {
150               if (val == ')' && bracketsDepth >= 1) {
151                 char temp = (char) stack.pop();
152                   while(temp != '(') {
153                     output += String.valueOf(temp);
154                     output += " ";
155                     temp = (char) stack.pop();
156                   }
157                   bracketsDepth -= 1;
158               } else {
159                 if((char)stack.top() == '(') {
160                   stack.push(val);
161                   bracketsDepth++;
162                 } else {
163                   if (this.getPriority((char)stack.top()) >= this.getPriority(val)) {
```

```
                    stack.push(val);
                  } else {
                    ArrayStack tempStack = new ArrayStack();
                    tempStack.push((char) stack.pop());
                    while(!stack.isEmpty()) {
                      if(this.getPriority((char)stack.top()) >= this.getPriority(val)) {
                        stack.push(val);
                        break;
                      } else {
                        tempStack.push(stack.pop());
                      }
                    }
                    if (stack.isEmpty()) {
                      stack.push(val);
                    }
                    /*Unload temp stack*/
                    while(!tempStack.isEmpty()) {
                      stack.push(tempStack.pop());
                    }
                  }
                }
              }
            }
          }
        }
      }
      //Drain Stack to output string
      while(!stack.isEmpty()) {
        char temp = (char) stack.pop();
        if (temp != ')' && temp != '(') {
          output += String.valueOf(temp);
          output += " ";
        }
      }
      System.out.println(output);
      if(valid) {
        double result = evaluate(output);
        String messageBox = "The result of the expression is:\ninfix: " + input
        ↪  +"\nPostfix: " + output +"\nResult: " + result;
        JOptionPane.showMessageDialog(null,messageBox);
      } else {
        continue;
      }
      /*Yes: 0 No: 1 Cancel: 2*/
      switch(JOptionPane.showConfirmDialog(null,"Would you like to run again?")) {
        case 0:
          System.out.println("Running again");
          break;
          default:
          looping = false;
          break;
      }
      //Flush stack to avoid typecasting errors in the next run through
      while(!stack.isEmpty()) {
        stack.pop();
      }
    }
  }

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

```
223     Main application = new Main();
224     application.loop();
225   }
226 }
```

3 Testing