Cathal McSweeney : 19731485

OOP ASSIGNMENT TWO – INFIX TO POST FIX CALCULATION

Problem Statement:

The problems for this assignment are centralised around four main parts that are to be coded.

The first of which is the reading in of a users input into a string that is then to be validated.

The validation part of the assignment is focused around the parameters that were given to us in which we are to ensure that the user inputs only valid characters as part of their string. The use of 'x' instead of '*' in the equation for multiplying could prove as a common error in user inputs for an example. The way I attacked this problem was by creating an array of only valid characters. Then using a method called "validateString()" I would pass the users String into where a loop is used to assign a char variable each iteration and then using a nested loop is compared to each character contained within the valid array. If a char is not found in the array then a Boolean flag is set to false and the loops are both ended using the break command return a false Boolean value to the line of code that called it. This line of code is contained within a while loop that will iterate infinitely until a Boolean value of true is returned from the validate method.

Next aspect of the assignment to be tackled was the infix to postfix notation. This was tackled using the guidance document provided. The largest issues faced during this was identifying precedence of a operator character. A "precedence()" method was created that was used to identify which chars had greater precedence than others. The class ArrayStack was first introduced during this method as the stack was to be used to store the operator characters. Many obstacles were faced in this method such as how to handle parentheses and then how to remove parentheses so they didn't end up on the postfix string.

The final aspect of the assignment to tackle was to calculate the postfix string and get the correct result for the users equation using only the postfix version of the users initial input. To achieve this the stack was again used by utilising the ArrayStack class. This time however the numbers are to be placed onto the stack opposed to the operators. This allows the code to iterate through the string, when meeting an operator, the top two numbers on the stack are popped off. Depending on the operator then a math equation would be carried out using the two numbers popped off and then the resulting number would be pushed back onto the stack. Each subsequent operator in the string would correspond to two numbers that are to be present on the stack to be calculated with. Once completed the answer is returned and the programme prints out the infix entered, its postfix representation and the answer.

Analysis and Design Notes

Requirements:

- Prompt user to enter infix equation.
- Read in users equation as a string.
- Validate string to ensure that it meets the specifications laid out in assignment PDF.
- Convert from infix notation to postfix notation.
- User postfix notation of the equation to calculate answer.
- Display results of calculation.
- End.

Design:

- Main method
    - used to prompt user for equation.
    - Stores user input as string.
    - Calls validation method.
    - Calls infix to postfix method.
    - Call calculation method.
    - Display user input, postfix and final result to user.

- Validation method
    - Takes in user input string
    - Checks length of string is greater than 3 & less than 20.
    - Loops through each character of the input string.
    - Compares each character of the input string to all the characters contained within a valid character array through a nested loop.
    - If a valid character is found, the next character in the string is checked until all chars have been validated.
    - If a char in the string is not found in the valid char array a Boolean flag is set to false and returned to the main method.
    - A check to ensure that only single digit numbers are entered on the string.
    - A check to ensure that operators are not entered two at a time. Other than beside brackets.
    - Returns Boolean to main method if any errors are found in users string.

- Infix to Postfix method
    - Precedence method is created to ease checking operators.
    - Takes validated string to be translated.
    - Iterates through the string character by character.
    - If the character is a number it appends the number to a string.
    - If the character is a operator checks stack and precedence.
    - Pushes operator to stack if greater precedence than what's on the stack.
    - If top of stack is equal or greater than operator pops stack and appends to string until no longer true then pushes operator onto stack.

    - Identifies parentheses and adds to stack until closing parentheses are found then pops all on the stack until opening parentheses is on top of stack. Then discards the parentheses.
    - Checks stack after infix string has all been assessed to ensure any remaining operators on the stack are appended to the string.
    - Return string to main method

- Calculation method
    - Reads in the postfix notation of the equation as a string.
    - Iterates through each character of the string.
    - If a number is met store on the stack.
    - If an operator is met, store top two numbers on the stack in two variables.
    - Identify what equation is to be carried out through a switch case statement.
    - Carry out the calculation and store result onto stack.
    - Carry out until all operators in string have been used.
    - Return the result to main method

- Precedence method
    - Read in char from infix to post fix method.
    - Using a series of if statements identify which level of precedence the char meets.
    - Returns the value of precedence set in if statements.
    - If brackets return 0.

Testing:

Using various user inputs to test the code and ensure that the validation method is correct along with ensuring that the postfix output is also correct.

- Check that two simultaneous numbers aren't accepted in the validation method.
- Check that two simultaneous operators aren't accepted in the validation method.
- Check the length of the user input is correctly identified by the validation method.
- Ensure that the infix input is correctly converted to postfix.
- Ensure brackets are correctly removed.
- Check that if a number is placed beside a bracket that a multiplication symbol is appended to the calculation in place of the bracket.
- Ensure the calculation carried out is returning the correct result.

CODE

```
import java.awt.*;
import javax.swing.JOptionPane;

public class InFix_To_PostFix {

	private static String inFix, postFix;
	private static double ans;
	private static char[] validChars = {'0','1','2','3','4','5','6','7','8','9','*','/','-','^','+','(',')'};
	private static ArrayStack s = new ArrayStack(20);

	public static void main(String[] args) {

		boolean validated = false;

		//loop that will only run once if the entered calculation is correct. wrong entries result in requesting again input off user
		while(validated == false) {
		inFix = JOptionPane.showInputDialog("Enter Calculation: ");

		//Validates user input in method
		validated = validateString(inFix);
		}

		//shows user what they have ipput to be calculated
		JOptionPane.showMessageDialog(null,"You have entered "+inFix+" to be calculated");

		//changing inFix to postFix
		postFix = inFixToPostFix();

		//calculate user input
		ans = calculate(postFix);

		//final output showing the user the result of their postfix and result of their input
		JOptionPane.showMessageDialog(null,"The result of the expression is: \nInFix: "+inFix+"\nPostFix: "+postFix+"\nResult: "+ans);

	}

	//validation method checks string input of user to ensure its valid
	public static boolean validateString(String check) {
		boolean valid = false;
		boolean tmo = false; //too many operators
```

```
		//checks length of input is correct
		if(check.length()<3 || check.length()>20) {
			return valid;
		}

		//loops through chars in input string checking each one is valid
		for(int i=0;i<inFix.length();i++) {
			char nextInChar = inFix.charAt(i);
			int count=1;

			//checks to make sure that its single digits only
			if(i!=(check.length()-1)) {
				if(Character.isDigit(nextInChar) &&
Character.isDigit(inFix.charAt(i+1))) {
					valid = false;
					break;
				}
			}

			//ensure that users have not entered 2 operators together other than
parentheses
			for(int j = 10; j<15;j++) {
				if(nextInChar==validChars[j]) {
					for(int k = 10; k<15;k++) {
						if(inFix.charAt(i+1)==validChars[k]) {
							tmo = true;
							break;
						}
					}
					if(tmo == true) {break;}
				}
			}
			if(tmo == true) {
				valid = false;
				break;
			}

			//nested loop to compare current char against each valid character in
array
			for(char v: validChars ) {

				count++;//count used to identify if loop has iterated through
length of valid char array & no valid char is found

				//if valid char is found breaks from nested loop onto next char
in string
```

```
                        if(nextInChar==v) {
                                valid = true;
                                break;
                        }

                        //if no valid char is found valid flag set to false and breaks from
nester loop
                        if(count == validChars.length) {
                                valid = false;
                        }
                }
                //if valid flag is set to false breaks from loop and returns a false for the
boolean
                if(valid == false) {
                        break;
                }
        }
        return valid;
    }


    //inFix to postFix function
    public static String inFixToPostFix() {

        String nums = "";

        //for loop iterates through all the chars in the infix String
        for(int i = 0 ; i < inFix.length() ; i++) {
                char curChar = inFix.charAt(i);

                //if current char in string is a number adds it to the string
                if(Character.isDigit(curChar)) {
                        nums += curChar;
                }

                //use the stack to implement the correct order of operators for the
calculation
                else {

                        //adds the current char in the string to the stack depending on
precedence & what may be currently on it
                        if(s.isEmpty() || precedence(curChar)>precedence((char)s.top())
|| (char)s.top()=='(') {
                                s.push(curChar);
                        }
```

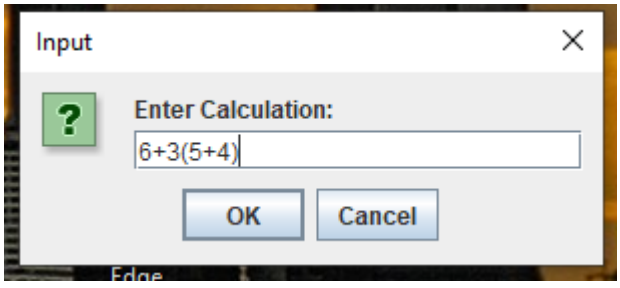
```
                        //pops off operators appends to string depending on current
character & what is already on the stack
                        else {
                            while(precedence(curChar)<=precedence((char)s.top())
&& curChar!='(' && curChar!=')') {

                                //breaks out of loop if a parentheses is met on
the stack
                                if((char)s.top()=='(') {
                                    break;
                                }

                                //continues to pop operators off the stack until
the while == false OR the stack is empty
                                nums+=(char)s.pop();
                                if(s.isEmpty()==true) {break;}
                            }
                            s.push(curChar);
                        }
                    }

                    //pushes a multiply operator to the stack if opening brackets are found
& its not the start of the equation
                    if(curChar=='(' && s.size()!=1) {
                        s.push('*');
                        s.push(curChar);
                        }

                    //when a closing parentheses is met the contents of the stack are
appended to the string until a opening bracket is met
                    if (curChar==')') {
                        while((char)s.top()!='(' && (char)s.top()!=')') {
                            nums+=(char)s.pop();
                        }
                        //the opening parentheses is then popped from stack
                        s.pop();

                    }
                }
                //any operators left on the stack are popped and appended to the string in their
order of precedence
                    while(s.isEmpty()!=true) {
                        if((char)s.top()=='(' || (char)s.top()==')') {
                            s.pop();
                            continue;
                        }
```

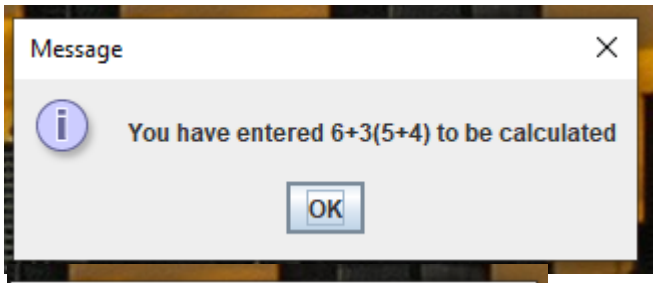
```
                    nums+=(char)s.pop();
                    }
            return nums;
    }

    //calculate the postFix equation
    public static double calculate(String postFix) {
            //shows the postfix equation that needs to be calculated
            JOptionPane.showMessageDialog(null,postFix);

            double num1,num2,result=0;

            for(int i=0;i<postFix.length();i++) {
                    char curChar = postFix.charAt(i);

                    //if the current character is a digit its added to the stack
                    if(Character.isDigit(curChar)) {
                            s.push((double) Character.getNumericValue(curChar));
                    }

                    else {
                            //when an operator is met the next 2 numbers on the stack are
assigned to variables and used in a calculation
                            num1=(double)s.pop();
                            num2=(double)s.pop();
                            switch(curChar) {
                            case '*':
                                    result = num1*num2;
                                    System.out.println(num1+"*"+num2+"="+result);
                                    break;
                            case '/':
                                    result = num2/num1;
                                    System.out.println(num2+"/"+num1+"="+result);
                                    break;
                            case '+':
                                    result = num1+num2;
                                    System.out.println(num1+"+"+num2+"="+result);
                                    break;
                            case '-':
                                    result = num2-num1;
                                    System.out.println(num2+"-"+num1+"="+result);
                                    break;
                            case '^':
                                    result = (int)Math.pow(num1, num2);
                                    break;
                            }
```

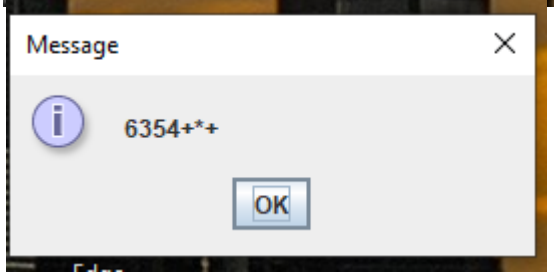
```
                        //the result is then pushed to the stack for further calculations
                        s.push(result);
                    }
                }
                return result;
            }

            //gives each operator a value to identify its precedence as per 'BOMDAS'

            public static int precedence(char x) {
                char check = x;
                int y=0;

                if(check == '^') {
                    y= 3;
                }
                if(check == '*' || check =='/') {
                    y= 2;
                }
                if(check == '+' || check == '-') {
                    y= 1;
                }
                return y;
            }
}
```

TESTING

Correct input by user

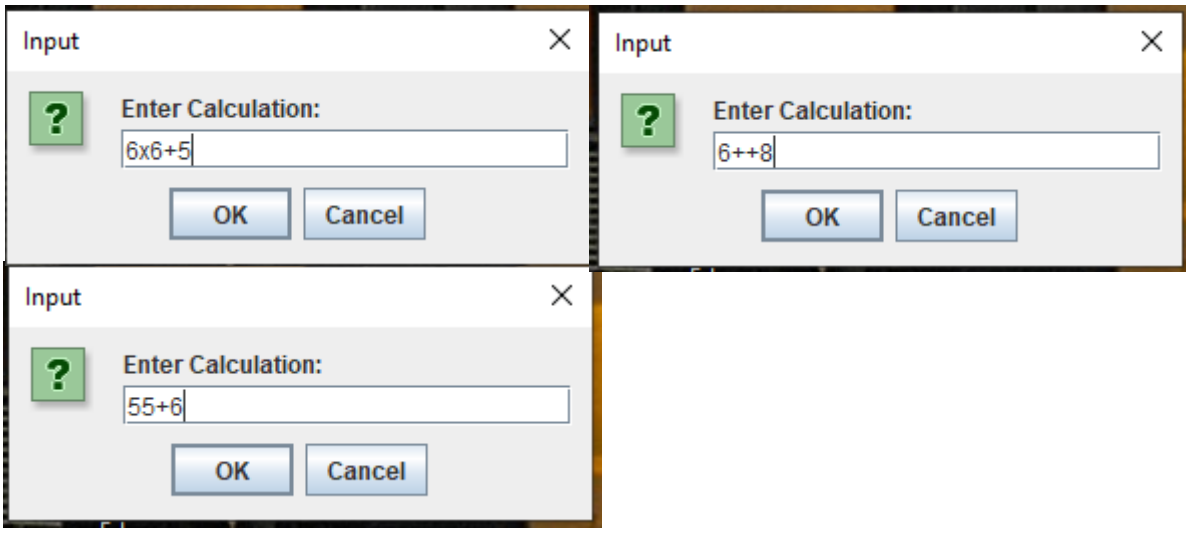




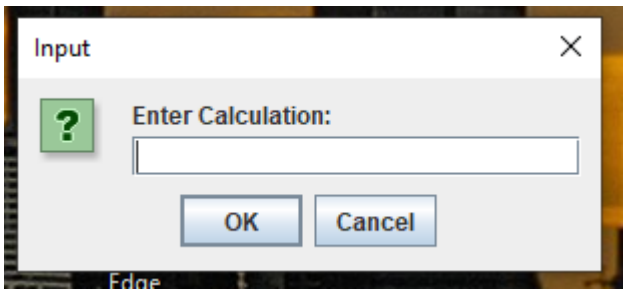


Final message is displayed showing
The infix, postfix and result.

Error input “x” is not a valid character, two operators and two digits are invalid.



Infinitely loops in a while loop until valid input is entered.



Ensuring correct postfix notation is received and calculation works correctly.