# Assignment 2

Object orientated programming : data structures and algorithms

**Problem Statement**

This problem reads in an infix expression given by the user it then converts this expression to a post fix expression and later solves the expression in question.

I think the code will firstly have to read in the values using the scan tool. It will then read these values and differentiate between operands and operators and tier the operators in the correct order in order to correctly convert it to post fix notation.

It will store the expression in a stack and

* if the character is an operand it will be placed in a string
* else if the precedence of the operator is greater than the operator in stack push it
* else pop all operators which are greater than or equal to precedence of the scanned operator and append them into them to the output string.
* Else if it’s a ‘(‘ push it into the string.
* Else If it’s a ‘)’ which means you got to pop all the operators into the output string until the top of the stack is a ‘(‘. Then pop it once more to get rid of the open bracket.

The trickiest thing about this part of the assignment is keeping the if statements organised and making sure they flow correctly through the characters. I think you’ll have to test it very well as you go along as it will be very hard to find your mistakes in this kind of method if you don’t test it incrementally as you progress down through the method

If this logic is carried out correctly we should it should convert from infix to postfix successfully. We then got to evaluate the postfix expression which will be done using stacks also. We essentially place the numbers into the stack until we come across an operator.

Once we come across an operator we got to pop the top two numbers in the stack and carry out the operation on these two numbers.

We’ll have to use if statements though as each of the operators are characters and are unable to carry out mathematical operations. Therefore we’ll have to say if the character equals a certain operation carry out this operation on the top two members of the stack.

The evaluation of the expression will be quite tricky as you will be dealing with a lot of different types as it is an object coming out of the stack and you’ll want to cast it to a character or double to be able to use it in operations or if statements. I think this will be the biggest challenge of the evaluation part of this assignment.

**Code**

**Test**

**import** javax.swing.JOptionPane;

**public** **class** Test {

**public** Test()

{

}

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

PostfixToInfix postfix = **new** PostfixToInfix();

String entered = postfix.getInput();

String postfix1 = postfix.infix(entered);

Double evaluated = postfix.postfixEvaluation( postfix1 );

String str1 = "";

str1 += "Infix" + entered + "\n";

str1 += "Postfix: " + postfix1 + "\n";

str1 += "Evaluated: " + evaluated + "\n";

JOptionPane.*showMessageDialog*(**null**, str1);

}

}

**Stack**

**public** **interface** Stack

{

// most important methods

**public** **void** push(Object n); // push an object onto top of the stack

**public** Object pop(); // pop an object from top of the stack

// others

**public** Object top(); // examine top object on stack without removing it

**public** **boolean** isEmpty(); // true if stack is empty

**public** **boolean** isFull(); // true if stack is full (if it has limited storage)

}

**ArrayStack**

/\*\* Array implementation of Stack ADT \*/

**import** javax.swing.JOptionPane;

**public** **class** ArrayStack **implements** Stack

{

**protected** **int** capacity; // The actual capacity of the stack array

**protected** **static** **final** **int** ***CAPACITY*** = 1000; // default array capacity

**protected** Object S[]; // array used to implement the stack

**protected** **int** top = -1; // index for the top of the stack

**public** ArrayStack() {

// default constructor: creates stack with default capacity

**this**(***CAPACITY***);

}

**public** ArrayStack(**int** cap) {

// this constructor allows you to specify capacity of stack

capacity = (cap > 0) ? cap : ***CAPACITY***;

S = **new** Object[capacity];

}

**public** **void** push(Object element) {

**if** (isFull()) {

JOptionPane.*showMessageDialog*(**null**, "ERROR: Stack is full.");

**return**;

}

top++;

S[top] = element;

}

**public** Object pop() {

Object element;

**if** (isEmpty()) {

JOptionPane.*showMessageDialog*(**null**, "ERROR: Stack is empty.");

**return** **null**;

}

element = S[top];

S[top] = **null**;

top--;

**return** element;

}

**public** Object top() {

**if** (isEmpty()) {

JOptionPane.*showMessageDialog*(**null**, "ERROR: Stack is empty.");

**return** **null**;

}

**return** S[top];

}

**public** **boolean** isEmpty() {

**return** (top < 0);

}

**public** **boolean** isFull() {

**return** (top == capacity-1);

}

**public** **int** size() {

**return** (top + 1);

}

}

**PostfixToInfix**

**import** javax.swing.JOptionPane;

**public** **class** PostfixToInfix {

**public** PostfixToInfix()

{

}

**private** String entered = "";

**private** String postfix = "";

ArrayStack stack = **new** ArrayStack();

**public** String infix(String entered)

{

//message board to prompt the user to enter there infix expression

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

{

//turning the string into a character array

**char** c = entered.charAt(i);

//If the character is a number between 1 and 9 it is placed directly into a string

**if**(Character.*isLetterOrDigit*(c))

{

postfix += c;

}

//if the stack is empty or the operator is greater then the one on the top of the stack or it is a ( it is pushed into the stack

**else** **if** (stack.isEmpty() || foodChain(c) > foodChain((**char**)stack.top()) || c == '(')

{

stack.push(c);

//System.out.println(foodChain((char)stack.top()) );

}

//if the operator has less precedence then operator on top of stack it iterates operators out till all operators that are of higher precedence are removed

**else** **if**( foodChain(c) <= foodChain((**char**)stack.top()) && c != ')' )

{

**while**( !(stack.isEmpty()) && foodChain(c) <= foodChain((**char**)stack.top()) && !((**char**)stack.top() == '(') )

{

postfix += (**char**)stack.top();

stack.pop();

}

stack.push(c);

}

//if there is a ) operator then iterate through stack and place all operators into string until you get to ( operator

**else** **if**( c == ')')

{

**while**( (**char**)stack.top() != '(' )

{

//place all operators before the ( sign into the string

postfix += stack.pop();

}

//pop the final parenthesis and don't put into string

stack.pop();

}

}

//place any remaining operators into the string until stack is empty

**while**(!(stack.isEmpty()))

{

postfix += stack.pop();

}

System.***out***.println(postfix);

**return** postfix;

}

**public** String getEntered()

{

**return** entered;

}

//this method used to find out if the characters are operators or not

**public** **boolean** isOperator(**char** c)

{

**if**( c=='\*' || c=='/' || c=='+' || c=='-' || c=='^')

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

//This method is used to set the precedence of each operator in order of highest to lowest

**public** **int** foodChain(**char** c)

{

**if**(c == '^' )

{

**return** 3;

}

**if**(c == '\*' || c == '/')

{

**return** 2;

}

**else** **if**(c == '+' || c == '-')

{

**return** 1;

}

**else**

{

**return** 0;

}

}

//Evaluating the postfix expression

**public** Double postfixEvaluation(String postfix)

{ //Creating an array stack to place the operands into

ArrayStack evaluation = **new** ArrayStack();

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

{

//converting the output postfix string into characters

**char** c = postfix.charAt(i);

//If it is a number push it into the stack

**if**(Character.*isLetterOrDigit*(c))

{

//before we push the character into the stack we cast it to a string and then cast it to a double

//in order to be able to carry out operations when we pop it out

evaluation.push(Double.*parseDouble*(Character.*toString*(c)));

}

**else** **if**(isOperator(c))

{ //If an operator is encountered pop top 2 operands in stack and cast them to a double

**double** y = (**double**)evaluation.pop();

**double** x = (**double**)evaluation.pop();

//if the operator equals any one of these operations carry out the suggested operation.

//then push it back into the stack

**if**(c == '+')

{

evaluation.push(x+y);

}

**else** **if**( c == '-')

{

evaluation.push(x-y);

}

**else** **if**( c == '\*')

{

evaluation.push( x\*y );

}

**else** **if**( c == '/')

{

evaluation.push( x/y );

}

**else** **if**(c == '^')

{

evaluation.push(Math.*pow*(x, y));

}

}

}

//message pop up to print the answer of the evaluation

**return** (Double)evaluation.top();

}

**public** String getInput()

{

entered = JOptionPane.*showInputDialog*(**null**, "Please enter the expression you want to be solved:");

Boolean correctInput = **true**;

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

{

**char** c = entered.charAt(i);

//if it is between the numbers of 0 and 9, it is a operator or it is a brackets

**if**( (c>='0'&& c<='9') || isOperator(c) == **true** || c == '(' || c == ')')

{

}

**else**//if it isn't one of the above set the boolean to false

{

correctInput = **false**;

}

}

**for**(**int** i = 0; i<entered.length() - 1; i++)

{

**char** c = entered.charAt(i);

**char** p = entered.charAt(i+1);

**if**( Character.*isLetterOrDigit*(c) && Character.*isLetterOrDigit*(p) )

{

correctInput = **false**;

}

}

//Tests that the entered expression is of the correct length

**if**( entered.length() <= 20 && entered.length() >= 3)

{

}

**else**//If it isn't it sets the boolean to false

{

correctInput = **false**;

}

**if**(correctInput == **true**)//if the boolean is true the string follows all the parameters

{

**return** entered;

}

**else** //if it doesn't try re enter the correct expression

{

JOptionPane.*showMessageDialog*(**null**, "There is an invalid character entered, please enter a number between 1-9 or one of the following characters +,-,\*,/,^. Also have your statement longer then three characters and shorter then 20 characters.");

**return** getInput();

}

}

}

**Testing**

Graphical user interface, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Graphical user interface, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

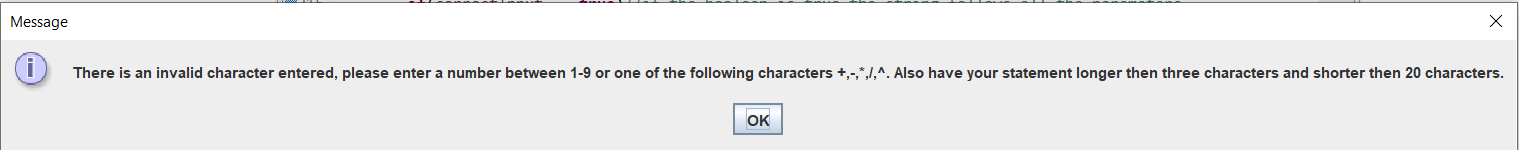
Graphical user interface, application

Description automatically generatedGraphical user interface, text, application

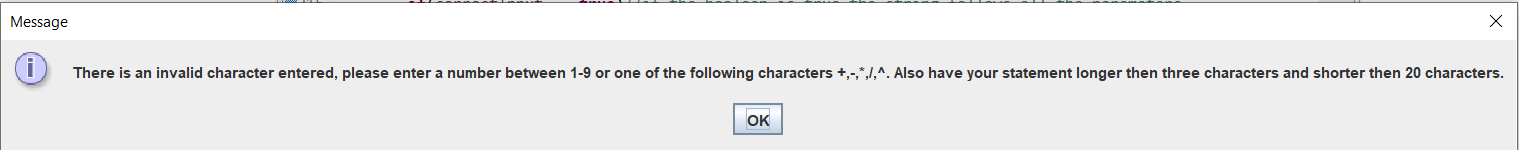
Description automatically generated

**Invalid expressions**

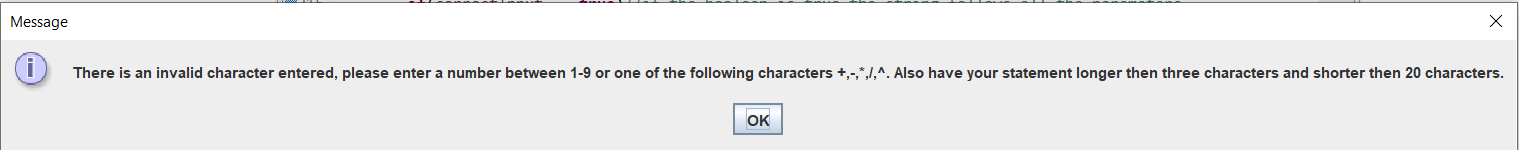
Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

