OOP: Algorithms and Data structures Assignment 2

Problems Statement

The aim of this assignment is for the program to read in a numerical infix expression from the user and convert it to a postfix expression.
It is very easy for humans to read and calculate equations in the infix form whereas computers can't differentiate the operators and parenthesis easily, that's why it is more efficient to convert the expressions from infix to postfix.

Analysis and Design notes

An infix expression is converted to a postfix expression using this Java application, which then evaluates the postfix expression to determine its numerical value. JOptionPane uses a GUI to prompt the user to enter an infix expression. The programme verifies the input's validity and issues an error message if necessary. Using a stack data structure, the input is transformed into a postfix expression, and operators are pulled from the stack based on their precedence levels and appended to the result string. Then, after looping through it and executing arithmetic operations, the postfix expression is evaluated. In this program we will create an input method, infix to postfix method and a postfix evaluator along with other helped methods.

The getOutput() validates a user-entered infix numerical expression falls to method. The user is initially asked to enter a string of characters, which is then saved in the input variable. After that, it generates a character array with the supplied string's length. A set of helper methods, including isValidCharacter(), isValidFirstCharacter(), isValidLastCharacter(), and isValidMiddleCharacter(), are then used to iteratively validate each character in the input string ().

The methods return a boolean value indicating whether the character is valid or not after determining whether it is a numeric, an operator, an open bracket, or a close bracket. Our postfix to infix method is the main method used to carry out our operations on the expressions given by the user and to print the postfix result. We scan from left to right carrying out our checks while pushing and popping where necessary. The precedence of the characters are very important as the computer will calculate the sum in the order of precedence. POP takes the top element off the stack and PUSH adds to the stack.

The method asks the user to submit a new expression and calls itself again if the input length exceeds the range of 3 to 20 characters or if any of the input string's characters are illegal.
The approach additionally counts the number of open and closed brackets and determines whether the final bracket count is balanced. The method asks the user to enter a new expression and then calls itself again if the bracket count is not balanced. The method then returns the character array in if the input is valid.

The infixToPostfix function converts an expression in the infix notation to postfix notation. The code takes an input characters array 'in; and returns the equivalent expression in the postfix notation as a string. This conversion is done using a stack data structure.

If x is an operand, it is appended to the output string. For x as an operator:
if the operator has higher precedence than the one on top of the stack, or the stack is empty, or the stack contains an opening parenthesis, then x is pushed onto the stack. If the operator has lower or equal precedence than the one on top of the stack, then the operators on the stack are popped and

appended to the output string until an operator with lower precedence than x is found. After the end of the loop, any remaining operators on the stack are popped and appended to the output string. The output string now contains the expression in postfix notation.

The function precedenceLevel is used to determine the precedence of an operator, with higher-precedence operators having a higher level.

The postfixEvaluator evaluates a postfix expression and returns the outcome as a double value. A string that represents the postfix expression to be evaluated is sent as the function's input. In order to store values and carry out the expression evaluation, the function makes use of a stack data structure. The function loops through the string's individual characters, and if it encounters a digit, it pushes the character's numerical value on the stack. If an operator is found, the top two values are removed from the stack, the operation is carried out, and the outcome is pushed back into the stack. At the function's conclusion, the final outcome is returned and stored in the variable output.

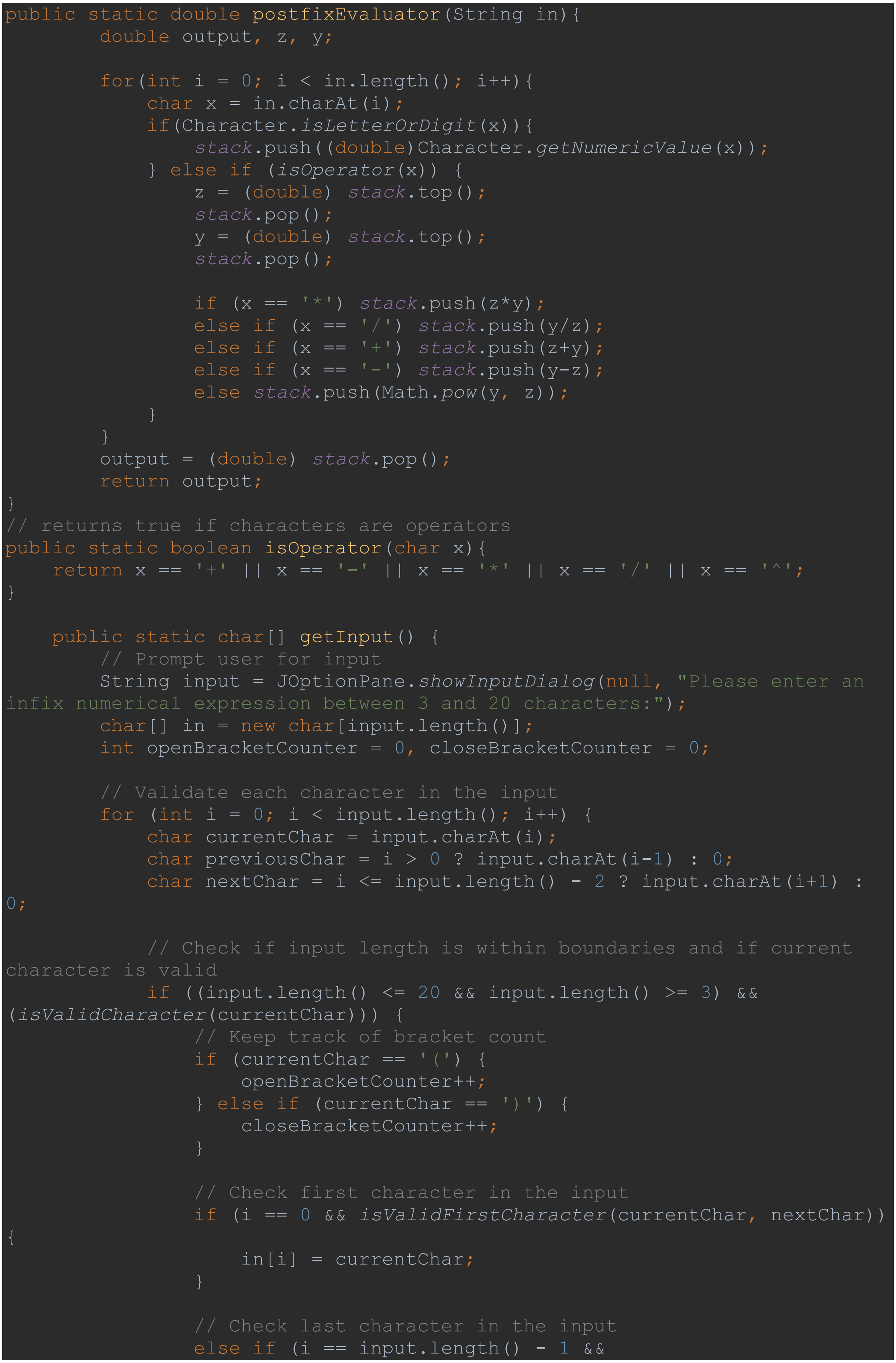
We also created an isOperator method to avoid writing unnecessary code that would involve writing out the operators multiple times. Our precedence level method assigns the operators their appropriate precedence values that the computer must take into account. i.e. ^ has the higher precedence followed by the * and / signs then the + and - .

We use the arrayStack from our example code and in our stack.java interface we declare our stack operations.

Code

Test.java

```
import javax.swing.*;
import java.lang.*;

public class Test {
    // create a new stack
    private static ArrayStack stack = new ArrayStack();

    public static void main(String[] args)  {
        char[] in = getInput();
        String postfix = infixToPostfix(in);
        JOptionPane.showMessageDialog(null, "Postfix: " + postfix);
        double answer = postfixEvaluator(postfix);
        JOptionPane.showMessageDialog(null, "Answer: " + answer);
    }

    // return the precedence of the given operators with the higher values
    // having higher precedence
    public static int precedenceLevel(char x){
        // switch statement to determine the precedence levels of the
operators
        switch (x){
            case '+':
            case '-':
                return 1;

            case '*':
            case '/':
                return 2;

            case '^':
                return 3;
        }
        return -1;
    }

    // convert infix to postfix
    public static String infixToPostfix(char[] in) {
        StringBuilder output = new StringBuilder();

        // for loop to loop until the end of the array
        for(char x : in){
            // if character is an operand: append (add) to output string
            if(Character.isLetterOrDigit(x)){
                output.append(x);
            }
            // PUSH if scanned operator is > in precedence, empty or
contains an opening parenthesis
            else if((precedenceLevel(x) == 2) || stack.isEmpty() ||
stack.contains( '(' )){
                stack.push(x);
                continue;
            } else { // Pop operators >= in prec and append to output
                while(precedenceLevel(x) >= 1 && !stack.isEmpty() &&
!stack.contains('(')){
                    output.append(x);
                    stack.pop();
                }
                stack.push(x); // push operator
            }
            if(!stack.isEmpty()){
                if(x == '('){
                    stack.push(x);
                }
                if(x == ')'){
                    stack.pop();
                    while(!stack.contains( '(' )){
                        output.append(x);
                        stack.pop();
                    }
                    stack.pop();
                }
            }
        }

        // Once the whole string has been scanned, pops all operators on
the stack to output string
        while (!stack.isEmpty()) {
            if ((char)stack.top() != '(') {
                output.append((char) stack.top());
            }
            stack.pop();
        }
        return output.toString();
    }

    public Test(){

    }

    // utility method, returns value for an operator based on its
precedence
```



```
public static double postfixEvaluator(String in){
        double output, z, y;

        for(int i = 0; i < in.length(); i++){
            char x = in.charAt(i);
            if(Character.isLetterOrDigit(x)){
                stack.push((double)Character.getNumericValue(x));
            } else if (isOperator(x)) {
                z = (double) stack.top();
                stack.pop();
                y = (double) stack.top();
                stack.pop();

                if (x == '*') stack.push(z*y);
                else if (x == '/') stack.push(y/z);
                else if (x == '+') stack.push(z+y);
                else if (x == '-') stack.push(y-z);
                else stack.push(Math.pow(y, z));
            }
        }
        output = (double) stack.pop();
        return output;
}
// returns true if characters are operators
public static boolean isOperator(char x){
    return x == '+' || x == '-' || x == '*' || x == '/' || x == '^';
}

    public static char[] getInput() {
        // Prompt user for input
        String input = JOptionPane.showInputDialog(null, "Please enter an
infix numerical expression between 3 and 20 characters:");
        char[] in = new char[input.length()];
        int openBracketCounter = 0, closeBracketCounter = 0;

        // Validate each character in the input
        for (int i = 0; i < input.length(); i++) {
            char currentChar = input.charAt(i);
            char previousChar = i > 0 ? input.charAt(i-1) : 0;
            char nextChar = i <= input.length() - 2 ? input.charAt(i+1) :
0;

            // Check if input length is within boundaries and if current
character is valid
            if ((input.length() <= 20 && input.length() >= 3) &&
(isValidCharacter(currentChar))) {
                // Keep track of bracket count
                if (currentChar == '(') {
                    openBracketCounter++;
                } else if (currentChar == ')') {
                    closeBracketCounter++;
                }

                // Check first character in the input
                if (i == 0 && isValidFirstCharacter(currentChar, nextChar))
{
                    in[i] = currentChar;
                }

                // Check last character in the input
                else if (i == input.length() - 1 &&
isValidLastCharacter(currentChar)) {
                    in[i] = currentChar;
                }

                // Check middle characters in the input
                else if (isValidMiddleCharacter(currentChar, previousChar,
nextChar) || isOperator(currentChar)) {
                    in[i] = currentChar;
                }

                // If input is invalid, prompt user again
                else {
                    JOptionPane.showMessageDialog(null, "Only the following
characters are valid: +, -, *, /, ^, (, ) and numbers 0-9 in single
use\n");
                    return getInput();
                }
            } else {
                // If input is invalid, prompt user again
                JOptionPane.showMessageDialog(null, "Only the following
characters are valid: +, -, *, /, ^, (, ) and numbers 0-9 in single
use\n");
                return getInput();
            }
        }

        // Check if bracket count is balanced
        if ((openBracketCounter + closeBracketCounter) % 2 == 0) {
            return in;
        } else {
            JOptionPane.showMessageDialog(null, "Only the following
characters are valid: +, -, *, /, ^, (, ) and numbers 0-9 in single
use\n");
            return getInput();
        }


    }
    // these are our method to check for the valid inputs of the characters
    public static boolean isValidCharacter(char currentChar) {
        return Character.isDigit(currentChar) || isOperator(currentChar) ||
currentChar == '(' || currentChar == ')';
    }

    public static boolean isValidFirstCharacter(char currentChar, char
nextChar) {
        return Character.isDigit(currentChar) || (currentChar == '(' &&
(Character.isDigit(nextChar) || nextChar == '('));
    }

    public static boolean isValidLastCharacter(char currentChar) {
        return Character.isDigit(currentChar) || currentChar == ')';
    }

    public static boolean isValidMiddleCharacter(char currentChar, char
previousChar, char nextChar) {
        return Character.isDigit(currentChar) || (currentChar == ')' &&
(Character.isDigit(previousChar) || previousChar == ')')) || (currentChar
== '(' && (Character.isDigit(nextChar) || nextChar == '('));
    }
```

```
}
```

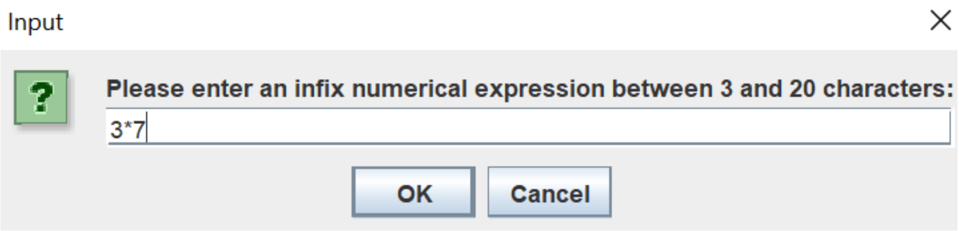
Stack.java

```
public interface Stack {
    public void push(Object n);
    public Object pop();
    public Object top();
    public boolean isEmpty();
    public boolean isFull();
}
```

ArrayStack.java

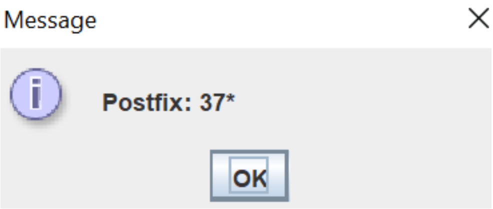
```
import javax.swing.JOptionPane;
import java.util.Stack;

public class ArrayStack extends 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 Object push(Object element) {
            if (isFull()) {
                JOptionPane.showMessageDialog(null, "ERROR: Stack is
full.");
                return element;
            }
            top++;
            S[top] = element;
            return 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);
        }
}
```

Test

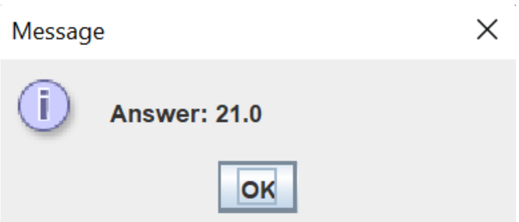


Please enter an infix numerical expression between 3 and 20 characters:

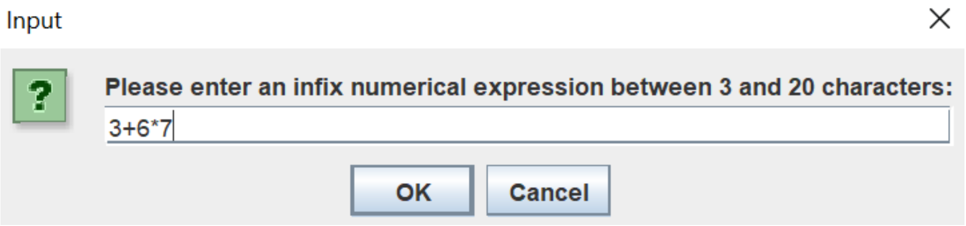
3*7



Postfix: 37*

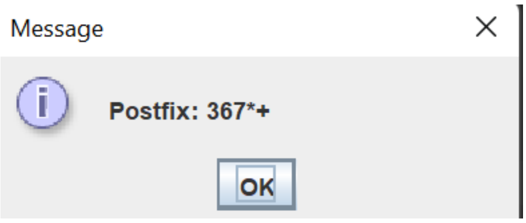


Answer: 21.0

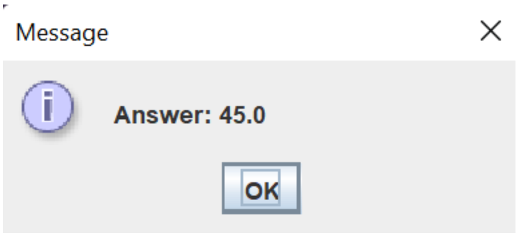


Please enter an infix numerical expression between 3 and 20 characters:

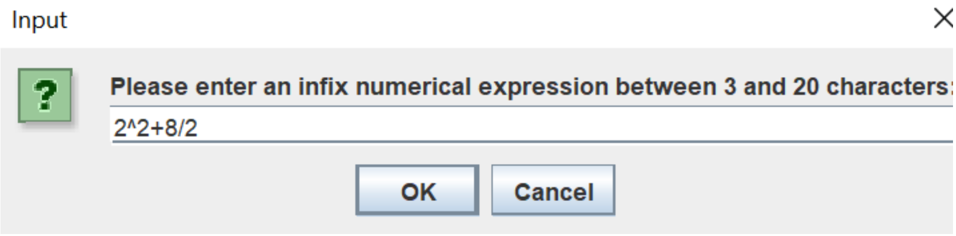
3+6*7



Postfix: 367*+

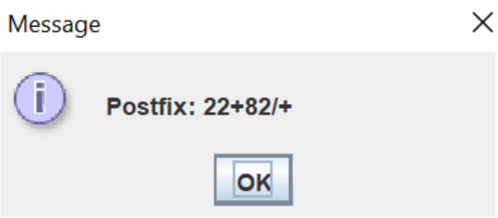


Answer: 45.0

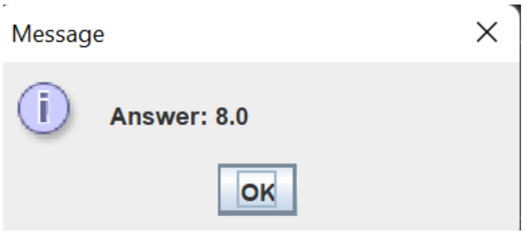


Please enter an infix numerical expression between 3 and 20 characters:

2^2+8/2

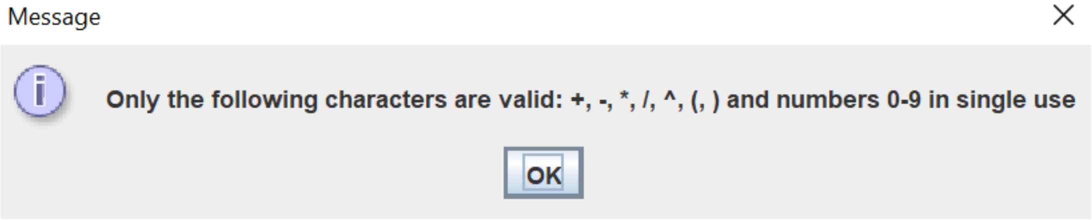


Postfix: 22+82/+

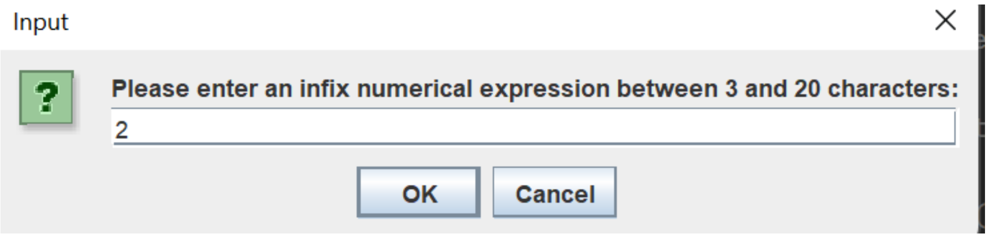


Answer: 8.0

Examples of error in the input and output message:

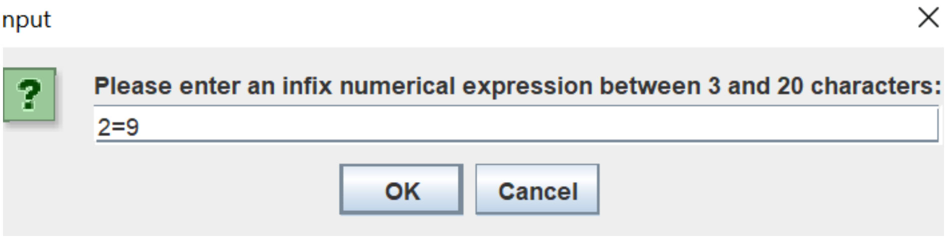


Only the following characters are valid: +, -, *, /, ^, (,) and numbers 0-9 in single use



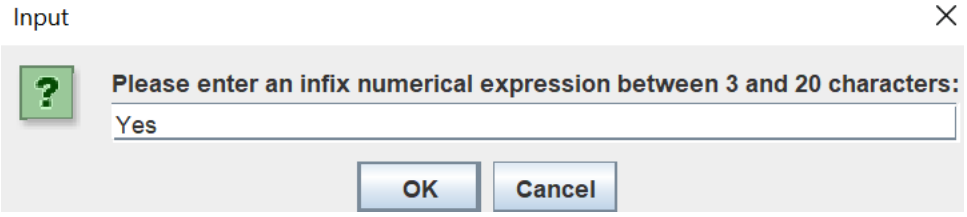
Please enter an infix numerical expression between 3 and 20 characters:

2



Please enter an infix numerical expression between 3 and 20 characters:

2=9



Please enter an infix numerical expression between 3 and 20 characters:

Yes