Developers' Guide

Assignment 1: Gui Calculator and EvaluationTester

CS413 Spring 2016 Professor Ilmi Yoon

February 13,2016

Jeffrey Ilar

Table of Contents

Introduction:

The task of Assignment One is to create an expression evaluator that derives the answer correctly given correct string input. The Assignment also users asks to create a GUI program which allows users to easily create and solve their own expressions. Two programs EvaluatorTester.jar and EvaluatorTestGui.jar were created to complete these two tasks. This guide covers how these programs were developed as well as how to use them.

EvaluatorTester.jar is used in the command window, and allows users to type multiple expressions that can be solved at the same time. EvaluaterTestGui.jar provides an easy to understand interface that allows users to enter an expression through button presses.

Both programs rely on the same Evaluator Class to solve expressions. This class only takes integers as operands and can cover the four basic mathematical operations, as well as the use of parentheses and the power operator. We were given several hints as to how to implement parenthesis and right association, such multiple ways to hand priority and the suggestion of method overriding.

Given Resources:

The complete EvaluatorTester.java source code was given to us to use. The basic outlines of the classes EvaluaterTestGui and Evaluator were given to us. We were also given both the Operand Class and Operater Superclass with suggested methods to place inside them.

Scope of work:

In EvaluaterTestGui.java the method actionPerformed(ActionEvent arg0) needed to be implemented in order to read input and evaluate the expression. Also the “.” button needed to be redesigned as the “^” button for the power operator and an Evaluator variable needed to be initialized. The eval method of the Evaluation class needed to be completed, more specifically parenthesis and right association needed to work . The methods of the Operater Class, its subclasses, and the operand Class needed to be implemented as well. Mutiple Expression tests were performed to make sure both programs work correctly.

Assumptions:

Development of classes were performed in Netbeans IDE 8.0.2 environment. Basic order of operations rules were followed. Operations (+, -, *, /) follow left association rules while ^ operator follows right association.

How to Use Jar Files:

EvaluatorTester.jar must be executed through the command window. To run, in the command window first move to the directory to where it is located. Then input the following line:

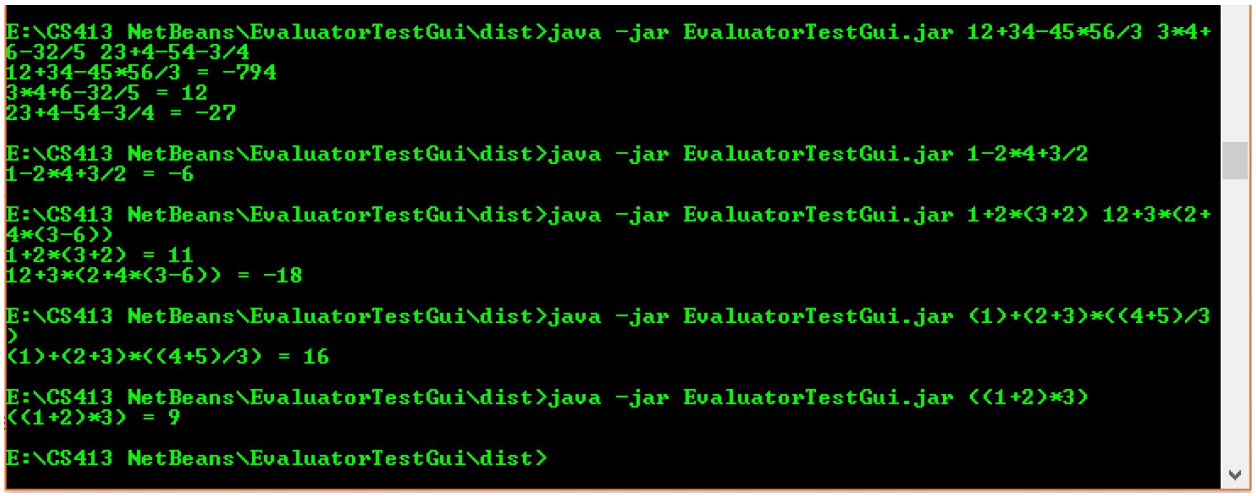
java -jar EvaluatorTester.jar Expression<WhiteSpace>Expression2 ect...

Some things to note:

-the available operators are “+,-,*,/,^,(,)” and must be inputted as such
-when you are using the power operator the whole expression must be quoted or the command line won't read it correctly
-When finished typing expressions hit enter to execute

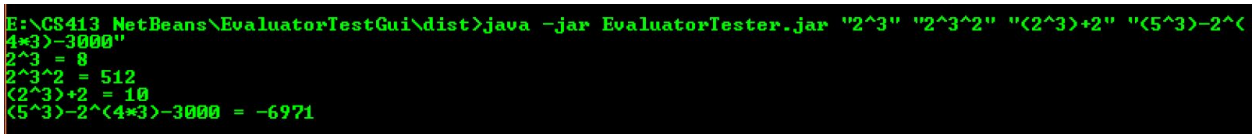
Below are screenshots demonstrating EvaluatorTester.jar:

1.)iLearn Test Cases:

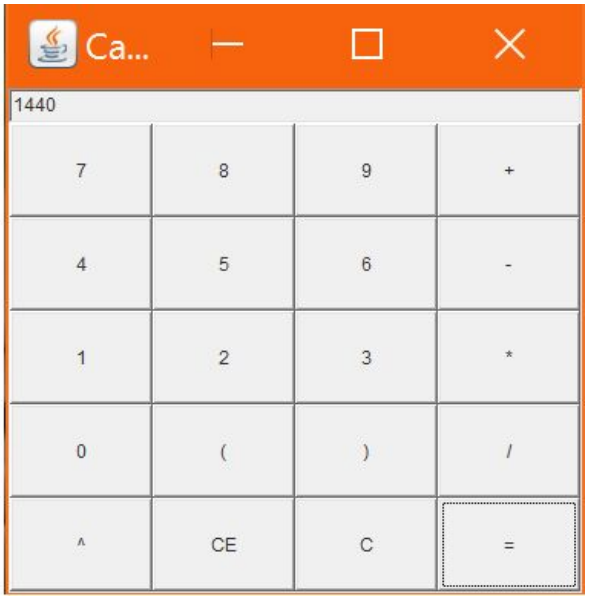


(Note :In this screenshot EvaluatorTester.jar was misnamed EvaluatorTestGui.jar)

2.)Demonstration of Power Operator and and right association though four test cases.



To execute EvaluatorTesterGui.jar simply execute the file. The interface will pop up and you can proceed to create an expression. Clicking the “C” button erases the last input, while “CE” while erase the entire expression. When you are done creating the expression pressing equal will evaluate it. Below are screenshots of the interface solving the expression “45*32”.



Implementation Details:

There are a total of thee java source files: Evaluator.java, EvaluatorTester.java and EvaluatorTestGui.java.

Evaluator java contains several classes. The operand class, the operator superclass and its subclasses, and most importantly the Evaluator class which evaluates the expressions for both jar files.

EvaluatorTester.java and EvaluaterTestGui.java contains the classes EvaluatorTester and EvaluaterTestGui respectively. These classes create the interfaces of the .jar files.

Class Overview:

Evaluator Class:

The Evaluator class contains two stacks:

```
private Stack<Operand> opdStack;
private Stack<Operator> oprStack;
```

These stacks are both used in evaluating the expression, which will be explained further down. The evaluator class contains three methods.

```
Evaluator()
int eval(String expr)
int eval(Stack<Operand> opd, Stack<Operator> opr)
```

Evaluator() is the constructor method, it initializes opdStack and oprStack.

The method int eval(String expr) is where the evaluation of the expression takes place. First off, pushes the # operator into the stack, and then concatenates a “!” at the end of the expression to signify the end of it. In effect these two operators make the code more efficient because the while loop does not have to check if the stack is empty with each iteration.

```
public int eval(String expr) {
        String tok;
        expr=expr.concat("!");                          //adds ! to the end of the sting, act
        oprStack.push(Operator.operators.get("#"));
        String delimiters = "+-*^/#!() ";

        StringTokenizer st = new StringTokenizer(expr,delimiters,true);

        while (st.hasMoreTokens()) {
                if ( !(tok = st.nextToken()).equals(" ")) {
                        if (Operand.check(tok)) {
                                opdStack.push(new Operand(tok));
                        } else {
                                if (!Operator.check(tok)) {
                                        System.out.println("*****invalid token******");
                                        System.exit(1);
                                }

                                Operator newOpr = Operator.operators.get(tok); // POINT 1
```

String expr is then tokenized so that each piece of it can be placed into the correct stack. The first while loop makes sure all tokens are placed into a stack. The program exits if the token is not valid.

```
                        while (((Operator)oprStack.peek()).priority() >= newOpr.priority()&&
                        newOpr.priority()!=3 && newOpr.priority()!=6){
                        Operator oldOpr = ((Operator)oprStack.pop());

                        if (oldOpr.priority()==3)
                                break;

                        Operand op2 = (Operand)opdStack.pop();
                        Operand op1 = (Operand)opdStack.pop();
                        opdStack.push(oldOpr.execute(op1,op2));
                        }
                if(newOpr.priority()!=2){                                   //close parenth
                        oprStack.push(newOpr);
                }
            }
        }
}
return eval(opdStack, oprStack);
```

Below explaining the above code first priority numbers need to be talked about. Here is the priority table of the operations

Operation	Priority
#	0
!	1
(	3
)	2
+-	4
*/	5
^	6

The while loop has three conditions. The first condition looks at the newest token Operation priority number and compares it to the priority number of the operation at the top of the stack. If it's smaller than the one in the stack the while loop executes. Inside the while loop the top operator is popped from its stack, and the last two operands are popped from thiers. The operator then executes the method execute(operand1, operand2) which takes in the two operand and pushed the result into the operand stack. The if statement inside the while loop ensures that (never affects the operations to the left of it.

The second while condition of the while loop "newOpr.priority()!=3" deals with parenthesis. It ensures once a ")" is detected everything inside the parenthesis gets evaluated. "(" is never actually pushed onto the stack.

The third while condition allows for right association, because ^ has the highest priority so it should be executes right to left. Which does not happen in the current loop (the method int eval(Stack<Operand> opd, Stack<Operator> opr) performs the evaluation through right association)

When there are finally no tokens int eval() executes
int eval(Stack<Operand> opd, Stack<Operator> opr) and returns the value.

Lastly, the method int eval(Stack<Operand> opd, Stack<Operator> opr) takes the two stacks and repeats the while loop for any remaining operations in the stack. Before it finishes it pops the opdStack twice to remove the ! and # operaters. It also pops the remaining operand from the operand stack and returns its integer value. The only time this method is used is as the return value for int eval(String expr).

Abstract Operator:

```
abstract class Operator {
    static HashMap <String, Operator> operators= new HashMap<String, Operator>();
    static{
        operators.put("#", new BeginOperator());
        operators.put("!", new TerminationOperator());
        operators.put("+", new AdditionOperator());
        operators.put("-", new SubtractionOperator());
        operators.put("*", new MultiplicationOperator());
        operators.put("/", new DivisionOperator());
        operators.put("^", new PowerOperator());
        operators.put("(", new OpenOperator());
        operators.put(")", new CloseOperator());
    }
```

The abstract operator is a superclass. Because it is abstract it can not be initialized. Instead, the abstract class initializes a HashMap, operators, and sets the keys and values. The keys are the operations, and the resulting value is the representative subclass of the operator. In effect this makes the code more efficient because a new subclass does not need to be initialized each time its operation appears in the expression

Operator contains three methods:

boolean check(String tok){
abstract int priority();
abstract Operand execute(Operand op1, Operand op2);

The method check(string tok) takes a string token and checks if it is a valid operand in the hashmap though the use of containtsKey(string) method.

int priority() and Operand execute() are both abstract classes, which basically means it's a promise to the subclasses that they will implement these methods. int Priority() returns the respective priority number of the subclass that it is implemented in. Operand execute(Operand op1, Operand op2) takes two operands and does the respective operation between them according to the subclass execute is implemented in. For the operations (,),!, and # there execute return value is null, because they do not actually do anything to the operands.

Operator Subclasses:

The following classes are all subclasses to the Operand Superclass.

class BeginOperator()
class TerminationOperator()
class AdditionOperator()
class SubtractionOperator()
class MultiplicationOperator()
class DivisionOperator()
class PowerOperator()
class OpenOperator()
class CloseOperator()

Each of these subclasses implement the abstract classes int priority() and Operand Execute(Operand op1, Operand op2). (Explanation of the methods works can be found in Abstract Operator section).

Operand Class:

The Operand Class contains a single variable:

int val- Hold the numerical value of the operand

It also holds three methods:

Operand (string tok)
Operand (int value)
static boolean check(String tok)

Operand (string tok) and Operand (int value)are the class constructors. Operand(string tok) makes use of the Interger.parseInt(string) method to convert a string to an int and then sets val to that value. The method check(string tok) is used before Operand(string tok) is called to be sure that the constructor takes a string that can be converted to an int.

EvaluatorTester Class

The EvaluatorTester class contains the main method for EvauatorTester.jar multiple strings as an argument, then passes each one into eval(String) of the initialized Evaluator. A print line shows both the initial expression as well as its evaluation.

EvaluatorTestGui Class

The EvaluatorTestGui class contains the main method for EvauatorTestGUI.jar. It initializes 5 vaiables: a TextField, Panel, of 20 Buttons, Array of a all possible inputs (including a clear and clear everything input), and lastly an evaluator.

It contains 2 methods:

void EvaluatorTestGui()

void actionPerformed(ActionEvent arg0)

EvaluatorTestGui() was given as a resource. It provided the template of the caculator, and labels each button through use of the input array and a for loop.

```
if(arg0.getSource()==buttons[0])
        txField.setText(txField.getText()+"7");          //concat 7 to text field
```

The method ActionPerformed(ActionEvent arg0) directs the action of each button press. Because most of the button presses revolve around adding to a text field the use of mutiple if statements were used. .getText() was used to recieve the string from the text field, and then concatinating with the respctive character represented by the button. .setText () was then used to replace the TextField with the new string. CE and C buttons directives made similar use of .getText and setText(). Lastly, when the = button is pressed, it takes the string of the TextField, and puts into the eval method. It then turns the returned int value into a string, and places that string into the TextField through .setText().

```
if(arg0.getSource()==buttons[19]){
        int answer=e.eval(txField.getText());
        String toString=String.valueOf(answer);
        txField.setText(toString);
}
```

Results and Conclusions:

1) The result of this assignment is GUI Calculator and command line Program which both solve expressions in the same manner.
2) The assignment was a lesson in critical thinking. To create a functional eval() method you must have full understanding of how the operands move through the stack according to priority.
3) Further development is needed to create a more efficient eval() method. In the larger while loop an if statement was added to prevent the “)” from being placed on the stack. Another if statement was placed inside the inner loop to prevent the “(“ from executing it’s execute method. That being said, I feel accomplished in the current efficiency I was able to produce. I was able to prevent the use of a separate method to execute right association, as well as remove third if statement which further optimized the code.
4) Further development may also include more exception cases to test bad input.