CSC 413 Project Documentation

Spring 2021

Kyle Harvey

915139815

CSC 413-01

https://github.com/csc413-su21/csc413-p1-k-harvey

Table of Contents

1 Introduction

1.1 Project Overview

For this assignment, our goal is to help build a program that's an Expression Evaluator being able to look an expression with any operator and give the correct results. Aside from being able to calculate any expression, it will also have a fully functional Calculator GUI to do our calculations on.

1.2 Technical Overview

For a more in-depth overview of this project, we are provided with a semi complete Evaluator class which handles most of the evaluating portion when it takes the operands and operators the user inputs. Within this class we are to complete the evaluateExpression method initializing the operator stack by putting the necessary operator priority which also includes the priority of any operator in the operator stack besides the mathematical operators such as "+-*/" which would be less than the priority of the usual operators that we would use. The other class that we need to implement is our Operand class. This is where we need to put together ourselves to use in representing an operand in a valid mathematic expression. We need our Operands to be able to construct operand in the expressionToken in a string and an int value. We also need to return the value of the operand so we fill in the getValue method. Lastly in the Operand class we'll be checking the expression with a Boolean to check if the given token is a valid operand returning true and false. Next we'll need to complete our abstract Operator class. In here we the instance of a HashMap that would use keys as the tokens we're interested in the most whereas the values will be instances of the operators. Once we put together our HashMap, we make sure we have our priority, execute, getOperator, and check methods with proper returns for all the operators. After we put together the most important classes and created new operators in a separate file for our program to read to and know what to do with it, it should be able to run and pass all the EvaluatorTests. Then we could focus on the Calculater GUI and how we want the interface to perform similar to what we would see on a calculator with all the button functions.

1.3 Summary of Work Completed

What I contributed to this assignment to get it to work correctly, I started with the Operand class. From there got the Operator class to work and put together my HashMap. Next I made a separate class file for each Operator such as Add, Subtract, Multiply, Divide, Power, and Parenthesis that executes those functions. Once all those were working, I put together my Evaluator Class. This is where some things started to not work properly. Although doing a test for each operator, all of them passed except when it came the parenthesis, only 9 passed while 7 of them failed in the Evaluator test. Something seems to be wrong with stack logic as the when it comes to doing Parenthesis on the left, it causes expressions to not pass.

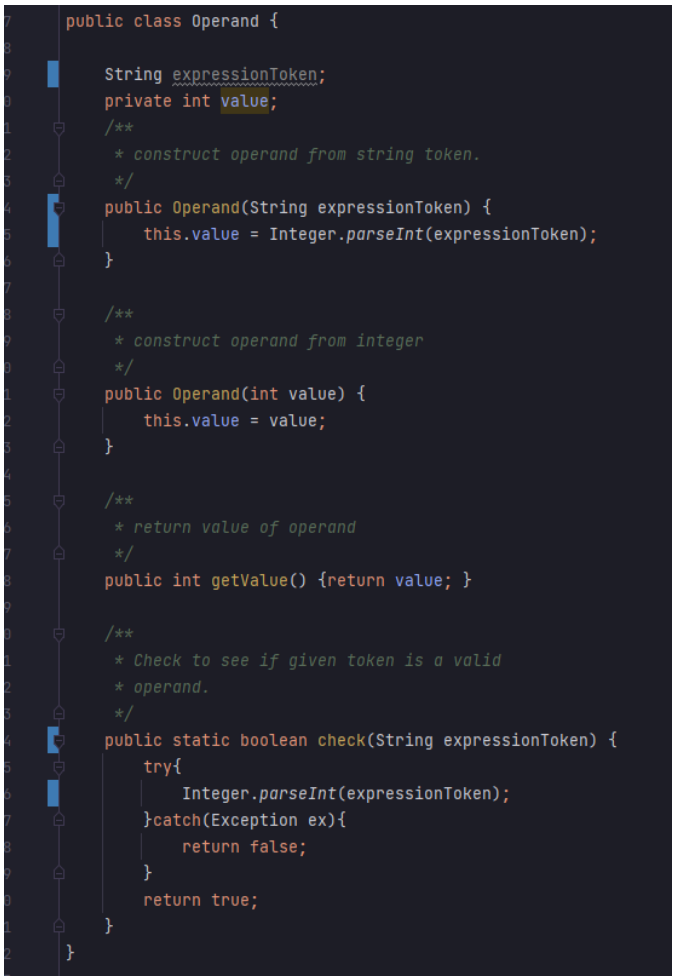
2 Development Environment

Version of Java Used: Java 16

IDE Used: InteliJ IDEA 2021.1.2 Ultimate Edition

3 How to Build/Import your Project

I imported this project from Github using Git to clone the project on to my Desktop from there. Began working and building the project. The parts I contributed to get the assignment to work correctly, I first started in the Operand class:



```
public class Operand {

    String expressionToken;
    private int value;
    /**
     * construct operand from string token.
     */
    public Operand(String expressionToken) {
        this.value = Integer.parseInt(expressionToken);
    }

    /**
     * construct operand from integer
     */
    public Operand(int value) {
        this.value = value;
    }

    /**
     * return value of operand
     */
    public int getValue() {return value; }

    /**
     * Check to see if given token is a valid
     * operand.
     */
    public static boolean check(String expressionToken) {
        try{
            Integer.parseInt(expressionToken);
        }catch(Exception ex){
            return false;
        }
        return true;
    }
}
```

Here I added in the private int value to be used in this class. Next I swapped out token to be experssionToken similar what's used in the other classes but assigning it the value there, that Operand method would be able to construct operands from both string token and integers when also assign the value to that value for the method. I also changed the getValue method to return that value of the operand when called. Lastly I filled out the Boolean check method that checks to see if the given token is a valid operand if it catches an exception returns false if not it's true.

For the Operator class, we implemented the HashMap for all the operator classes I created to make as priority within a stack. We have the Add, Subtract, Multiply, Divide, Power, Left, and Right Operators all that we would be using and testing.

```java
static final HashMap<String, Operator>operators = new HashMap<>();


public abstract int priority();

//private static Map<String, Operator> operators;


static{

    operators.put("+", new AddOperator());
    operators.put("-", new SubtractOperator());
    operators.put("*", new MultiplyOperator());
    operators.put("/", new DivideOperator());
    operators.put("^", new PowerOperator());
    operators.put("(", new LeftParenthesisOperator());
    operators.put(")", new RightParenthesisOperator());
}
```

Also in the Operator class we need to have an abstract method to execute an operator given for two operands.

```java
/**
 * Abstract method to execute an operator given two operands.
 * @param operandOne first operand of operator
 * @param operandTwo second operand of operator
 * @return an operand of the result of the operation.
 */
public abstract Operand execute(Operand operandOne, Operand operandTwo);
```

We have the getOperator method in there as well to retrieve an operator from our HashMap that we made.

```java
/**
 * used to retrieve an operator from our HashMap.
 * This will act as out publicly facing function,
 * granting access to the Operator HashMap.
 *
 * @param token key of the operator we want to retrieve
 * @return reference to a Operator instance.
 */
public static Operator getOperator(String token) { return operators.get(token); }
```

Lastly in our Operator Class we have the Boolean check method to make sure that the given token is a valid operator.

```java
/**
 * determines if a given token is a valid operator.
 * please do your best to avoid static checks
 * for example token.equals("+") and so on.
 * Think about what happens if we add more operators.
 */
public static boolean check(String token) {
    return token.equals("+")||token.equals("-")||token.equals("*")||token.equals("/")||token.equals("^")||token.equals("(")||token.equals(")");
}
```

In order to get our Operator class's HashMap to work properly we need to make our operators, to make it easier to know what each operator is doing I've separated all of them into their own classes to perform their functions.

AddOperator class to add together two operands:

```java
public class AddOperator extends Operator{
    @Override
    public int priority() { return 1; } //return 2
    @Override
    public Operand execute(Operand operandOne, Operand operandTwo){
        Operand total = new Operand( value: operandOne.getValue() + operandTwo.getValue());
        return total;
    }
}
```

SubtractOperator class to substract two operands from each other:

```java
public class SubtractOperator extends Operator{
    @Override
    public int priority() { return 1; } //return 2
    @Override
    public Operand execute(Operand operandOne, Operand operandTwo){
        Operand total = new Operand( value: operandOne.getValue() - operandTwo.getValue());
        return total;
    }
}
```

MultiplyOperator class to multiply the two operands to each other:

```java
public class MultiplyOperator extends Operator{
    @Override
    public int priority() { return 2; } //return 3
    @Override
    public Operand execute(Operand operandOne, Operand operandTwo){
        Operand total = new Operand( value: operandOne.getValue() * operandTwo.getValue());
        return total;
    }
}
```

DivideOperator class to divide the two operands from each other:

```
public class DivideOperator extends Operator{
    @Override
    public int priority() { return 2; } //return 3
    @Override
    public Operand execute(Operand operandOne, Operand operandTwo){
        Operand total = new Operand( value: operandOne.getValue() / operandTwo.getValue());
        return total;
    }
}
```

PowerOperator takes the operand given and equates the power that the second operand is set:

```
public class PowerOperator extends Operator{

    @Override
    public int priority() { return 3; } //return 4

    @Override
    public Operand execute(Operand operandOne, Operand operandTwo){
        Operand total = new Operand(power(operandOne.getValue(),operandTwo.getValue()));
        return total;
    }
    public int power(int i, int j){
        int total =i;
        for(int count =2; count <=j; count++){
            total = total *i;
        }
        return total;
    }
}
```

LeftParenthesisOperator uses priority to set and return the first operand once executed:

```
import edu.csc413.calculator.evaluator.Operand;

public class LeftParenthesisOperator extends Operator {
    @Override
    public int priority(){ return 0; }

    @Override
    public Operand execute(Operand operandOne, Operand operandTwo) { return operandOne; }
}
```

RightParenthesisOperator uses priority to set and return the first operand once executed:

```java
public class RightParenthesisOperator extends Operator {
    @Override
    public int priority() { return 1; }

    @Override
    public Operand execute(Operand operandOne, Operand operandTwo) { return operandOne; }
}
```

Once we got our Operand class and Operator class together, we can focus on the Evaluator Class which will go through an expression evaluating it. There were parts of it provided to help implement it but for the rest of the stack we needed to figure out how it will be taking in those expressions. I've added the addition operators to the delimiters so we have all the operator classes that I made to use.

```java
public class Evaluator {

  private Stack<Operand> operandStack;
  private Stack<Operator> operatorStack;
  private StringTokenizer expressionTokenizer;
  private final String delimiters = " +/*-^()";
```

I created a process method to go through the stack and peek to see if the priority is more than 1 from there it'll pop both operand two and one then executing it putting it a result. At the end it'll push that result back into the stack then popping the whole process out the stack. We'll be able to use this else were when calling this method.

```java
public void process(){
  while(operatorStack.peek().priority() > 1){
    Operator operatorFromStack = operatorStack.pop();
    Operand operandTwo = operandStack.pop();
    Operand operandOne = operandStack.pop();
    Operand result = operatorFromStack.execute( operandOne, operandTwo );
    operandStack.push( result );
  }

  operatorStack.pop();
}
```

Using the Operator class that contains an instance of a Haspmap, we can call it all to the Evaluator class by setting it to a newOperator each time. We write some if and else statements to take in our Left and Right parenthesis into our stack. It checks the expression for each thing and continues until it gets to the while loop to check that stack. If the stack isn't empty and the top of the stack is greater than or equal priority the new operator it will then process and push that operator out.

```
        // TODO Operator is abstract - these two lines will need to be fixed:
        // The Operator class should contain an instance of a HashMap,
        // and values will be instances of the Operators.  See Operator class
        // skeleton for an example.
        Operator newOperator = Operator.getOperator(expressionToken);



        if(expressionToken.equals("(")){
          operatorStack.push(newOperator);
          continue;
        }

        else if(expressionToken.equals(")")){
          process();
          operatorStack.pop();
          //operatorStack.push(newOperator);
          continue;
        }



        else if(operatorStack.isEmpty()){
          operatorStack.add(newOperator);
          continue;
        }

        while (operatorStack.peek().priority() >= newOperator.priority() ) {
          // note that when we eval the expression 1 - 2 we will
          // push the 1 then the 2 and then do the subtraction operation
          // This means that the first number to be popped is the
          // second operand, not the first operand - see the following code
          //process();
          Operator operatorFromStack = operatorStack.pop();
          Operand operandTwo = operandStack.pop();
          Operand operandOne = operandStack.pop();
          Operand result = operatorFromStack.execute( operandOne, operandTwo );
          operandStack.push( result );
        }

        operatorStack.push( newOperator );
      }
    }
```

Lastly we have another while loop to check look at the top of the stack to see if priority is less than zero repeating the process method. The reason for that is to make sure for us to finish the evaluation, we need to empty the stack so we keep evaluating the operator stack until it's empty.

```
    while (operatorStack.peek().priority() > 0 ) {
      //process();

      Operator operatorCur = operatorStack.pop();
      Operand operandTwo = operandStack.pop();
      Operand operandOne = operandStack.pop();
      Operand result = operatorCur.execute(operandOne, operandTwo);
      operandStack.push(result);

    }
    // Control gets here when we've picked up all of the tokens; you must add
    // code to complete the evaluation - consider how the code given here
    // will evaluate the expression 1+2*3
    // When we have no more tokens to scan, the operand stack will contain 1 2
    // and the operator stack will have + * with 2 and * on the top;
    // In order to complete the evaluation we must empty the stacks,
    // that is, we should keep evaluating the operator stack until it is empty;
    // Suggestion: create a method that processes the operator stack until empty.

    return operandStack.pop().getValue();
  }
}
```

The last class we need to add on to for this project is the EvaluatorUI class, which is the GUI for the calculator once we have a fully function Evaluator for all the expressions it could calculate. Right now it's set up to look like a calculator that you would see if you opened your PC's calculator applications except the buttons don't work. We need to write the code for taking in those to make the actionEventObject to generate when a button is pressed. I just wrote the an if statement for each button by copying and pasting but changing the button text number for each one. I also included the try and catch for the invalid token exception in case it gets thrown in.

```
public void actionPerformed(ActionEvent actionEventObject) {

    if(actionEventObject.getSource()==buttons[0]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[0]);
    }
    if(actionEventObject.getSource()==buttons[1]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[1]);
    }
    if(actionEventObject.getSource()==buttons[2]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[2]);
    }
    if(actionEventObject.getSource()==buttons[3]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[3]);
    }
    if(actionEventObject.getSource()==buttons[4]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[4]);
    }
    if(actionEventObject.getSource()==buttons[5]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[5]);
    }
    if(actionEventObject.getSource()==buttons[6]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[6]);
    }
    if(actionEventObject.getSource()==buttons[7]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[7]);
    }
    if(actionEventObject.getSource()==buttons[8]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[8]);
    }
    if(actionEventObject.getSource()==buttons[9]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[9]);
    }
    if(actionEventObject.getSource()==buttons[10]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[10]);
    }
    if(actionEventObject.getSource()==buttons[11]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[11]);
    }
    if(actionEventObject.getSource()==buttons[12]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[12]);
    }
    if(actionEventObject.getSource()==buttons[13]){
```

```
    }
    if(actionEventObject.getSource()==buttons[14]){
        Evaluator cal = new Evaluator();

        try {
            expressionTextField.setText(Integer.toString(cal.evaluateExpression(expressionTextField.getText())));
        } catch (InvalidTokenException e) {
            e.printStackTrace();
        }

        //expressionTextField.setText(Integer.toString(cal.evaluateExpression(expressionTextField.getText())));
    }
    if(actionEventObject.getSource()==buttons[15]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[15]);
    }
    if(actionEventObject.getSource()==buttons[16]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[16]);
    }
    if(actionEventObject.getSource()==buttons[17]){
        expressionTextField.setText(expressionTextField.getText() + buttonText[17]);
    }
    if(actionEventObject.getSource()==buttons[18]){
        expressionTextField.setText(" ");
    }
    if(actionEventObject.getSource()==buttons[19]){
        expressionTextField.setText(" ");
    }
    if(actionEventObject.getSource()==buttons[20]){
        expressionTextField.setText(" ");
    }

}
}
```

4 How to Run your Project

The project runs when you click run after building the project to make sure it compiles properly with no errors. Although there isn't a compiling issues, there's still issue with expressions not outputting correctly when it comes to Parenthesis.

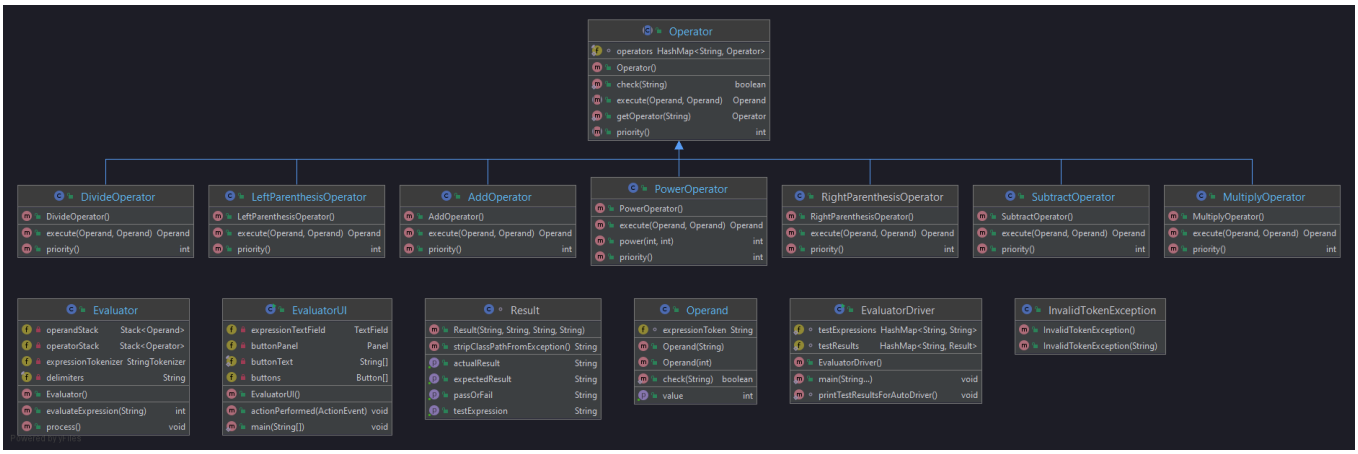
5 Assumption Made

Some assumption made when it came to designing this project that it would be easy to follow from the videos alone but that was not the case. Most of the design and implementation portion came from help off others on how to do so or where to start. Also putting together what I found on how to implement certain commands. Although I thought it would be simple that was not entirely the case.

6 Implementation Discussion

For implementing this project, it seemed we needed to start with the simpler classes such as Operand and work our way into Operator. Once we do both we could create our operators that we'll be using in an expression and need to evaluate it which is where the Evaluator class comes into play. This part is where most the difficulty came from, I tried to follow the logic for going through the expression but the issues mainly came from the Parenthesis, everything else worked out fine but half the tests were failing while the other half passed. For the most part running this program will have the Calculator GUI working and functioning but the outputs will be incorrect if using a parenthesis.

6.1 Class Diagram



7 Project Reflection

My thoughts on this project, although it wasn't too difficult in the sense of what we needed to do the hard part was trying to figure out where to start and go from there. Most the time consuming part came from piecing everything to together and making sure they work with each other.

8 Project Conclusion/Results

Overall this project was a good experience for learning Java programming, taking things from what we learned in pervious classes our goal was to finish the incomplete project and get it working properly. Although we started with the simple calculator, the functionality of it can get a bit more complex. Slowly going through the program and learning what we needed to do to get portion of the code to work and to pass the built in tests helped us understand where we went wrong and what we could fix. Although I wasn't able to pass all the tests provided and was very close, I think given a bit more time I could have gotten the Evaluator test to work. Hopefully this will give us more experience for the next assignment.