CSC 413 Project Documentation

Spring 2019

German Perez

918637543

CSC 413.03

GitHub Repository Link

https://github.com/csc413-03-spring2020/csc413-p1-germanp123.git

Table of Contents

1 Introduction

1.1 Project Overview

The program creates a virtual calculator that evaluates basic operations using the order of operations. The user has limited operations key set {+, -, *, /, (,)} and may use any positive integer numbers as it's operands. The calculator also recognizes errors such as invalid keys, improper double operators (i.e. 2**3), and imbalances of parenthesis.

1.2 Technical Overview

We designed the calculator program from 3 separate classes. In our Operator class, we defined all the operators from the set and implemented how each would execute two operands. We provided methods like getOperator to retrieve an operator, check to verify if the key entered is a valid operator, priority so the calculator can know which operator has the highest precedence before executing, and method execute!

In the Operand class, we have a method named check to verify if the key entered is a valid operand, getValue which retrieves the integer value of that operand, and two operand methods which take different parameter types and transforms it into an operand.

In our Evaluator class, we will have 2 separate stacks which will help us operate our mathematical expressions. In our main code, we iterate through the string expression entered by the user (or test cases) and once verified as a *sound* mathematical expression, we evaluate it**. We run through while loops and if and else statements until finally we come up with a result and return that value.

** Note: it's not necessarily in that order. Our code catches invalid tokens anywhere inside our expression. As for parenthesis that don't closed, that error can only be caught right towards the end of our code as we are evaluating.

1.3 Summary of Work Completed

As stated, we implemented three classes to create the functionality of the calculator. Our Operator class is actually an abstract class, meaning we can only create instances through its subclasses. So, we created subclasses for each key operator such as AddOperator, MultiplyOperator, etc. Since priority and execute methods were originally coded as abstract in the Operator class, those methods needed to be overridden in the subclasses according to each operator. Writing the execute method was simple because it was a matter of doing basic math; for the AddOperator, the two operands in the parameters were added together and returned the value. In the priority method, we assigned the following priority values:

[^] = 3, [/, *] = 2, [+, -] = 1, [(,)] = -1 (we will explain why this is important on the next page).

Operand class didn't have the same issue as the Operator class since it was public, we could create instances directly, so each method created in Operand was defined. Inside our Operand class the methods were called: check which verified valid operands and returned a Boolean value, two Operand constructors so both could take different parameters (string and int) to transform them into operands and getValue which returned an integer value from an operand key.

After completing those two classes, we began working on the Evaluator class. Since our Evaluator and Operand classes are both public and they belong inside the same project, we can create instances of Operand in our Evaluator class easily. At the top of our Evaluator class, we declared some private data members operandStack, operatorStack, expressionTokenizer and some delimiters. We created instances of Stack for both stacks. Inside our main code, we begin iterating through our string expression, that was taken as a parameter, one token at a time. We will treat operators {+, *, -, /, (,), } we find as delimiters but they will still be used as tokens themselves. If a token is an empty space [2 +3], we will skip it. The second if statement checks for valid operands, correct operands will be push inside the operandStack. Otherwise, an exception will be thrown for invalid operands (such as "c" or "2.3"). If our token is an operator, it will retrieve an instance of that operator's subclass and it will be assigned to newOperator. All "(" operators will be immediately pushed inside the operatorStack. If the token is ")", then while the operatorStack isn't empty and the peek's priority is greater than 1, then we will be continuously looping through the method processOperatorStack until those conditions are no longer valid. Afterwards we pop the top operator from the operatorStack. *[We will explain the processOperatorStack later].* We also check for the case when operatorStack is empty, then we immediately push the operator token inside the operatorStack and move to the next token. Or else, we enter a while loop that verifies our operatorStack isn't empty and our stack's peek's priority is greater than our current token's priority then we can use the method processOperatorStack. Once exiting the while loop, we again push the current operator token from the stack. Finally, once we have finally iterated through our whole expression, we reach another while loop. When the operatorStack isn't empty and its peek's priority is greater than one, then we keep looping through the processOperatorStack. Once exiting that while loop, we have the last if statement to check when there's still a "(" inside the operatorStack. If so, this is problematic because we have already traversed through the whole expression and haven't encountered a ")" to close it with, so we throw an exception and state we have an imbalance of parenthesis. Lastly, we return our result which should be the last key left inside our operandStack and use the getValue method to convert it to integer.

The method processOperatorStack is a private method accessible only to the Evaluator class, it doesn't take parameters and doesn't have a return type. If called, it will essentially evaluate our stacks. It pops an operator from our operatorStack and pops two operands from our operandStack and uses the unique execute method based on what operator we have, then records that Operand answer in result. We then push that Operand result back into our operandStack. Which explains why our result should be located in our operandStack.

2 Development Environment

My Java is version 12.0.2 and I use IntelliJ version 2019.3.2 (Community Edition) as my IDE platform.

3 How to Build/Import your Project

I started with opening my MacBook's terminal used command *cd Documents* to reach the documents folder. Once there, I cloned my GitHub repository typing *git clone <URL>* and I retrieved the HTTPS URL from GitHub where it states, "Clone or Download". Once the repository is cloned in my documents, I opened IntelliJ and it asked me what folder I'd like select as my root. I made sure

not to select **csc413-p1-germanp123** as my root folder but **Calculator** instead. Afterwards, I followed the simple steps on IntelliJ to completely import my project and ta-da! It was uploaded.

4 How to Run your Project

To run my test cases, I hover my mouse over the test package and extend it. Inside is a folder called java, I right-click it to select *Run 'All Tests'* . My program runs through all the tests and verifies my program is functioning properly.

To run my virtual calculator, I open the main package, open java, open edu.csc413.calculator and open the folder called evaluator. Inside I right-click EvaluatorUI and select Run 'EvalutorUI.main()' which opens and runs my virtual calculator.

5 Assumption Made

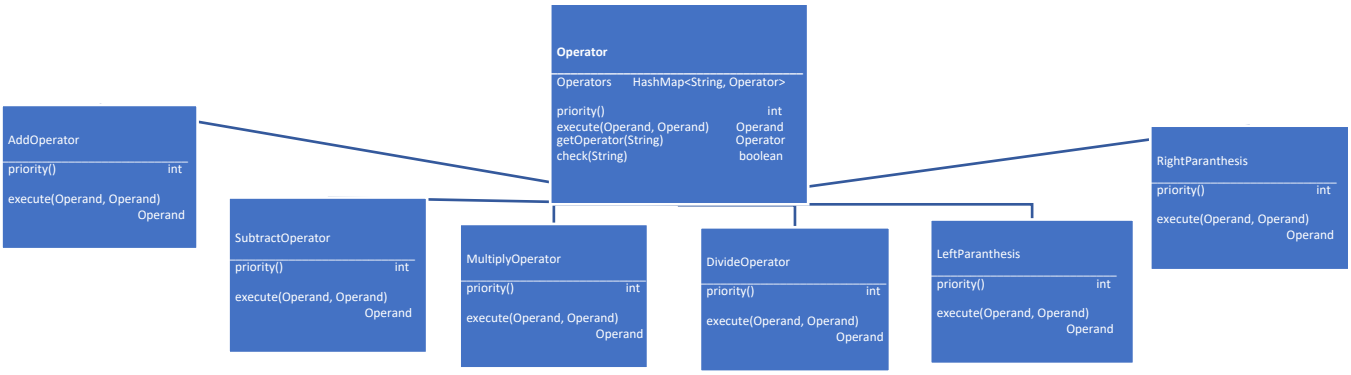
We need to assume that the user will only be entering operand values as positive integers and will use normal mathematical expressions that abide by our operations set {+, -, *, /, (,)}.

6 Implementation Discussion

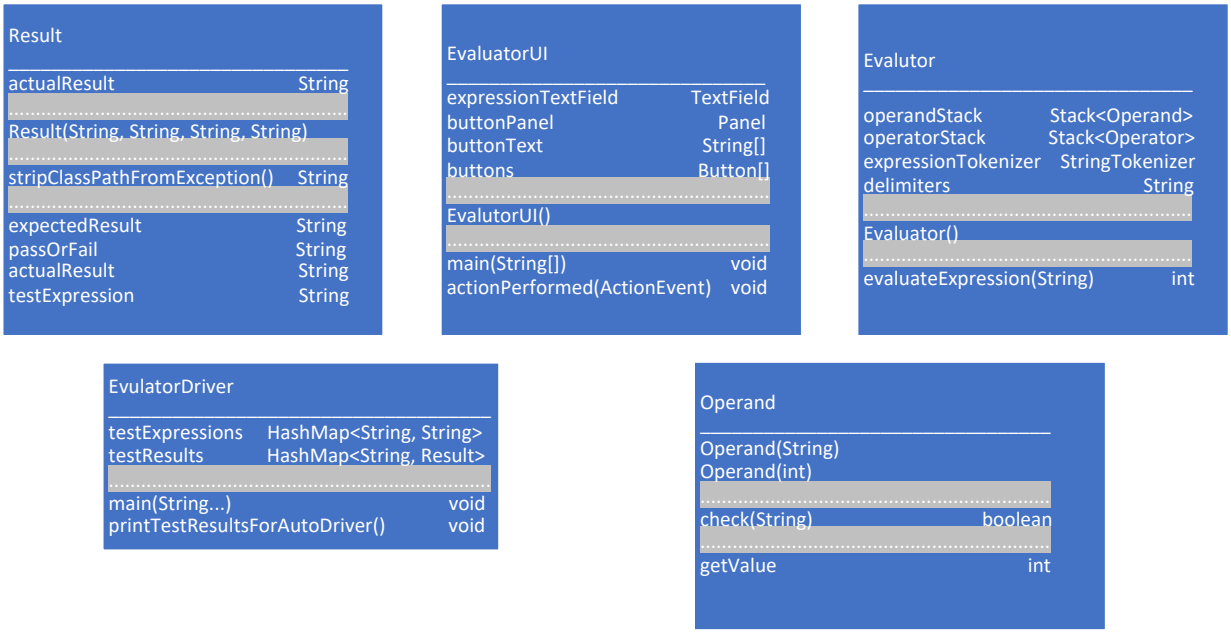
I chose to implement a HashMap in the Operator class because my professor suggested it, I would of probably taken a more tedious route if it wasn't for him. In my evaluator class I choose to use a lot of if-and-else statements because I was inspired by pseudocode uploaded in our stack channel for our class. I'm comfortable implementing them and using while loops throughout my code. Initially, I had completed my code without creating the processOperatorStack, but I realized I had 5 lines of code that kept being repeated inside three of my while loops. So, I shortened my code by adding that method and calling it a few times.

6.1 Class Diagram

This diagram below shows our abstract parent class, Operator, connected to its subclasses of operators. This diagram belongs to a package called operators. Inside each class and subclass, is a list of methods they possess [notice that Operator is the only class with a data member, Operators]. As we can see, the methods *priority* and *execute* were overridden is ALL of the subclasses because they were defined as abstract in the Operator class.



The diagram right below us is a list of classes that belong to the Evaluator class. Each class is represented in its own blue box which contains their own data members, method, and main code. The key words hanging on the right side is the return type for that method.



7 Project Reflection

This project was quite fun to work on. Initially it was quite challenging to keep track of the iteration with all the if loops, while loops, and stacks. When my professor suggested the debugging feature,

it made the whole project a lot simpler. I was able to make meaningful progress once I knew where exactly my program was crashing instead of tweaking my code line by line.

8 Project Conclusion/Results

The project was able to function properly, and it passed all of my tests. What I have learned was how to effectively use the debugging tool to complete my code and I refreshed my knowledge on abstract classes and creating instances in subclasses. My code is able to capture the incorrect mathematical expressions and calculate the correct ones.