Table of Contents

1. Introduction

1.1. Project Overview:

This Expression Evaluator, does exactly what it sounds like, it evaluates mathematical expressions. This can also be referred to a Calculator. The calculator programmed here, evaluates addition, subtraction, multiplication, division, exponents, and even takes parenthesis into account. Since it takes all the previously stated operators into account when mathematically commuting an expression, it is also capable of using PEMDAS, which is the order (from left to right) that mathematical expressions and equations are solved in: Parenthesis, Exponents, Multiplication, Division, Addition, and Subtraction. This expression evaluator is also linked to a GUI (Graphic User Interface) calculator; which makes it easier for the user to simply use this code as if they were using an actual calculator.

1.2. Technical Overview

The main data structures which we are working with to create this calculator are Stacks and a HashMap; another main component is the StringTokenizer method. To begin understanding the code, one can look at Evaluator.java class. Here is where we create the operatorStack, where our operators will get pushed into, and our operandStack, where our operands will get pushed into. These stacks are important because in order to actually solve the expression, we need to begin by sorting and storing our values, and in the end, emptying the stacks to execute the calculation. The StringTokenizer method is also crucial, since it reads our string input by the user, which is compromised of Strings which we translate to either operators or operands. Our delimiters for the StringTokenizer, are equal to our operators. The string operator allows us to actually consider our delimiters, aka our operators which we defined, as tokens too; this allows us to store them in the operatorStack. We end up going though if statements which help us scan the expression and determine if it's an operand, which immediately gets pushed the operatorStack. Now, once we determine the next token is not an operand, or an invalid token, we are safe to say it is an operator. This is where we, now, construct an object ('newOperator') of the Operator.java class to access our method '.getOperator(Token)' which will actually determine (based off the token), which operator class to call; this has to be done this way since the Operator class is abstract. Our operators are operator classes (AdditionOperator, MultiplyOperator, etc.,) are inside a HashMap, inside our Operator.java class; the token passed through '.getOperator' returns the value, which is one of the operator class. This is the convenience of using a HashMap. This newOperator created is now used in the following while loop to continue our process which will eventually lead to the execution of our expression. First we begin by checking if the operator is an open or closed parenthesis, since these have the greatest priority according to PEMDAS. Following the assignments outline of the logic behind the stack: If the token is "(", and `Operator` object is created from the token, and pushed to the operator `Stack;` If the token is ")", the `process Operator`s until the corresponding (is encountered. Pop the(`Operator`. This is a crucial step for the expression evaluator to properly process parenthesis. If the operator does not fit into any of these cases, then we still continue to process the operators, as long as the stack is not empty and the operator we are processing is not of greater priority the the operator at the top of the stack . The priority is outline in the assignment document, from 0-3, following PEMDAS. Once these conditions are met we are able to begin popping out the two operands (our numbers), and then popping our operator and executing the operands according to which Operator class is called. We then push the result back into the operandStack and continue the process or continue the while loop until we have no more tokens to scan. This is where the evaluateHelper() method comes in which allows the process to continue, as long as the operatorStack is not empty, but once it is empty we then pop the operandStack to return the int result. The Operand.java class basically serves the purpose of storing our numbers as operand values ('operandValue') and creating a .getValue() method to retrieve these integers within our operator classes performing execution. This is how the Evaluator.java method implements Operator.java, Operand.java, and the rest of the operator classes created.

The GUI portion involves creating several if statements which serve to translate the buttonPressed, to the visual output of the buttonPressed onto the calculator. The operator and operand keys did not require accessing the evaluator class, but configuring the "=" button to actually do something with the operands and operators did require getting the text in expressionTextField and calling the evaluateExpression method which is what does most of the work in Evaluator.java. After this, the Evaluator.java did the work, so the int result we got from calling evaluateExpression was placed in the expressionTextField. The "CE" function of the calculator, cleared all the input, so this only required the expressionTextField to be blank: " ". The "C" function was used to delete the last string, and can do so until there is nothing left in the expressionTextField on the calculator; this was done using the subString method which originally removes the first and last string, but ignoring the removal of the first string. All the if statements for the operator and operand buttons was placed in a Try Catch, incase the user inputs invalid entries (ex: 2+2+++*/)); if they enter an invalid entry they will be prompted to press CE and try again.

1.3. Summary of Work Completed

For this project I was able to successfully complete the EvaluatorDriver.java, EvaluatorUI.java, Operand.java, and all the Operator classes. The only thing that I could not figure out was how to get an accurate result when the input expression involves an operand directly next to a parenthesis (ex: (2+2)3; 3*5*4(4-2)). In general, when a number is next to a parenthesis it is the equivalent of multiplying like normal, but my program requires and operator before any parenthesis expression. Other than that, as long as an operator comes before a parenthesis and after, it runs. It passes all the tests and the GUI is able to access the Evaluator.java class and accurately print results. The work is explained in more detail in the Technical Overview.

2. Development Environment

a. Java Version - 16.0.1

b. IDE - IntelliJ

3. How to Build/Import your Project

Once you copy the HTTP link from GitHub, from my repository, you can use your terminal to perform git clone and it will clone a repository into a new directory. An easier way might be to go to GitHub and search for the GitHub repository link (https://github.com/csc413-su21/csc413-p1-adadallison). If you press code, now you can see 'Download ZIP files,' which will allow the folder to directly download to your computer. To import the files correctly, you can open your IDE and selected "import project," select the folder and be sure to select the *calculator folder* as the source root of your project. The project should be named "calculator."

4. How to Run you Project

Now that the project is opened. A user who is interested in using the Calculator GUI, should simply go to the EvaluatorUI.java class and run it. This will simulate the GUI calculator, which looks sort of like a calculator on a smartphone, and from there it is to be used like a normal calculator. It will continuously run so you can calculate expression after expression, even after clearing it (using CE). If a user is interested in using the calculator with out the GUI, then running the EvaluatorDriver.java class is to be ran. The user will be prompted to "Enter an Expression:" and the same input requirements apply for the Calculator GUI. If a user enters an expression that is not valid input, like consecutive operators, then they will be prompted to re-enter their expression on the Calculator GUI; if this occurs in the EvaluatorDriver.java file, then the program will exit out due to an the error aka exception and will need to be ran again.

5. Assumption Made

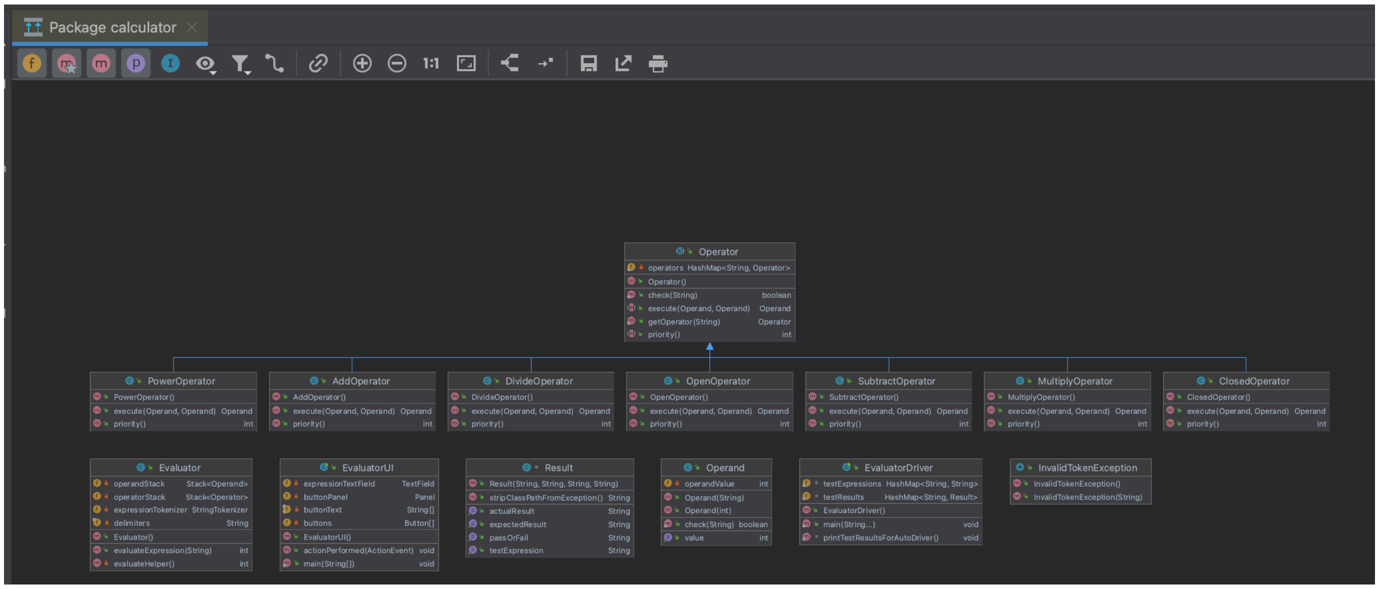
I assumed we were going to create the GUI and expression evaluator from scratch, but I guess I'm thankful we did not have to. I did not think we would use a hash map, but now I understand why. I also assumed that the operations could be done in one class. I did not think there would be very many instances for exceptions, but try catch statements ended up being of great use. I did not assume too much entering the project to be honest,

6. Implementation Discussion

6.1. Class Diagram

a. The operator classes created from the abstract Operator.java class was important tot he design. This allowed for better organization and thorough understanding of abstraction. The implementation of the HashMap also resulted to be very helpful in ways I did not understand before since there are methods to return values from keys. The Evaluator.java is where most of the work occurs and where a lot of the other classes are called which made the project easier to understand. The tests were also helpful.

b.



7. Project Reflection

Overall, the project was somewhat hard for me to conceptualize at first. I needed to envision the way the stack was working and ended up drawing it. I think that being given skeleton code makes me feel overwhelmed because first I have to throughly understand what is going on before I start, which has made me realize I need to practice reading/writing programs because it's very was for me to take hours to understand how to put together two abstract java concepts. I learned a lot of new things I did not know after this project, such as using a StringTokenizer, Abstract classes, Try-Catches, and the uses of a HashMap. Going through the code and adding comments, and explaining it in technical detail at the end also helped me gather my thoughts and visualize how this calculator works in greater detail.

The GUI portion was a bit foreign to me, but with the video example I was able to understand how to implement if statements to print operators and operands on my calculator. I thought this was the most fun part because seeing my results is always so satisfying and seeing what my code doing physically, like on the GUI, helped me code more easily.

8. Project Conclusion/Results

All the tests pass! This project helped me feel more confident about reading code, and getting to know my IDE better to use it to my advantage, such as debugging, searching for classes, and learning about java classes.