CSC 413 Project Documentation

Spring 2019

Calculator Project (Eval Express)

Greesan Gurumurthy

918737305

CSC413.03

Repo: https://github.com/csc413-03-spring2020/csc413-p1-Greesan

Git clone https://github.com/csc413-03-spring2020/csc413-p1-Greesan.git

Table of Contents

1 Introduction

1.1 Project Overview:

I wrote code that can calculate any integer-based expression input involving five operations, addition, subtraction, multiplication, division, power, and parentheses and provide the expression's answer, using order of operations/PEMDAS. The user interface allows a user to push buttons to input their own expression. After pressing the equals button, the input expression will be evaluated, and the answer will appear in place of the expression. The C button allows the user to clear the input expression, while the CE button allows the user to clear the last operand value in the input expression

1.2 Technical Overview

A calculator has different types of inputs: operands, operators, and clears. Operands are integers and operators are characters. During the user's input, we allow users to append their operand and operator values to a string that we set in a Java textField object, through the use of various buttons. Once the user presses the enter button, the expression is then evaluated, then the result is updated and displayed to the user through the same textField. To evaluate the input string, we parse through it, using operands and spaces as delimiters. The tokens that result from the tokenizing of the input string can be split into two buckets, an operand bucket and an operator bucket, implemented through stacks. Although these two groups give us the ability to evaluate small expressions, we must implement some way to let the computer know to perform calculation based off PEMDAS, to ensure that longer expressions are calculated properly.

In order to ensure that tokens passed in to operands were numerical values, we made a separate Operand Class. In this Operand class, we implemented checking methods that verified that an input string token was in fact an integer value, and mutator methods that stored the value of the integer into Operand objects, which we accumulate in the Operand stack, so that they can be processed.

To take care of PEMDAS, we made various operator classes: AdditionOperator, SubtractionOperator, MultiplicationOperator, DivisionOperator, PowerOperator, and ParenthesisOperator. Each of these Operators were based off an abstract parent Operator class, which statically created singular instantiations of each of these specific operator classes, and placed them in a HashMap object, so they could be easily called using strings of "+", "-", "*", "/", "^", and "(" as keys. I also implemented static methods to interact with the HashMap. Each operator child class contains an execute function that takes in two operands, performs the associated math, and returns an operand. The main exception to this is the Parenthesis Operator, as it does not require any operands to manipulate, but serves as a boundary operator to tell the computer when to stop operations due to end parentheses. As the Parenthesis Operator extends the Operator class and is therefore expected to implement the abstract parent's execute class, we bypass the compiler complaining by instantiating the execute method with a plain null return, as we do not plan on using the execute method if a parenthesis is found in the stack. They also contain priority methods that allow us to compare the priority of various operands while parsing through our operand stack, allowing us to implement PEMDAS easily. Executing operations is done by popping the operator stack once, and popping the operand stack twice, then calling the popped operator's execute function with the two popped operands as inputs. PEMDAS in particular is done by pushing higher priority operations to the operator stack and executing operations until lower priority operations can be pushed to the stack. After all operands and operators have been pushed to the respective stacks, then we execute through the rest of the operand and operator stacks

until the operator stack is null. At this point, we are left with a single value in our operand stack: the expression's simplified value.

The GUI revolves around a pretty standard calculator layout with a standard textField object at the top of the GUI that starts as a blank string, and buttons for operands and operators in the middle of the GUI. It relies on an ActionListener object to notice what buttons have been pressed. If the button pressed is a number or operator other than the equals sign, it will append the value to the TextField string. If the button pressed is the equals sign, the textField string is evaluated, and the result replaces the textField string. If the CE button is pressed, the last operand is removed from the textField string. If the C button is pressed, the whole textField string is cleared. In general, this works as a pretty dang good calculator, although it crashes for

1.3 Summary of Work Completed

This process involved understanding the code already given to us through the Evaluator class. This starter code implemented two stacks, one containing operators and another containing operands. I edited the Operand and abstract Operator classes. In the Operand class, I wrote checking methods to check whether an input is an integer, and a constructor that created an Operand object that stored an integer value. In the Operator abstract class, I implemented various static methods to manipulate a static HashMap containing keys of string operators ("+","-","*","/","^","(") and values of instances of all Operator child classes. The child classes implement the proper associated priority and execute methods inherited from the parent class. By finetuning these classes, I was able to pass the OperatorTests that were provided. In the Evaluator class, I've worked off the starter code, and written if statements and loops that ensure that operands and operators are properly processed according to PEMDAS, while also fiddling with the parenthesis operator. After finetuning this, my code was able to pass all tests. After this, I wrote some simple if statements in the EvaluatorUI class to enable the buttons to impact the textField string, and I implemented a few more methods that would help me change the textfield appropriately, based on the button pressed. In the end, I was successfully able to create the expression evaluator, and implement the GUI/Calculator version of it.

2 Development Environment

I believe I used Java SE Development Kit 11.0.6 and I used IntelliJ as my IDE.

3 How to Build/Import your Project

Pull the project from Github. Link: git clone https://github.com/csc413-03-spring2020/csc413-p1-Greesan.git

4 How to Run your Project

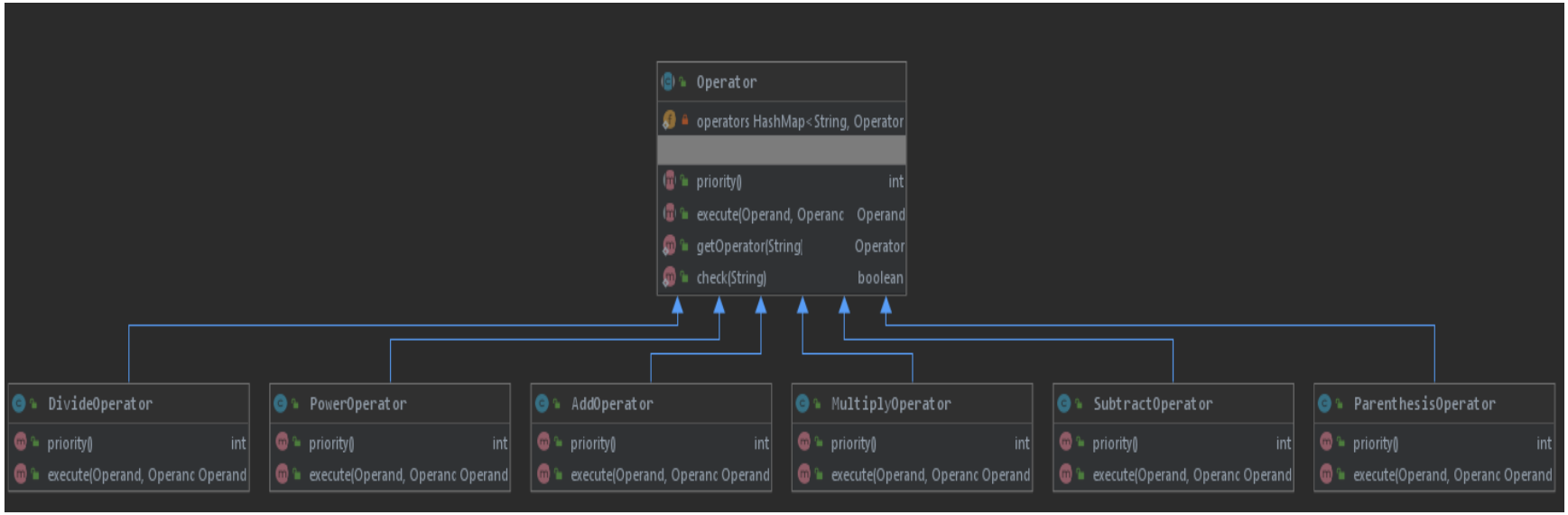
Testing with tests provided in EvaluatorTest class: select all in test, press the run button or shift+F10.
Testing the EvaluatorUI: right click on EvaluatorUI in edu.csc413→ calculator→ evaluator→EvaluatorUI

5 Assumption Made

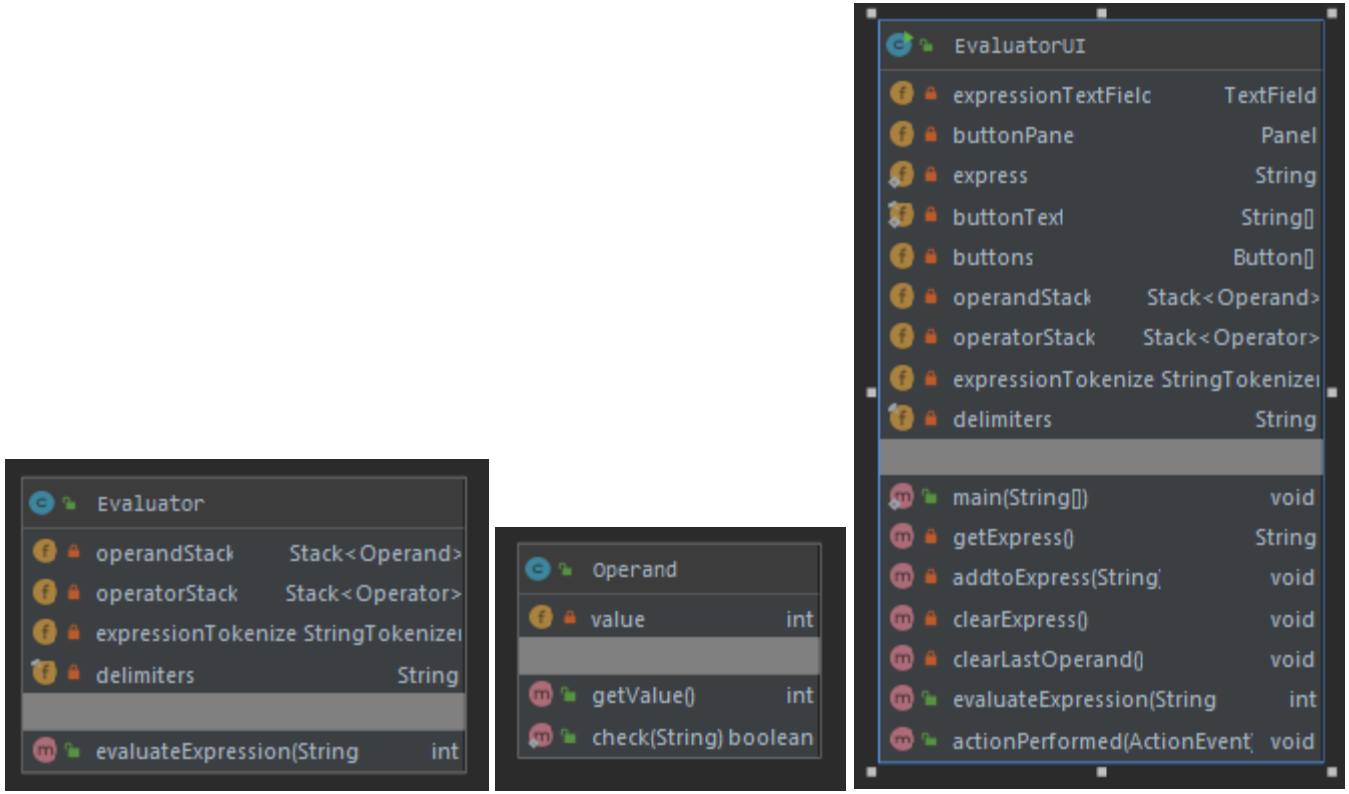
We are assuming that there are no negative inputs. We can however have negative outputs, due to the subtraction operator. We are expecting to process only integer inputs, and we expect valid expressions to be passed in. If an invalid character is passed in, we expect the program to throw an InvalidTokenException.

6 Implementation Discussion

6.1 Class Diagram



The Operator class is the abstract parent class of all the different types of operators. It also implements a HashMap with a static instantiation of each specific operator class. Each child specific operator class implements the priority and execute functions, while the ParenthesisOperator's execute function just returns null, as I do not expect to execute an operation using the Parenthesis Operator.



Most of the fields/methods for the Evaluator were already provided for us, and just required me to fill in if statements and loops accordingly. The Operand class has a Constructor that takes in a String or int value, and sets it equal to the object's value variable. The class also has a check function to check if an input string is an operand or not. The EvaluatorUI class has all the GUI elements. I particularly created the express String to hold the expression, and also wrote the getExpress, addtoExpress,clearExpress, and clearLastOperand myself. I used the previous Evaluator evaluateExpression function in the EvaluatorUI, and I added to the actionPerfomed function to properly execute the associate function based on the button clicked.

7 Project Reflection

I liked this project. I got stuck on operator test cases for like an hour and found that I was instantiating the HashMap incorrectly. At some point, I, realized I didn't understand the algorithm, so I started coming up with my own examples of expressions and running through how it logically made sense to use the stacks to properly arrange and solve the expressions. This and the website provided were both immense help in this journey. Errors with the parentheses allowed me to brush up on my debugging skills. The GUI was nice and easy though, considering the base code and GUI elements were already provided in the starter code. I feel tired, but victorious.

8 Project Conclusion/Results

This was a dope project. I was victorious in creating an expression evaluator, and calculator GUI. When I started, I truly felt like I didn't know what I was doing in this class, as the first couple of days involved tussles with abstract classes, static blocks, HashMaps, and stacks but I built my confidence back up and I feel ready as I ever will be for the next project. I am extremely happy we did not have to implement the Hashmaps and stacks from scratch, as I have had some tough experiences with that in the past. I heard the next one's a doozy, but this will definitely be fun.