CSC 413 Project Documentation

Spring 2019

David Chung

917604212

CSC 413.02

https://github.com/csc413-02-spring2019/csc413-p1-davidchungcode

Table of Contents

1 Introduction 3
1.1 Project Overview 3
1.2 Technical Overview 3
1.3 Summary of Work Completed 3
2 Development Environment 4
3 How to Build/Import your Project 4
4 How to Run your Project 4
5 Assumptions Made 5
6 Implementation Discussion 5
7 Project Reflection 7
8 Project Conclusion/Results 7

1 Introduction

1.1 Project Overview

There are three main parts of this project:

1) Taking in an expression, tokenizing it, and storing it in the appropriate stacks

2) Evaluating the information we have stored in our stacks with proper priority (PEMDAS)

3) Taking the above and creating a user interface for the user to use

1.2 Technical Overview

As mentioned in Section 1.1, this project consists of three main parts. Below are some of the ideas/concepts we used to accomplish each section. I will go further into depth on the algorithm and implementation of these concepts in Section 6.

1) StringTokenizer, Stacks

2) Operator priority to determine order to evaluate operands, Operator subclasses that contain priority and execution code for said operator, and executing expressions once priority is established

3) JFrame, ActionListener to construct our calculator and perform something whenever a button is pressed, TextField to display input/output

1.3 Summary of Work Completed

A lot of this project was provided to us, including the framework and the algorithm for how to store operands/operators as we tokenized them. This algorithm can be found at: http://csis.pace.edu/~murthy/ProgrammingProblems/16_Evaluation_of_infix_expressions.

The skeleton code of the Operator/Operand classes were given to us, but we did have to add a few things like checking whether they were Operators/Operands, returning priority of the Operator, values of the operands, creating constructors to initialize the Operand object whether we were given a String or an Int type, and a HashMap. We also had to create a subclass for each of the following operators: addition, subtraction, multiplication, division, powers, and parenthesis (open and closing). I also added an InitialOperator to make things easier. All these files can be found in the operators folder.

While the tokenizer was provided for us, we did have to implement a way to determine which stack to store them in and how to evaluate them if the operator at the top of the stack had higher priority than the one we were adding to the stacks.

We also added execute methods to our operator subclasses to tell our code how to execute said operator. The final result would be stored in our Operand stack, which we would pop and then retrieve the value of.

The last thing we did was implement a User Interface to take in expressions through buttons similar to a real life calculator and return a result.

No work was done in the EvaluatorDriver.java file, and the tests were written for us.

2 Development Environment

Version of Java used (java -version):	1.8.0_191
IDE Used:	IntelliJ IDEA Community Edition 2018.3.4

3 How to Build/Import your Project

1. To import this project, you first need to clone it from the repository. The repository can be found at: https://github.com/csc413-02-spring2019/csc413-p1-davidchungcode. Please note that this repository is private. Open Git Bash and clone it with the command "git clone https://github.com/csc413-02-spring2019/csc413-p1-davidchungcode"
2. The next thing you want to do is open your IDE of choice (IntelliJ was used to construct this project, YMMV with other IDEs, but it definitely runs in IntelliJ) and import this project. Make sure you have the source code as well as the jar files for the tests
3. To run the tests, you can right click each individual test and select "Run ____Test", or you can right click the java file and opt to run all the tests at one time with the "Run 'All' Tests" option
4. To see the User Interface we created, you want to run the "EvaluatorUI.java" file. From here a calculator interface will pop up where you can perform calculations.
5. To evaluate expressions without the use of our UI or if you wanted to enter expressions manually, you can run the "EvaluatorDriver.java" file

4 How to Run your Project

- The final culmination of this project would be the Calculator Interface, and this can be run by running the "EvaluatorUI.java" file. Please refer Section 3 in this PDF for instructions on how to import/build this project.
- If you would like to see the individual Operator files in action, there is also the option to right click the tests in the java file and run either all tests or that individual file's tests.
- To make use of our Evaluator without the use of our User Interface, you can run the "EvaluatorDriver.java" file to enter expressions manually.

5 Assumptions Made

1. We are assuming that the expression the user submits is proper, meaning each open parenthesis is accompanied by a close parenthesis and vice versa, there are no out of order parentheses, etc.
2. If we want to multiply two operands, we always use the * operator, an expression like 2(3) will NOT work
3. Users may continue calculations after evaluating their expression with correct results, but if they want a new expression completely, they will need to use the C or CE button
4. The only operators we account for are: addition, subtraction, multiplication, division, powers, and parenthesis. More complicated expressions like sin are not implemented
5. All expressions in the UI are to be inputted by clicking the buttons, editing of the expression via keyboard is disabled
6. C and CE do the same thing, clear the TextField and whole expression

6 Implementation Discussion

As mentioned in Section 1.1, this project could be divided into three main parts:

1) Taking in an expression, tokenizing it, and storing it in the appropriate stacks

2) Evaluating the information we have stored in our stacks with proper priority (PEMDAS)

3) Taking the above and creating a user interface for the user to use

In this section we will take a more in depth look at each part and break down concepts used and the logic implemented.

We start with our Operator class. This is an abstract class that we use to make our Operator subclasses. This whole class revolves around a HashMap for each of the different operators we will run into. These operators are: addition, subtraction, multiplication, division, powers, left/right parenthesis, and a space. We also have an InitialOperator subclass that isn't included in our HashMap. Our Operator class is used to verify whether or not something is an operator or not. This is done by checking if that key is in our HashMap. Our Operator class also returns the appropriate subclass when given a key. There are also two additional methods that our subclasses make use of: priority() and execute(). The former returns the priority of that operator (think PEMDAS), while the latter tells us what to do with the given operands. For example, in our AddOperator subclass, the execute method will add two parameters and return the result.

Next up is our Operand class. This class features two constructors which allow us to set the value of said Operand regardless of whether it's given a String or an Int as input. We also have a getValue() method which returns the value of that operand, and just like the Operator class, a check() method that checks if something is an operand or not.

Now that those are taken care of, we can go into how exactly we take in an expression, tokenize it, and store them. We store these operators/operands in stacks, one stack for each. Before we start

tokenizing our expression however, we will start by placing an InitialOperator onto our operatorStack. This makes things a lot easier when we are evaluating our expression.

We tokenize our expression based on a set of given delimiters, denoted by our DELIMITERS variable in line 17 of Evaluator.java. We will break up our expression into tokens until there are no more tokens. So what do we do once we have a token? I will explain how we implemented this in our code, but the full algorithm for how Parts 1 and 2 works can be found at: http://csis.pace.edu/~murthy/ProgrammingProblems/16_Evaluation_of_infix_expressions

Once we have our token, the first thing we do is see if it's an operand or an operator. If our token is an operand, it gets pushed to our operandStack. However, if it's determined to be an operator (again this is done with our check() methods), then we must do a couple things before we can add it to the operatorStack because of priority. What I mean by priority is that some operators have precedence over others. For example, if we had the expression 1 + 2 * 3, the 2 * 3 must be evaluated before the 1 can be added. Traditionally we do things from left to right, so that's why priority checking is so important in this project. Now we'll look at the steps we take before we can push an operator to the stack.

If our operatorStack is empty, then we can just push the new operator onto the stack. One important thing to note is that we can't just push the token to the stack, as it's an Operator stack. This is where the getOperator() method comes into play. This returns the appropriate operator subclass based on the token. After we push it onto the stack we have a continue, which essentially is telling our code to skip to the next iteration of that loop. The reason we have it here and in a couple other places, is that there is no reason to continue with that loop. In this case it's because at that point there will only have been 1 operand and 1 operator stored, so it makes no sense to continue into the section of code that begins to execute these operators.

The following operators (assuming there are some) now look at operator priority to decide if we just push it to the operatorStack or if we need to do some executing before we can push it. If the operatorStack is not empty and the operator's priority is higher than that of the one at the top of the stack, we can push it to the operatorStack. The reason we do this is because if it has a higher priority that means it must be done before that operator. If the operator at the top of the stack had higher priority, then we would need to execute that operator before we could push our new operator to the stack. This is why we are constantly comparing the operator at the top of the stack's priority and our new operator's priority.

The next thing we need to take into account is parenthesis. If we find an opening parenthesis '(', then we can just push it to the stack and jump to the next iteration of the loop. However, if it's a closing parenthesis ')', then we must go into the stack and execute until we find the opening parenthesis. Then we can simply pop the opening parenthesis and continue tokenizing our expression. Essentially whenever we find a parenthesis, we are basically freezing and calculating whatever's inside the parenthesis until it's just one number, and pushing that back to the stack as if everything inside the parenthesis was just an operand. Then we can continue tokenizing and executing the expression.

Once our tokenizing is complete, we have one final while loop that will run until our InitialOperator is found, which executes the rest of the expression until all that's left is one Operand in our

operandStack. Finally, we pop off this operand, get the value of it, and return this number as our result.

The last part of our project is a User Interface, which is comprised of 20 buttons, 10 digits, our 7 operators, a C/CE to clear the textField, and an = to evaluate the expression entered. In this file, all we're doing is checking the input. If it's a digit or one of the 7 operators, we just append it to the expression every time one of these buttons is pressed. If C or CE is pressed, we clear the expression and textField. Lastly, if '=' is clicked, then we pull whatever we have in the textField down, and feed it to our evaluator, and set the textField to the result.

7 Project Reflection

I enjoyed this project a lot; it was the first time I really got to work with JFrame and ActionListener. In addition, although I have worked with HashMaps in the past, it felt much more useful/necessary in this project, whereas before I feel like there were other options, but I had to use a HashMap just to learn it. I will definitely be using it more in future projects.

One thing I would have liked to implement in this project that I didn't was the fact that after you finish an expression and get your result, pressing a number doesn't clear the TextField and expression variable. You can still use more operators and it will work, but inputting numbers will just add to the expression field. So that's one thing I would have liked to implement.

8 Project Conclusion/Results

I am happy with the results of this project. I feel like I accomplished what Professor Souza (client) was asking. It would have been nice to implement the additional feature I mentioned in Section 7, but as far as I can tell, all tests were passed and the User Interface does what it's supposed to.