CSC 413 Project Documentation

Spring 2019

Brian Lai

916818167

CSC413.03

https://github.com/csc415-03-spring2019/csc413-p1-blai30

Table of Contents

1 Introduction

1.1 Project Overview

The project is a calculator that reads math expressions and calculates a result using the elementary rules of basic operations, applying the traditional order of operations.

1.2 Technical Overview

This Expression Evaluator calculates a result based on a mathematical expression given as a string input. The string is tokenized into two stacks: an operand stack for storing numbers and an operator stack for storing mathematical operators. When evaluating the expression, operands and operators are popped from the top of the stack while following the traditional order of operations to obtain the correct result.

1.3 Summary of Work Completed

Created new subclasses from the Operator class to serve as different operators that execute different calculations. Initialized a static HashMap in the Operator class that stores each operator subclass. Filled out the eval function in the Evaluator class to have it tokenize the expression into operand and operator stacks and calculate the expression based on order of operations including parentheses. Added a check for if the operator stack is not empty in order to process the stacks. By the time the evaluation completes, the operand stack will contain the result which will be returned by the eval function.

2 Development Environment

Java version: JDK 11.0.2

IDE: JetBrains IntelliJ IDEA Ultimate 2018.3.3

3 How to Build/Import your Project

Importing and building project

1. Open IntelliJ IDEA
2. Click import project
3. Navigate to the “calculator” directory
4. Click OK
5. Make sure “Create project from existing sources” is selected and click NEXT
6. Proceed to click NEXT until import is finished, making sure all JDK and resource files are detected
7. Build project using IntelliJ IDEA IDE by navigating to Build > Build Project

Building executable .jar application

1. Open project in IntelliJ IDEA and navigate to File > Project Structure
2. Choose “Artifacts” from the side panel
3. Click the blue + button to add a JAR > From modules with dependencies…
4. Select the Module dropdown and choose “main”
5. Ensure “extract to the target JAR” is selected under “JAR files from libraries” and click OK
6. Click OK to exit Project Structure and navigate to Build > Build Artifacts… and the executable .jar application should be located in calculator\out\artifacts\main_jar

4 How to Run your Project

After building the executable .jar application of the project, navigate to its directory at calculator\out\artifacts\main_jar and double click main.jar.

If running from the command-line, run this command while in the executable .jar application directory: “java -jar main.jar”. On Windows, ensure JDK 11 is installed and at the top of %PATH%.

5 Assumption Made

- If all the evaluator tests pass, my evaluator function and all associated classes are complete.
- Must use HashMap for operators.
- Must use Stack for operands and operators when evaluating expression.
- Parentheses count as operators.
- Parentheses do not calculate multiplication.
- Expression is fully evaluated when operator stack is empty.
- Spaces cannot be in DELIMITERS because the tokenizer while loop already takes care of spaces.

6 Implementation Discussion

I decided to make a separate class for each operator all in the same directory, including the parentheses operators. This puts all operators into one spot and makes it clear what operators are available. Creating a new operator is made easier by simply creating a new class that extends Operator, filling out the abstract functions, and adding a new entry to the HashMap in the Operator file. For operators that only affect one operand, such as the factorial operator, the priority will need to be 4 or higher. The Evaluator class will pick up all new operators without the need to update the code.

6.1 Class Diagram

```
calculator/
└── src/
    ├──────────────────────────────────────────┐
    └── main/java/edu/csc413/calculator/       test/java/
        ├── evaluator/                         ├── AddOperatorTest.java
        │   ├── Evaluator.java                 ├── DivideOperatorTest.java
        │   ├── EvaluatorDriver.java           ├── EvaluatorTest.java
        │   ├── EvaluatorUI.java               ├── MultiplyOperatorTest.java
        │   └── Operand.java                   ├── OperandTest.java
        └── operators/                         ├── OperatorTester.java
            ├── AddOperator.java               ├── PowerOperatorTest.java
            ├── CloseOperator.java             └── SubtractOperatorTest.java
            ├── DivideOperator.java
            ├── FactorialOperator.java
            ├── ModulusOperator.java
            ├── MultiplyOperator.java
            ├── OpenOperator.java
            ├── Operator.java
            ├── PowerOperator.java
            └── SubtractOperator.java
```

7 Project Reflection

Before starting this project, I was intimidated by the amount of code already written and the number of files there were even though it was only a handful. It turns out most of the files do not have much code in them and I was tasked with only writing a few lines to complete them. When I started working with the Evaluator class, I struggled to read the existing code, so I had to take a step back and analyze what the code was trying to do before I knew what I needed to do. I was able to figure out that once I had all the Operator classes complete, the eval function became much simpler since I was only using the public functions of the abstract superclass Operator.

Once all the Evaluator test cases passed, I decided to attempt at an extra credit opportunity and started adding new operators. The new operators include factorial and modulus. The factorial operator was a special case because it only needed one operand to evaluate so I gave it a priority of 4 and modified my Evaluator class to accommodate for it.

The EvaluatorUI class was pretty simple. Since it deals with user input, I would have to end up using “if” statements and modifying the text field. The text field serves as a string that can be used as input for the Evaluator. I went ahead and stylized the UI to give it a dark theme. I also tried using a “switch” statement but I would need to use “if” statements inside some of the cases anyway so I ended up going with the “if” statements.

8 Project Conclusion/Results

Operators with a priority of 4 or greater will only take one operand argument to evaluate.

In an attempt for extra credit, I added a factorial operator and modulus operator. I also gave the EvaluatorUI certain behaviors for clearing and typing the expression based on whether or not a result to a previous expression was displayed in the text field. The text field will always clear to “0” and never show blank.