CSC 413 Project Documentation

Spring 2019

George H. Jr. Escobar

916947881

CSC413.2

https://github.com/csc413-02-spring2019/csc413-p1-georgeescobra.git

Table of Contents

1 Introduction

1.1 Project Overview

This project generates a calculator window, where you are able to punch in numbers through clicking buttons to calculate mathematical expressions.

1.2 Technical Overview

This project uses two stacks: operator and operand. The operators accepted with this calculator are '+'(addition), '-',(subtraction) '/'(division), '^'(power), and '( )'(parentheses) operators. There are three main classes that handle the bulk of the implementation of the algorithm, an Operator Class, Operator Abstract Class, and an Operand Class. The Operator Abstract Class has 6 subclasses, a class for each operator. In addition to this there is an EvaluatorUI class that creates the Graphical User Interface and combines all the other classes in order to generate a working Calculator application.

1.3 Summary of Work Completed

Most of the Evaluator Class was already pre-written, and I only needed to implement a few logistics in terms of the infix algorithm in dealing with expressions and pushing them into the stack and evaluating them. I also had to implement the Operand Class and making the constructors, when a string or integer is passed through the constructor, and implementing an Exception if there were any non-valid arguments for the constructor. In addition to this, I had to create 6 operator Subclasses in order to implement the abstract super class Operator. In the super class, there contains a getOperator method, in order to bypass creating an object of the actual Operator class because you cannot instantiate an abstract class. And in each subclass, there contains a method for the priority of each operator and the execution method relative to the operator(If the object is an Addition Operator then this call the execute() method for Addition).

2 Development Environment

The development environment I chose to work on is the Java Development Kit 11 and the IDE used for this project is IntelliJ IDEA Ultimate.

3 How to Build/Import your Project

One of the first things that you need to do is cloning the repository into your desktop by using "$ git clone https://github.com/csc413-02-spring2019/csc413-p1-georgeescobra.git". With this you are able to open IntelliJ and import the project by finding the home directory you cloned it to. After you import it, click Create Project From Existing Sources and then next. Keep the project name and do not change it, it can ruin the whole file. IntelliJ should be able to find the source files for both the main and test files and click next. After that IntelliJ is also able to automatically find the resources and hit next. IntelliJ should also find the necessary dependencies for the test files as well. After this, IntelliJ will ask for which SDK you want to use and this should be JDK 11. After that the project should be imported properly and you're ready to build the project.

4 How to Run your Project

In order to run the file, locate the EvaluatorUI class under the main folder->java->edu.csc413.calculator->evaluator and right click and hit run, this should execute the file and have it so the GUI will run.

5 Assumption Made

The Assumptions made in this project is that the Parentheses' operator does not assume to multiply e.g. 3(9) != 3*9 instead it will throw an error. So there is an assumption that there will always be an operand right after an operator.

6 Implementation Discussion

So the way this project was implemented was by creating two Stacks, one for operands and the other for operators. The best way to explain this project is to go through an example.
Example: 2-(3/10)+2-5, the expected answer: -1
After the code: Evaluator ev = new Evaluator(); and result =ev.eval(expression); This creates an Evluator object and we are able to access the eval method through this object and we push an expression like: 2-(3/10)+2-5 as a string argument. After this, the program will parse through the string using the StringTokenizer imported class, and this will parse through the string using delimiters of, "+-^/* "(this includes an empty character or space). All the tokens will be check to see whether or not they are valid Operators/Operands. If the token is a space then it will be ignored, but if it is a character like 'a' then this will throw an exception. In the Operand class there is a check method, to see if the tokens being pushed are valid operands. Furthermore, the Operator is an abstract super class, and there are 6 subclasses that extend this superclass: AddOperator, SubtractOperator, PowerOperator, DivideOperator, MultiplicatoinOperator, and Parantheses. Starting with the '2' character this will be checked and passed through an Operand constructor to convert it from a string to an integer. The first operator in this function is '-', and this will be pushed into the operatorStack because the operatorStack is empty. The '-' goes through a check to see whether or not it is a valid operand if not then checks to see if It is a valid operand. If it is a valid operand it checks through a Static Hashmap, which contains all the valid operators as a key, and its corresponding new Object as its value. The '(' character, will be pushed because any operator will be pushed into the OperatorStack if the stack is empty or when the token is '('. Each of these classes have individual priority values, which is important for the program to know which operand to evaluate first(see Class Diagram 6.2 below). It is important to note that the algorithm we have implemented to solve these expressions is infix. The third token will be '3', and this will be converted to an integer from a string in the Operand constructor. The fourth token, '/', will also be pushed onto the OperatorStack because Paranthese.getPriority() == 0 while DivideOperator.getPriority() == 2. Then the character '10' will be checked as well and then pushed into the OperandStack. The next operator will be ')' and this one will not be pushed into the OperatorStack, because it will be caught by an if statement and will trigger a block of code that will run until the stack meets the '(' operator(see Diagram 6.3). First we can pop the OperatorStack and place it in an Operator object. As shown below, we have to pop the OperandStack and place the value into a variable op2 because this has to be evaluated as the second argument. A stack's data structure can be explained as "last one in, first one out". Then pop the OperandStack again in order

to get the second operand and place it in op1. We then execute the relative Operator and the two Operand and push its result back to the OperandStack. The result would be 0, because this calculator only takes ints! After this, we have to pop the OperatorStack again because the opening '(' is still there and since we evaluated the expression inside its' opposing closing ')' we can discard it. We are still parsing through the string, so the next token would be the '+' and this just like any other operator, it goes through a check method, then goes through a getOperator method, which returns its corresponding object (AddOperator()) through the static HashMap created in the Operator Class. This is important so we don't have to create 1000's of operators over and over just to call a function, and instead we can just allocate as much operators we need and just call it over and over by finding the correct key and getting the Object associated with the key. After it is connected to its corresponding Operator Object, and since the OperatorStack is not empty (remember there is still the '-' operator), the program will check which has the greater priority; because they both have equal priority, the '-' will go first in terms of being executed. This will be executed the same as before(see Diagram 6.3). And the value being pushed back to the OperandStack is 2, because '2-0 = 2'. Then, we can finally push the '+', operator into the OperatorStack. Everything is repeated for '2' and '-'. Since at the '-' operator there is already an operator in the OperatorStack, '+', we will evaluate the two Operands and execute the '+' Operator. The result will then be '2+2 = 4' and the result will be pushed back into the OperandStack and the '-' can finally be pushed into the OperatorStack. The final character '5', will also be pushed into the OperandStack. Now that the parsing of the string has ended, the Stacks will go through another block of code.(diagram 6.4) As shown below the Stacks will go through the process until the OperandStack reaches a size of 1, and will pop the final operand and return it. This will look like '4-5 = -1", which is the expected result.

6 Class Diagrams

6.1 NOTE: PARANTHESES HAS A PRIORITY OF 0.

The expressions are composed of integer operands and operators drawn from the set +, -, *, /, ^, (, and). These operators have the following.

Operator	Priority
+, -	1
*, /	2
^	3

6.2 How an Operator and 2 Operands are evaluated.

```
Operator oldOpr = operatorStack.pop();
Operand op2 = operandStack.pop();
Operand op1 = operandStack.pop();
operandStack.push(oldOpr.execute(op1, op2));
```

6.3 What Happens After the Control Reaches After an Expression has been parsed.

```
while (operandStack.size() > 1) {
  Operand op2 = operandStack.pop();
  Operand op1 = operandStack.pop();
  Operator op3 = operatorStack.pop();
  operandStack.push(op3.execute(op1, op2));
}
Operand done = operandStack.pop();
return done.getValue();
```

7 Project Reflection

I thought that the project was pretty straight forward, although I took way longer than expected because I haven't written in Java in over a year, and it was also my first time coding a GUI, so I had to learn exactly what the file was doing in order to finish it. Also I had to relearn about inheritance for Java because of the abstract class, but all in all it was doable. I thought that the algorithm that we had to implement was straight forward, but I messed up along the way and instead of trying to actually get the algorithm to work I tried to just solve the tests case by case. Figuring out the parentheses operators was not as hard, considering we just ignore them. I wonder what it would be like to implement it so that the calculator would take (3)9 and have the parentheses be like a multiplication operator. You would probably have to push both parentheses into the stack and check to see whether or not the next token is going to be an operand or operator is my guess. I thought this was a good refresher for Java and it showed me where I was and what I needed to work on.

8 Project Conclusion/Results

My Project Results were pretty good considering I passed all the test cases given and the expressions I manually put in through the EvalDriver, as long as it followed the assumptions made.