CSC 413 Project Documentation

Spring 2019

Christopher Eckhardt

ID#: 915736372

CSC413.02

https://github.com/csc413-02-spring2019/csc413-p1-username006

Table of Contents

1 Introduction

1.1 Project Overview

This project was our first assignment in CSC413.02. Our instructions were to make a calculator program in Java with the provided but incomplete classes. We're given a partially implemented Evaluator class, Operand class, and a complete but abstract Operator class. the EvaluatorUI class was optional and also partially implemented. Junit tests were provided to help with development.

1.2 Technical Overview

This calculator was intended to function using two stacks, one for operands (integers in this case), and another for operators (+, -, etc...). Both operators and operands would be defined by their own classes. Operators would be structured in a parent/child hierarchy due to each operator being required to perform a different task on the two operands each would be provided through function parameters.

1.3 Summary of Work Completed

My work completed includes The Operator class and all sub-classes implemented and pass their respective tests. Parenthesis, both open and close, were included along with an initial operator. Evaluator, EvaluatorUI, and Operand classes were implemented and pass their respective tests as well. The tests used were the included Junit tests.

2 Development Environment

The development environment for this project was as follows;

IDE - IntelliJ IDEA 2018.3.4 (Community Edition)

JRE - 1.8.0_152-release-1343-b26 x86_64

JVM – I used the OpenJDK 64-bit Server VM included with IntelliJ IDEA

OS – macOS Mojave, version 10.14.3 (18D109)

3 How to Build/Import your Project

As the IDE used in this class is Intellij IDEA, I will assume that will be import destination for the project files. Begin by building a "new project from existing sources", when prompted just navigate to the project source and click next. No additional building or 3rd party libraries are required.

4 How to Run your Project

the simplest way to execute the program is to navigate to the Evaluator class and locate the main method. To the left of the method signature, just before the line number should be a green

“play triangle”. Give that a click and choose to run the method. The program should start in the console below and prompt you for an expression.

5 Assumption Made

Assumptions that I made during this project were mostly about the expected input string. It may or may not include spaces, and all operands will be divided my operands other that parenthesis. Example would be 2+2 is valid, 2+(4*2) is valid, and 2(2) is invalid because a functional operator is not between the two integers.

6 Implementation Discussion

The ‘Implementation Discussion’ section will be divided into two parts. First I will discuss the less complex of the two packages, the Operator class family. The Operator class has a HashMap initialized and populated with values inside of a static block. Besides the included abstract function definitions, 3 other functions are implemented within. ‘check()’ and ‘getOperator()’ both function as intended and are implemented as shown in figure 6.1 below. ‘getKeys()’ was added to provide the Evaluator class with a correct and current list of delimiters in case future changes were to be made to the Operator family of classes.

```
47  @       public static boolean check( String token ) {
48              if(operators.containsKey(token)) {
49                  return true;
50              } else {
51                  return false;
52              }
53          }
54
55
56          public static Operator getOperator(String token) { return operators.get(token); }
59
60          //returns keys for for use in delimiting or other..
61          public static String getKeys() {
62              String keys = "";
63              List<String> list = new ArrayList<>(operators.keySet());
64              for (int i = 0; i < list.size(); i++)
65                  keys = keys + list.get(i);
66
67              // a space is added for delimiting purposes
68              keys = keys + " ";
69              return keys;
70          }
71      }
72
```

Figure 6.1

The Operator sub-classes were very straight forward, they required very small modifications for function as they should. The priority had to be changed to reflect what had been provided in the instructions. The priority was set between a range of 0 to 3, the higher the number, the higher the priority. Next was the execute() method, this was not implemented in any exotic fashion. It simply performed the operation on the values assigned to two different operands and returned a new

operand with the new value. Figure 6.2 is a simple example of this function from my AddOperator class.

```
@Override
public Operand execute(Operand op1, Operand op2) {
    return new Operand( value: op1.getValue() + op2.getValue());
}
```

Figure 6.2

Next I will focus on the Evaluator package. This contains the classes Evaluator, Evaluator Driver, EvaluatorUI, and Operand. I will start with Evaluator class because it contains the meat of the work done. At line 33 an Initial Operator is created and pushed to the stack. Once inside of the main while loop and just after the valid operator check (line 45) a new operator is created via the getOperator() method (see figure 6.3).

```
Operator newOperator = Operator.getOperator(token);

while (operatorStack.peek().priority() >= newOperator.priority()
        && !(newOperator instanceof OpenParenthesis)
        && !(newOperator instanceof CloseParenthesis)) {

  // note that when we eval the expression 1 - 2 we will
  // push the 1 then the 2 and then do the subtraction operation
  // This means that the first number to be popped is the
  // second operand, not the first operand - see the following co
  process();
}
```

Figure 6.3

After this, the body of a while loop checks and processes based on the priority of the operator. All that was added to this block was two new conditions checking for parenthesis and the actual process() function call. At the end of the main while loop, when an operator is just about to be added to the stack, one last if-statement checks if the operator is a closed parenthesis. If yes, it will call process() until an open parenthesis is popped from the operator stack. If the operator is not a closed parenthesis, then it pushes the newOperator to the stack. Once all of the tokens have been added to the two stacks the program exits the eval() methods main while loop. Now all of the operators and operands are processed until the initializing operator is the only thing remaining in the operator stack. Finally the resulting value is popped from the operand stack and returned.

In the Operand class the only function that required any actual thought was the check() method. The solution to this problem is shown in figure 6.4 below. It tries Integer.parseInt(String token), if

the token can be converted into an integer it will pass though the try/catch block, if not it will catch the exception and return false.

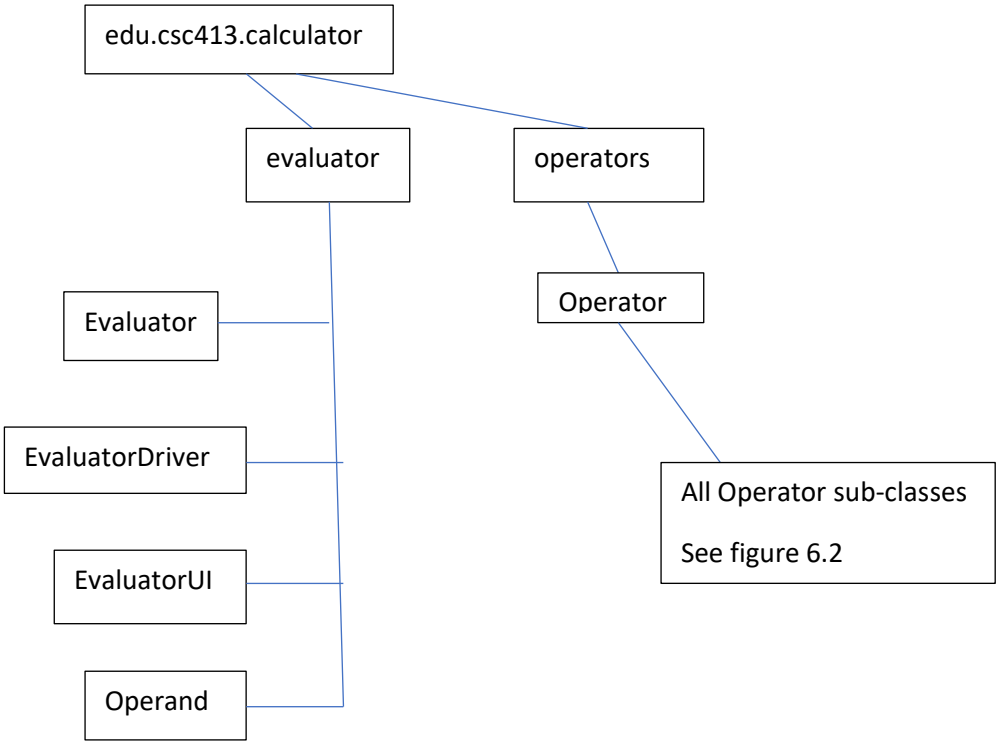
```
public static boolean check( String token ) {
  try {
    Integer.parseInt(token);
  } catch (NumberFormatException | NullPointerException nfe) {
    return false;
  }
  return true;
}
```

Figure 6.4

The last file that required modification to function was the EvaluatorUI class. All that this class needed was the actionPerformed() function block to be filled. This was a quick fix with one giant switch statement that takes e.getActionCommand() as its argument. It is ugly code and I'm sure there is a cleaner way to have done this, but it all works and so far I have not found any issues so I decided to stick with it until a more elegant solution presented itself.

6.1 Class Diagram

The Java classes are structured within the source folder as illustrated within 6.1.1.



My Operator classes are structured as implied by figure 6.1.2.

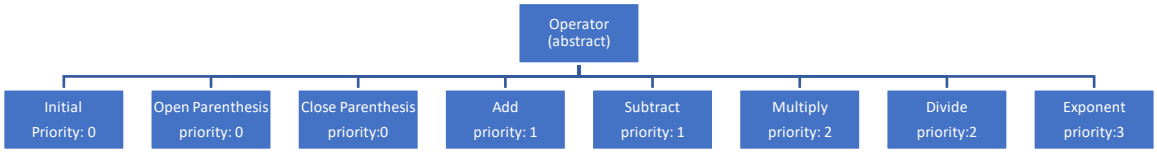


Figure 6.1.2

7 Project Reflection

I really enjoyed this project. It has been over a year since I last wrote in java, so this project was just right to knock some of the rust off. It was just difficult enough to make me struggle and have to think a little, but overall pretty manageable. Although I am not happy with some of my implementations because I'm sure there are bugs that I haven't found, Everything seems to be working fine and all of the Junit tests pass.

8 Project Conclusion/Results

I am fairly happy with the way this project turned out although there are a few changes I would have made. At the time of me writing this The program seems to function as intended. If I could do this project over again, one major change that I would make is to build multiple versions of the Evaluator class. I spent a lot of time removing and replacing code only to find that a certain solution was not working, so I would then spend time changing the code back to the way it was. I could have probably made the different eval() functions that worked and then tested each one to see what was best. I probably could have made better use branches in my git repository, but hindsight is 20/20. To summarize changes I would have made, It would be to make better use of the tools available to me.