Calculator Project Documentation (HW 1)

Spring 2019

Stephanie Sechrist

918679078

413.02

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

Table of Contents

1 Introduction

1.1 Project Overview

This project was designed to help us practice object-oriented design. I created an object (Evaluator) that calculates expressions. In addition, I was asked to create a GUI to interact with this object. Much of the code was filled out for me, and given the class hierarchy, I had to fill in the rest and pass all given tests. The calculator, in addition to evaluating according to parentheses, processes the following operations: addition, subtraction, multiplication, division, and power.

Technical Overview

1.2 Summary of Work Completed

The only file that has not been edited is EvaluatorDriver. Classes that were added implement each operator (7 child classes of Operator). These are: AddOperator, SubtractOperator, MultiplyOperator, DivideOperator, PowerOperator, OpenParenOperator, and CloseParenOperator. Each of these subclasses implements the priority() and execute(Operand op1, Operand op2) methods from the abstract methods in Operator.

Code was also added to Evaluator to: add " ", "(", and ")" to the delimiter list; consider open and close parentheses; and continue executing the expression (emptying the stack and processing operators and operands) even after the tokenizer has reached the end of the expression string.

The Operand class was filled out, implementing two constructors: one from a string token, and one from an int. Also implemented in this class is getValue(), where I get the value of the operand, and check(String token), where I check to see if the token is a valid operand.

The Operator class was also filled out. Here, I create an instance of a HashMap and initialize it with a static block of each operator subclass. I also implemented getOperator() and check(String token), which makes sure the token is a valid operator.

Lastly, in EvaluatorUI, I implemented the actionPerformed(ActionEvent arg0) method with a series of if-else statements to allow the user to use his or her mouse to enter an expression and evaluate it.

2 Development Environment

For this project, I worked solely in IntelliJ IDEA 2018.3.4 (Ultimate Edition) on Windows 10. The version of Java being used is 11.0.2.

3 How to Build/Import your Project

1. Use given git repository link to clone the project into desired folder.
2. Open IntelliJ IDEA and select Import Project in the Welcome window.
3. Find the folder that you cloned the git repository to and open the project folder (should be named csc-p1-stephlsechrist).
4. Highlight the calculator folder to be the root of the source files. Click OK.
5. Select "Create project from existing sources" and click Next.
6. Accept the defaults on the next page by clicking Next (should be naming your project calculator).
7. Accept the defaults on the next page by clicking Next.

8. Accept the defaults on the next page by clicking Next. “resources” should be selected under “Libraries.”
9. Select both “main” and “test” modules, if not already done for you. Click next.
10. Select project SDK. You should be using JDK 11. In Name field, type 11. If not already found for you, find your JDK home path. Click Next.
11. Next page should say “No frameworks detected.” Click Finish.

4 How to Run your Project

To run the project, one may either use EvaluatorDriver.java or EvaluatorUI.java (intended use).

In the Project window on the left, open folders until “evaluator” folder is found. To use the driver, right click EvaluatorDriver.java and click “Run ‘EvaluatorDriver.main()’”. Enter an expression and hit enter key to evaluate it. Manually stop the process by clicking the stop button to left of dialog box.

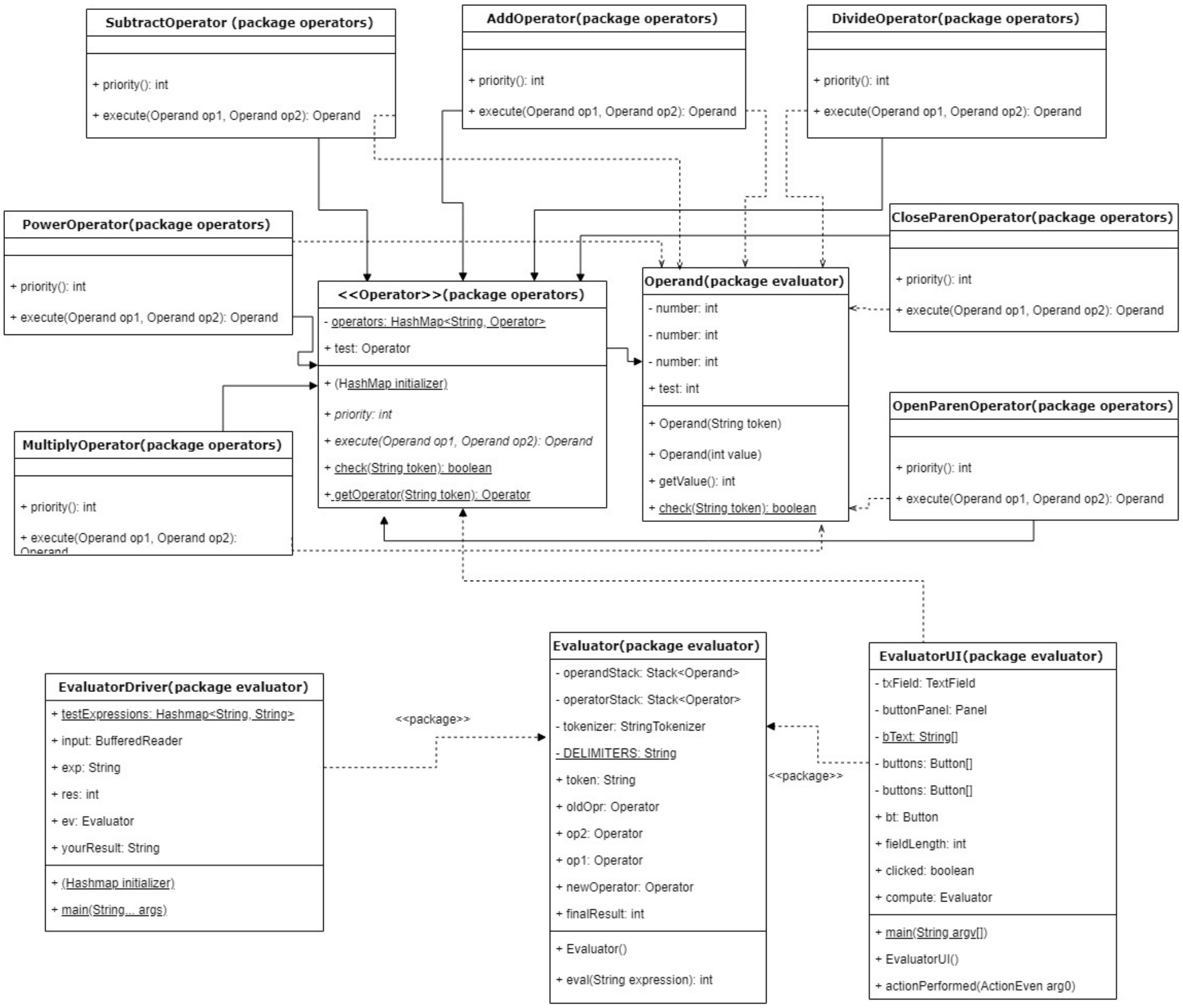
To use the GUI, right click on EvaluatorUI.java in Projects window and run it. Use your mouse to click on buttons to type an expression. Exit the window to close the program.

5 Assumption Made

In this project, I assumed that the user is going to use the GUI to interact with the program, so I do not have to worry about invalid operators, such as], [, %, !, x, etc. The calculator is a very basic calculator that only uses +, - , /, *, (, and). It is also assumed that users will not put operators next to each other, such as 3(2+1). I also assumed that I could edit any file provided to achieve the expected results. An overview of edits and additions made are in section 1.3. Lastly, in writing this documentation, I assumed the reader does not know anything about the program. However, I do not discuss the implementation of EvaluationDriver or the graphics in EvaluationUI.

6 Implementation Discussion

6.1 Class Diagram


SubtractOperator (package operators)
+ priority(): int
+ execute(Operand op1, Operand op2): Operand
AddOperator(package operators)
+ priority(): int
+ execute(Operand op1, Operand op2): Operand
DivideOperator(package operators)
+ priority(): int
+ execute(Operand op1, Operand op2): Operand
PowerOperator(package operators)
+ priority(): int
+ execute(Operand op1, Operand op2): Operand
CloseParenOperator(package operators)
+ priority(): int
+ execute(Operand op1, Operand op2): Operand
<<Operator>>(package operators)
- operators: HashMap<String, Operator>
+ test: Operator
+ (HashMap initializer)
+ priority: int
+ execute(Operand op1, Operand op2): Operand
+ check(String token): boolean
+ getOperator(String token): Operator
Operand(package evaluator)
- number: int
- number: int
- number: int
+ test: int
+ Operand(String token)
+ Operand(int value)
+ getValue(): int
+ check(String token): boolean
MultiplyOperator(package operators)
+ priority(): int
+ execute(Operand op1, Operand op2): Operand
OpenParenOperator(package operators)
+ priority(): int
+ execute(Operand op1, Operand op2): Operand
EvaluatorDriver(package evaluator)
+ testExpressions: Hashmap<String, String>
+ input: BufferedReader
+ exp: String
+ res: int
+ ev: Evaluator
+ yourResult: String
+ (Hashmap initializer)
+ main(String... args)
<<package>>
Evaluator(package evaluator)
- operandStack: Stack<Operand>
- operatorStack: Stack<Operator>
- tokenizer: StringTokenizer
- DELIMITERS: String
+ token: String
+ oldOpr: Operator
+ op2: Operator
+ op1: Operator
+ newOperator: Operator
+ finalResult: int
+ Evaluator()
+ eval(String expression): int
<<package>>
EvaluatorUI(package evaluator)
- txField: TextField
- buttonPanel: Panel
- bText: String[]
- buttons: Button[]
- buttons: Button[]
+ bt: Button
+ fieldLength: int
+ clicked: boolean
+ compute: Evaluator
+ main(String argv[])
+ EvaluatorUI()
+ actionPerformed(ActionEven arg0)

Disclaimer about UML: I have not created a UML diagram for a couple years and definitely not with these many files. It is possible I have too many details, or not enough.

6.2 Implementation Overview

The main point of this project was to create an Evaluator object that would process Operand and Operator objects to return an int to the user. The data structures used are a HashMap to store operators and stacks to keep track of to-be-processed Operators and Operands.

The overall flow of this project is as follows: the user enters an infix expression (String) into the calculator. Evaluator looks at each character (token) and pushes it to operandStack if it is a valid Operand (checked in Operand class) or to operatorStack if it is a valid Operator (checked in subclasses of Operator). The evaluator will execute operations if the operator on the top of the stack has an equal or higher priority than the one being added by popping operatorStack, popping operandStack into op2, and popping operandStack again into op1. execute(Operand op1, Operand2) is called from an Operator subclass, where the Operator is applied, returning an Operand constructed from an int, which is pushed onto the operandStack. This continues until operatorStack is empty, or until the priority of the Operator on the top of the stack is less than that being added. Priorities are as follows:

Operator	Priority
(,)	0
+, -	1
*, /	2
^	3

Parentheses are not executed like the other operators; they are used purely to determine the order of operations. This is discussed in detail below. Once the tokenizer has reached the end of the expression, if operatorStack is not empty, it will continue executing until it is empty, always pushing the result from execute onto operandStack. Once operatorStack is empty, there is one value left on operandStack; this is popped off and returned as the result from Evaluator.

6.3 Detailed Discussion

operandStack keeps track of Operands, and operatorStack keeps track of the operators. The user enters an infix expression, and the evaluator uses StringTokenizer to go through the expression, with *, /, +, - , ^, (,), and space being the delimiters. Each character is converted into a String token; this token is what I use throughout the program to check for valid Operators and Operands, as well as to get Operators from the HashMap (discussed later). As long as there are more tokens, the tokenizer checks each token in the following order: if it is a(n)

- space, it does nothing and moves to the next character.
- operand (checked with Operand.check(token)), push it to operandStack.
- operator (checked with Operator.check(token)), do another series of checks: if it is a(n)
    - open parenthesis, add it to the stack.
    - close parenthesis, then there must be an open parenthesis on the stack, so the evaluator executes until it is found. Then I pop the open off the stack and do not add the close to the stack.
    - any other operator, and the stack is not empty, I check priorities. Execute as previously described.

Once the tokenizer is done, the operatorStack is emptied as described and an int is returned from Evaluator.

In Operator, a private instance of a HashMap is created to store all the operators, initialized by a static block initializer. The operator symbol (+, - , etc.) is the key (looked up by token), and the Operator object is the value. Operator has check(String token) to make sure Operator is valid. I wanted to make an implementation that should work if other operations were added, so I used a try-catch block. It tries to execute the operator; if it works, it is valid, and if it does not, catch catches a NullPointerException and returns false for an invalid operation. I previously tried to just look for the token in the HashMap, but it would always find it for some reason, so I used execute.

Operator also has the abstract classes of priority and execute, which are implemented in the operator subclasses. Each subclass has priority(), returning an int from table above, and execute(Operand op1, Operand op2), returning an Operand object based on which operator was used. It always executes so that op1 is before op2 (op1 / op2, op1^op2, etc.).

In Operand, there are two constructors: Operand from an int, or Operand from a String. In Operand(String token), I use parseInt to turn the token into an int. I also have getValue, which just returns the value of Operand. Lastly, there is check(String token) that returns a Boolean. Here I have another try-catch block. It tries to cast the token into an int; if it can, it is valid. If it cannot, it is caught by NumberFormatException and returns false for an invalid Operand.

Finally, we have EvaluatorUI. The only code added to this was actionPerformed(ActionEvent arg0) that made the UI functional and responsive. I wanted the C button to clear the entire field. I wanted the CE button to do two things. If the last entry was a number, it will clear the number(may be multiple digits) up until the last operator. To do this, I used a for loop to go through the expression from the right until an operator was found, and I used substring to display the string up to and including the operator. If the last entry was an operator, it will clear that operator, again using substring to display the string minus one from the right. I do get a StringIndexOutOfBoundsException if the field is cleared with CE, so I have the for loop in a try-catch block. Other than that, as long as the user did not click “=”, the UI will keep displaying which buttons the user clicks. Once “=” is clicked, an Evaluator object is instantiated and evaluates the expression. I chose not to display “=” and to display only the answer after “=” is clicked.

7 Project Reflection

This was a great first project for CSC413 in my opinion. In past CS classes, I always had to work in teams. This time, I got to implement the program the way I wanted to 100% of the time. I still had to get some help from my peers, such as in the GUI, but not as much as I thought I would need to. Honestly, I had a lot of review to do, because there was a lot of moving parts. It took me a while to get started (I was a little afraid I would not be able to do it), but once I sat down and wrote down the basic structure and made sure I understood the algorithm to evaluate expressions, it was pretty fun. Also, passing all those tests in the end was a great feeling! I did learn while constructing my UML diagram that I need to do a bit more studying the different relationships classes may have (association, composition, aggregate, etc.).

8 Project Conclusion/Results

I am happy with the results of the project. There are definitely places I wish I had more time to find a better algorithm to improve efficiency. For example, another student on slack mentioned using one class for both open and close parentheses; I would like to implement that or a recursive solution if I had an extra day. Furthermore, I would fix the StringIndexOutOfBoundsException in the GUI for when CE is used to clear the field completely. Additionally, I would like to add to the UI, allowing the user to use the keyboard for input. I am also curious to see how easy it would be to add other operations, such as square root and factorial. The HashMap was presumably used to be able to add operations with relative ease, so I think it would be feasible. This project is something that can be upgraded continuously, adding more and more features.