Computer Science Department
San Francisco State University
CSC 413
Spring 2018

Assignment 1 - Expression Evaluator and Calculator GUI

Note that the due date applies to the last commit timestamp into the main branch of your repository.

Overview

The purpose of this assignment is to practice object oriented design to create two programs:

1. An object that evaluates mathematical expressions
2. A GUI around the artifact from (1)

Submission

You are required to submit a documentation PDF which is to be stored in the documentation folder in your repository, and provide your source code in the repository created in the github classroom.

The documentation must include the following sections:

* Project introduction and overview (practice concisely summarizing technical work, and provide information on execution and development environment). Include scope of work (what were you tasked with completing, what did you complete)
* Instructions to compile as jar and execute (you will be penalized if this is not provided)
* Assumptions (what assumptions did you make in order to complete the assignment)
* Implementation discussion (I strongly recommend the use of graphical artifacts to help describe your system and its implementation: class diagrams, hierarchy, etc. Implementation decisions, code organization)
* Results and conclusions (what did you learn, future work, what challenges did you encounter and how did you overcome them)

Organization and appearance of this document is critical. Please use spelling and grammar checkers - your ability to communicate about software and technology is almost as important as your ability to write software!

Requirements

You will be provided with an almost complete version of the Evaluator class (Evaluator.java). You should program the utility classes it uses - Operand and Operator - and then follow the suggestions in the code to complete the implementation of the Evaluator class. The Evaluator implements a single public method, eval, that takes a single String parameter that represents an infix mathematical expression, parses and evaluates the expression, and returns the integer result. An example expression is 2 + 3 * 4, which would be evaluated to 14.

The expressions are composed of integer operands and operators drawn from the set +, -, *, /, ^, (, and). These operators have the following precedence (# and ! will be discussed shortly, and they are extra credit).

Operator	Priority
#	0
!	1
+, -	2
*, /	3
^	4

The algorithm that is partially implemented in eval processes the tokens in the expression string using two Stacks; one for operators and one for operands (algorithm reproduced here from http://csis.pace.edu/~murthy/ProgrammingProblems/16_Evaluation_of_infix_expressions):

* If an operand token is scanned, an Operand object is created from the token, and pushed to the operand Stack
* If an operator token is scanned, and the operator Stack is empty, then an Operator object is created from the token, and pushed to the operator Stack
* If an operator token is scanned, and the operator Stack is not empty, and the operator's precedence is greater than the precedence of the Operator at the top of the Stack, then and Operator object is created from the token, and pushed to the operator Stack
* If the token is (, and Operator object is created from the token, and pushed to the operator Stack
* If the token is), the process Operators until the corresponding (is encountered. Pop the (Operator.
* If none of the above cases apply, process an Operator.

Processing an Operator means to:

* Pop the operand Stack twice (for each operand - note the order!!)
* Pop the operator Stack
* Execute the Operator with the two Operands
* Push the result onto the operand Stack

When all tokens are read, process Operators until the operator Stack is empty.

Requirement 1: Implement the above algorithm within the ***Evaluator*** class (this implementation need not be submitted, but it is strongly recommended that you begin with this version).

Requirement 2: Test this implementation with expressions that test all possible cases (you may use the included ***EvaluatorTest*** class to do this, or create JUnit tests).

The algorithm provided above has a number of inefficiencies - during each iteration of our token processing loop, we must make many comparisons. In order to improve the efficiency of this loop, we can introduce a number of optimizations. Introducing "bogus" # and ! Operators allow us to remove some comparisons from the loop, at the cost of one or two additional operations outside of the token processing loop.

Requirement 3 (EC): Refactor your implementation of the eval algorithm to remove all inefficiencies (this will be the submitted version).

Requirement 4: Implement the following class hierarchy

* Operator must be an abstract superclass.
  * boolean check(String token) - returns true if the specified token is an operator
  * abstract int priority() - returns the precedence of the operator
  * abstract Operand execute(Operand operandOne, Operand operandTwo) - performs a mathematical calculation dependent on its type
  * This class should contain a HashMap with all of the Operators stored as values, keyed by their token. An interface should be created in Operator to allow the Evaluator (or other software components in our system) to look up Operators by token.
* Individual Operator classes must be subclassed from Operator to implement each of the operations allowed in our expressions
* Operand
  * boolean check(String token) - returns true if the specified token is an operand
  * Operand(String token) - Constructor
  * Operand(int value) - Constructor
  * int getValue() - returns the integer value of this operand

Requirement 5: Reuse your Evaluator implementation in the provided GUI Calculator (EvaluatorUI.java).