Computer Science Department
San Francisco State University
CSC 413

Assignment 1 - Expression Evaluator and Calculator GUI

Warnings

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

DO NOT change the file structure of the project. CHANGING FILE/PACKAGE STRUCTURE WILL CAUSE A 20 POINT PENALTY TO BE APPLIED YO YOUR GRADE.

When importing the project into an IDE use the ***calculator*** folder as root of the source files.

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

1. Completed source submitted to your GitHub repository.
2. Completed documentation as described in the documentation guide lines file in the documentation folder in your GitHub repository.

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 priority:

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

The algorithm that is partially implemented in `eval` processes the tokens in the expression string using two `Stack`s; 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 Operator`s 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 `Operand`s
- Push the result onto the operand `Stack`

When all tokens are read, process `Operator`s until the operator `Stack` is empty.

Requirement 1: 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 the `Operator`s stored as values, keyed by their token. An interface/public method 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 sub classed 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( double value )` - Constructor
- `int getValue()` - returns the integer value of this operand

Requirement 2:

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 3:

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

Requirement 4:

Please make sure to that class members (static or not) have the correct access modifiers. You will be graded on correctly using the private, public, protected access modifiers. Class should not be ***directly*** accessing data-fields of another class.

Additional Notes.

Please use the following names for the operator classes. This will make the given unit tests work better. If not, then you must modify the unit tests.

operatornameOperator

For example:
AddOperator
DivideOperator
MultiplyOperator

PowerOperator
SubtractOperator