Calculator with a GUI

Michael Roger Swanson

Project Repository: github.com/csc413-02-sp18/csc413-p1-Swabisan

Introduction

This project is an assignment for my Software Development class at San Francisco State University, the purpose of this application is to compute arithmetic expressions programmatically with Java. There are four main class files that constitute the bulk of this program, Operator.java, Operand.java, Evaulator.java, and EvaluatorUI.java.

Instructions

To run this application, you will need Netbeans 8.2 or greater, or an equivalent IDE with Java compatibility along with a corresponding Java SE JDK 8 or better (older versions may work but are untested). If Netbeans is already set up on your compute you will need to download the project repository for the source files and import it into Netbeans. From there it is simply a matter of right clicking on the EvaluatorUI.java and selecting -> run file. Granted that this is an unwieldy way to open a calculator on your computer, the project is still technically in the development phase.

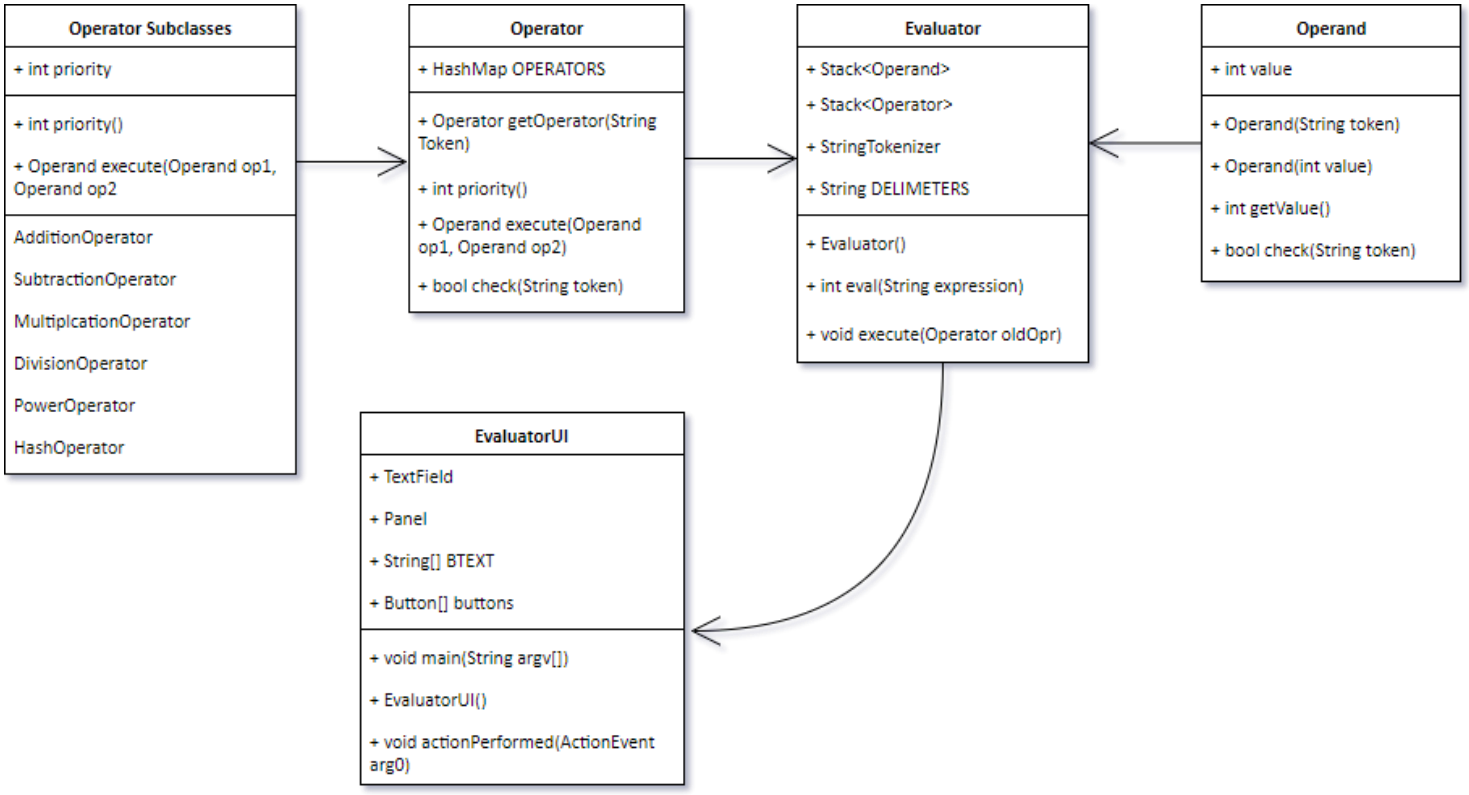
Java SE JDK 8 w/ Netbeans 8.2 Download:
http://www.oracle.com/technetwork/java/javase/downloads/jdk-netbeans-jsp-142931.html

Implementation

As discussed in the introduction, the bulk of this application resides in four classes, which can be visualized in the next page. The core idea behind the algorithm is that there are two stacks, one containing Operator and the other containing Operand objects. When a string expression is scanned, either through unit tests, inputted through the console (of which there is currently no implementation for), the String expression will be broken up through StringTokenizer, breaking the string up into single character tokens. These String tokens will be parsed through the Evaluator class which will call the Operator and Operand classes to run checks to determine the type of character that the token represents (I.e is “+” an Operator or an Operator?)

After the token is scanned and its type is determined Evaluator will run the token against a series of programmatic rules that are determined based off the type of token and the contents of Stack<Operator>. Detailed descriptions of each class will be described on page 3.

Class/ Object Structure,



Operator

HashMap OPERATORS

HashMap instance within abstract class Operator, this private static HashMap contains tokens as keys and *Operator Subclasses* as values, *Operator Subclasses* will never be called directly, all Operators lie within this HashMap

Operator getOperatoOpr(String token)

Accessor method for HashMap OPERATORS, returns Operator equivalent of token parsed, I.e "+" returns an AdditionOperator object

int priority()

Abstract, defined in *Operator Subclasses*, returns priority value

Operand execute(Operand op1, Operand op2)

Abstract, defined in *Operator Subclasses*, takes two numbers (Operands) and does a mathematical operation based on current Operator,

I.e SomeAdditionOperator.exectue(1, 2) returns 3.
Note: 1, 2, & 3 are Operand objects

boolean check(String token)

Determines if parsed token is a valid Operator symbol, i.e "K" is not an Operator

Returns true if token is an Operator, otherwise false

Operator Subclasses **actual class files differ slightly due to encapsulation*

Not just one class but a series of classes that all inherit from Operator, represents multiple forms that an Operator can take, I.e +, -, *, /, ^, (,), # Note: "#" is a pseudo Operator, never used.

int priority

Private data field representing priority of current Operator, I.e a MultplicationOperator would have higher priority than an AdditionOperator, following the rules of PEMDAS

int priority()

Accessor method for priority, returns priority

Operand execute(Operand op1, Operand op2)

Method that takes two numbers (Operands) and does a mathematical operation based on current Operator, I.e SomeAdditionOperator.exectue(1, 2) returns 3.
Note: 1, 2, & 3 are Operand objects

Operand

int value

Private data field representing the integer value of an Operand I.e Four.getValue = 4

Operand(String token) Operand(int value)

Constructors, Operand will be constructed with Integer and String values

I.e Operand("4") & Operand(4) are both valid constructors

int getValue()

Accessor method for value data field, returns value

boolean check(String token)

Method that checks if token is a valid Operand, returns true if token is a signed integer all other values should return false

Evaluator

Stack<Operand> Stack<Operator>

Evaluator contains two private stacks, both contain objects of their respective type, the purpose of this is to break up the String expression into a form that can be methodically broken down into individual operations,

I.e 4 + 2 * 3 -> can be broken down into 2* 3 -> 6, 6 + 4 -> 10

StringTokenizer -> Breaks down expression into single character Strings, parsed one at a time.

String DELIMETERS -> Constant that contains characters to be recognized for parsing.

Evaluator() -> Constructor

int eval(String expression)

Method where bulk of the algorithm resides, for algorithm details refer to implementation details on page 1. Specific cases for the algorithm are also defined in "Assignment1.pdf" inside the main repository. String expression is a mathematical expression, no "=", returns a single integer result.

I.e eval("((2 + 3) * 4)/ 2)") returns 10

void execute(Operator oldOpr)

Private method used for optimizing code to prevent repetition, in usage; remove one Operator from Stack<Operator> and two Operands from Stack<Operand> and runs the execute method from *Operator Subclass*

Results

Originally working around preexisting skeleton code in the Evaluator class threw me off while building this, you can see in the commit history of this git repository that there is a point where I delete a large portion of that class and rewrote it from scratch, I found that this actually helped in making the algorithm much more concise as I was no longer trying to bend over to conform to the edge cases provided in the skeleton code.

Future ideas for expanding on this application include adding support for unary operators like factorials and square roots.

As of writing this documentation there aren't any known bugs, but testing has been relatively minimal.

Note: There is a typo on the Class/ Object Structure visual, under **Operator***, on the method for Operator getOperator(String token), token is capitalized Token instead of token.*

Last Edited 2/15/2018