San Francisco State University
CSC 413
Dawson Zhou (zhoud@sfsu.edu)

Assignment 1: Expression Evaluator

Overview

The goal of this assignment is to apply object-oriented design in creating a simple program that evaluates arithmetic expressions. The assignment is split into two parts:

1. Implement a class hierarchy representing the components of arithmetic expressions.
2. Write a robust set of test cases for the classes and methods implemented in (1).

Submission

Assignment 1 is due on **Monday, September 28, 2020** at **11:59 pm PT**.
Late submissions will be accepted with a penalty:

- September 29, 2020 12:00 am PT to September 29, 2020 11:59 pm PT will receive 80%
- September 30, 2020 12:00 am PT to September 30, 2020 11:59 pm PT will receive 50%

Grading

100 pts total:

- 50 pts: Implementation of features
  - 3 pts: **Operator** class implemented correctly
  - 25 pts: All **Operator** subclasses implemented correctly, with 5 pts for each of the required operators (add, subtract, multiply, divide, power)
  - 12 pts: **Evaluator**'s **evaluateSimpleExpression** method implemented correctly.
  - 5 pts: **Evaluator**'s **evaluateExpression** still functions correctly (implementation is already provided)
  - 5 pts: Only **InvalidExpressionException** is thrown for invalid input.
- 50 pts: Implementation of unit tests for features
  - 5 pts: Unit tests for **Operator** class
  - 25 pts: Unit tests for all **Operator** subclasses, with 5 pts for each of the required operators (add, subtract, multiply, divide, power)
  - 10 pts: Unit tests for **Evaluator**'s **evaluateSimpleExpression**
  - 10 pts: Unit tests for **Evaluator**'s **evaluateExpression**

- 10 pts (**extra credit**): Correct implementation of negate operator and corresponding unit tests, including **Evaluator** unit tests that utilize the negate operator
    - 5 pts: Implementation of negate operator
    - 5 pts: Unit tests for negate operator

Detailed Requirements

An arithmetic expression is a sequence of operands and operators that can be evaluated to a single operand following a set of rules. Our goal in this assignment is to complete a program that can evaluate such expressions. See the lecture notes for Lecture 07 on iLearn for a basic overview of what comprises an arithmetic expression.

You will be provided with a fully implemented **Operand** class, skeleton code for the **Operator** class, and a partially implemented **Evaluator** class. You will need to:

- Implement the **Operator** class and its subclasses.
    - There must be subclasses for *add*, *subtract*, *multiply*, *divide*, and *power*.
    - You can optionally add a subclass for *negate* for extra credit.
- Implement the **Evaluator** class.
- Implement unit tests for **Operator**, all **Operator** subclasses, and **Evaluator**.

Operand and Operator

Rather than dealing with raw integers and **String** objects, we will use the **Operand** class, the **Operator** class, and a subclass of **Operator** for each specific operator that is supported (e.g. **AddOperator**, **SubtractOperator**, etc.).

The **Operand** class is relatively simple, so it is provided already implemented. It represents an integer operand in an expression, and can be constructed from a **String** or an **int**.

The **Operator** class is the abstract base class for all of the operators. Each supported operator should define its behavior in a separate subclass. Each of those subclasses must define two methods:

- **precedence** should return an integer value representing the what order operators should be evaluated in an expression with multiple operators
- **execute** takes two **Operand** parameters; it should apply the operator to those operands and return the result as a single, new **Operand**

The **Operator** class also defines a static **create** method which accepts a **String** token which indicates the operator being parsed. For example:

```
Operator operator = Operator.create("+");
```

Here, we would expect **create** to return a newly initialized **AddOperator**. The supported operators and their corresponding precedence values are:

Operator	Precedence
+ (add), - (subtract)	1
* (multiply), / (divide)	2
^ (power)	3
- (negate) (**extra credit**)	4

Larger precedence values indicate higher priority and dictate the order of operations. When an expression chains multiple operators together, the higher precedence operators should be evaluated first.

Note: because this assignment only deals with integers, when dividing two values, you should use the default integer division behavior in Java. Similarly, if you choose to implement negate to support negative operands for extra credit, raising a number to a negative power should result in the value of 0.

Evaluator

The **Evaluator** class exposes two public methods, **evaluateSimpleExpression** and **evaluateExpression**. Both methods take a single **String** parameter representing an infix arithmetic expression. **evaluateSimpleExpression** will parse the expression, evaluate the operations applying rules for order of operations, and return an integer result. For example:

```
int result = evaluator.eval("2 + 3 * 4");  // result has the value 14
```

evaluateExpression is similar, but should be able to handle expressions that have parentheses characters, which override the default order of operations rules. Operations inside parentheses should always be evaluated first. For example:

```
int result = evaluator.eval("(2 + 3) * 4");  // result has the value 20
```

The implementation of **evaluateExpression** is provided to you, but it depends on **evaluateSimpleExpression** being implemented correctly.

The high-level pseudocode for **evaluateSimpleExpression** is as follows:

```
Initialize stack for operands
Initialize stack for operators

Process every token in the expression:

    If the token is an integer operand:
        Create an operand from the token, called token_operand
        Push token_operand onto the operand stack

    If the token is an operator:
        Create an operator from the token, called token_operator
        As long as the operator stack is non-empty and the precedence of
          the operator on top of the operator stack is greater than or
          equal to the precedence of token_operator:
            Process one operator from the stack:
                Pop the top operator from the operator stack
                Pop the top two operands from the operand stack
                Apply the operator to the two operands
                  (pay attention to what order operands go into the stack
                  and what order they come out!)
                Push the result of that operation back onto
                  the operand stack

        Push token_operator onto the operator stack

Once all tokens are processed, there should be some leftover operators and
operands in the two stacks.

As long as the operator stack is non-empty:
    Process one operator from the stack (same as above)
```

Afterwards, there should be one operand left in the operand stack. Pop this value and return it as the value of the entire expression.

Some of this algorithm has already been implemented for you in **Evaluator**. Your task is to fill in the missing pieces, utilizing the **Operator** and **Operand** classes.

Errors

Your implementation of **Evaluator** should handle invalid input by throwing an **InvalidExpressionException**. If another exception ends up being thrown (e.g. if your code pops an empty stack and throws **EmptyStackException**) instead of your implementation handling the scenario, you'll be docked points.

See the provided implementation of **Evaluator**'s **evaluteExpression** method for examples of error handling.

Extra Credit: The Negate Operator

Implementing the negate operator is more challenging, so it is provided as an extra credit opportunity. A few hints to help you out:

- You will need to distinguish the difference between "-" being used as a negate operator and it being used as a subtract operator. It will help to keep track of the previous token processed in the main loop. What does the previous token look like for "-" as subtraction? What about for negation?
    - **Note**: you are allowed to change **Evaluator**'s **create** method signature (i.e. parameter list) to support the negate operator.
- Negate is a unary operator, meaning that it takes a single argument. The **Operator** base class expects all subclasses to implement the **execute** method accepting two arguments. How do you reconcile this difference? Hint: only one of the two operands needs to be used. The other can be any value if it's ignored.
    - **Note**: you are **NOT** allowed to change the method signature for **execute**. Negate *must* be implemented as part of the **Operator** class hierarchy.

Unit Tests

You will be graded on the thoroughness and quality of your unit tests. I strongly encourage you to write tests as you work through the implementation, as it will help you debug any issues that come up during implementation.

I have provided unit tests for the **Operand** class for reference. You will be responsible for providing unit tests for **Operator**, all subclasses of **Operator**, and **Evaluator**. Note that even though I provided the implementation for **evaluateExpression**, you are still expected to write unit tests for it.

Your unit tests should be:

- Thorough: cover a range of potential cases (both valid and invalid) to ensure that the class/method behave as expected
- Concise: tests should minimize extraneous setup and input that doesn't relate to what the test focuses on
- Consistent: tests should be named and structured with consistency, which helps a reader understand at a glance what the test is accomplishing
- Correct: the test code should match what the test claims it does

Additional Notes

You will be provided with the **EvaluatorUI** class. The class defines a **main** function, so it can be run as an executable. Running this program will bring up a simple calculator UI, which is hooked up to your implementation of **Evaluator**. You can use this to manually test **Evaluator** as you're working through it.

The **evaluateSimpleExpression** method should throw **InvalidExpressionException** for any invalid input provided, whether it's because the expression has an invalid character, or is badly formed. You must ensure that these cases are detected and reported with an **InvalidExpressionException**. Attempting to pop an empty stack, for example, will result in a different exception being thrown, so be sure to catch these cases.

Assignment Setup

Please refer to the handout for Assignment 0 on detailed steps for setting up a private GitHub repository, for setting up an IntelliJ project for the starter code provided on iLearn, and for steps on how to submit your assignment to GitHub.

You are expected to invite the instructor (Git name: dawsonmz) as a collaborator for the private repository you create. Your assignment will be graded based on what is submitted to GitHub.