EvaluatorTester & EvaluatorGUI

CSC 413 Spring 2016 Professor Yoon

Developer's Guide

February 12, 2016

Qihong Kuang

Table of Contents

1. Introduction

This document is a guide to explain how the developer solve the evaluator problem of CSC413. This java project contains three java files. They are Evaluator.java, EvaluatorTester.java and EvaluatorGUI.java.

Evaluator.java – a mathematical expression evaluator, which the users need to input the expression into command line.

EvaluatorGUI.java – a GUI calculator that users need to input the expression by mouse clicks.

EvaluatorTester.java – a test file that test the result of the input expression.

In project 1, Evaluator class needs to handle 5 sorts of operations – Addition, Subtraction, Multiplication, division and Power. Among the 5 operations, only power (^) operation is right-associated and the others are left-associated. Therefore, when pushing or popping operators, the Evaluator class needs to check whether the operation is left-associated or right-associated.

In addition, the EvaluatorGUI class gives the users in interface to input the expression. It looks very much like a calculator. Specially, button "C" is to clear a character from the text field and button "CE" is to clear the whole string in the text field. When user click the "=" button, the eval() function in the Evaluator class will be call and work out the result.

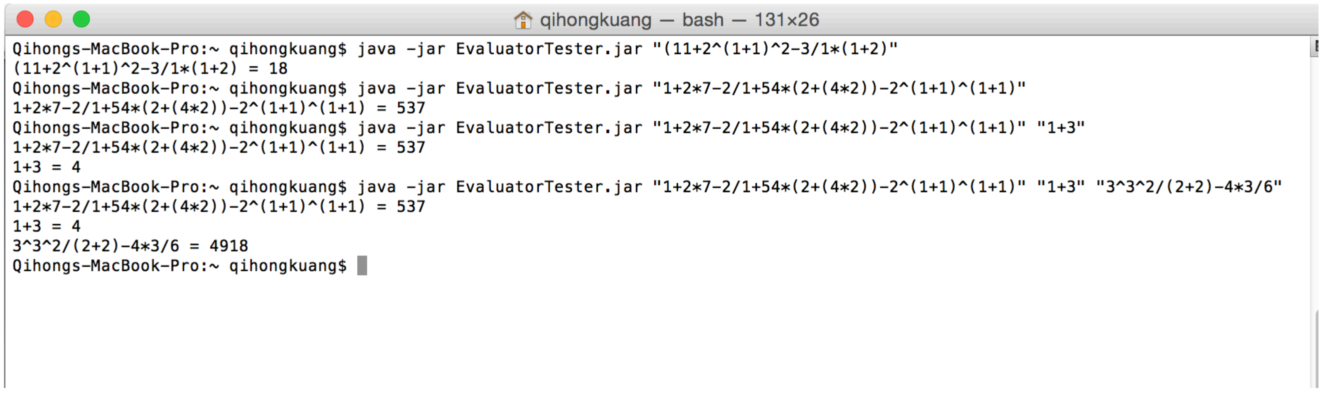
The Evaluator class can only do the calculation with integer number so the decimal part will be truncated.

2. Compile information

Two jar files are provided to the users. One is EvaluatorTester.jar, which do the command line calculation and the other one is EvaluatorGUI.jar, which generates a calculator interface for the users.

Below is my compile result for the EvaluatorTester.jar file and the EvaluatorGUI.jar file:

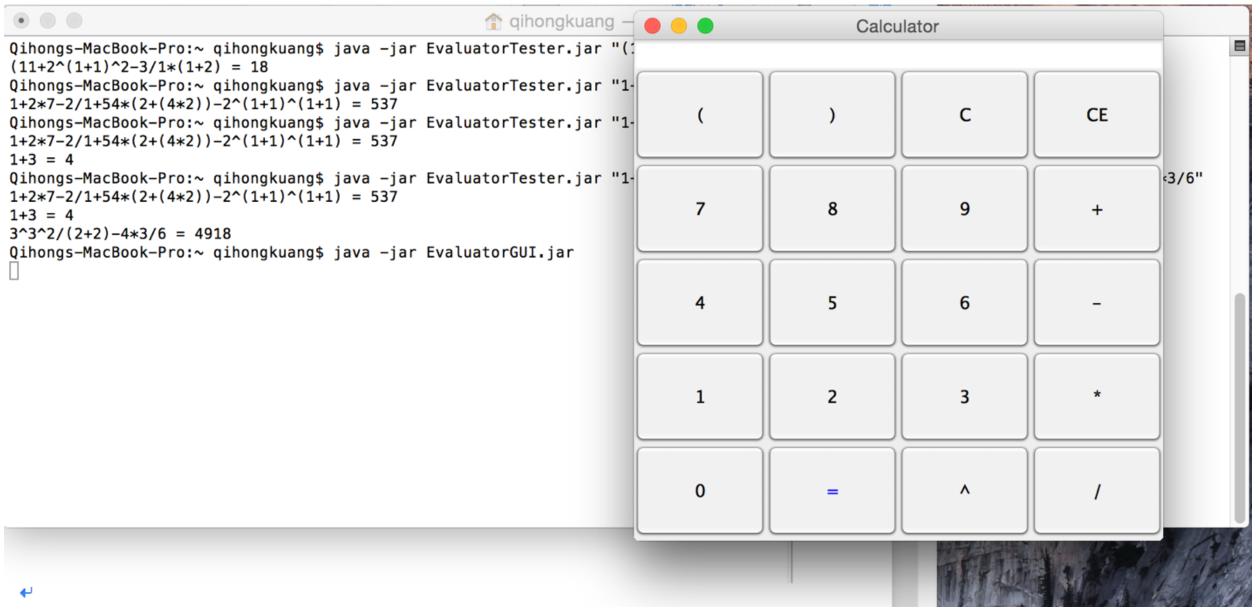
For the jar file, users need to type “java –jar THE_FILE_NAME.jar” followed by the expression



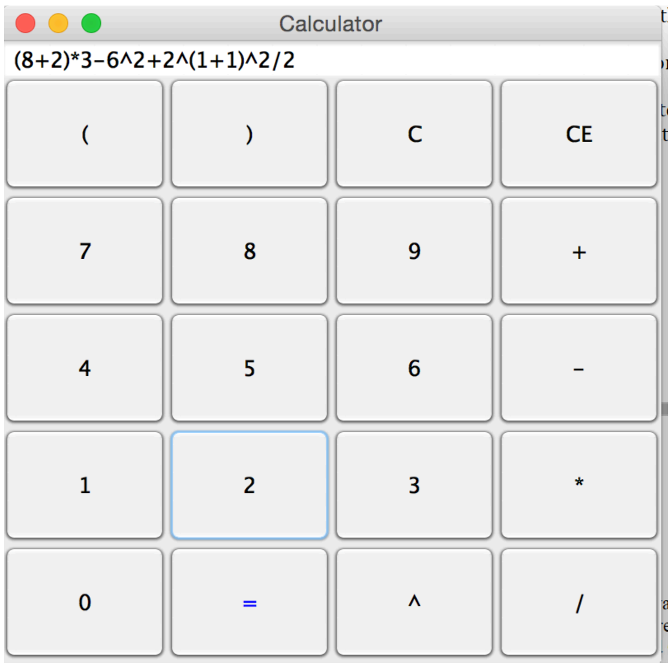
```
Qihongs-MacBook-Pro:~ qihongkuang$ java -jar EvaluatorTester.jar "(11+2^(1+1)^2-3/1*(1+2)"
(11+2^(1+1)^2-3/1*(1+2) = 18
Qihongs-MacBook-Pro:~ qihongkuang$ java -jar EvaluatorTester.jar "1+2*7-2/1+54*(2+(4*2))-2^(1+1)^(1+1)"
1+2*7-2/1+54*(2+(4*2))-2^(1+1)^(1+1) = 537
Qihongs-MacBook-Pro:~ qihongkuang$ java -jar EvaluatorTester.jar "1+2*7-2/1+54*(2+(4*2))-2^(1+1)^(1+1)" "1+3"
1+2*7-2/1+54*(2+(4*2))-2^(1+1)^(1+1) = 537
1+3 = 4
Qihongs-MacBook-Pro:~ qihongkuang$ java -jar EvaluatorTester.jar "1+2*7-2/1+54*(2+(4*2))-2^(1+1)^(1+1)" "1+3" "3^3^2/(2+2)-4*3/6"
1+2*7-2/1+54*(2+(4*2))-2^(1+1)^(1+1) = 537
1+3 = 4
3^3^2/(2+2)-4*3/6 = 4918
Qihongs-MacBook-Pro:~ qihongkuang$
```

Now the GUI part:

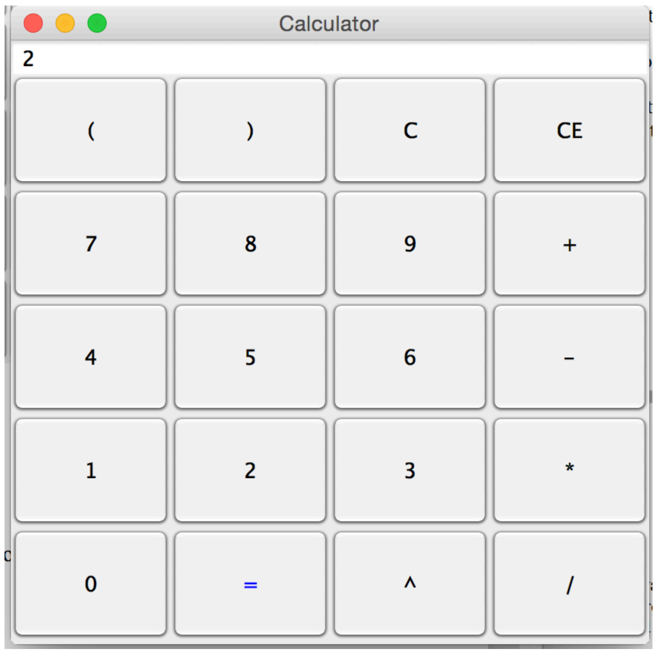
Users need to type “java –jar EvaluatorGUI.jar”. The the calculator interface will pop up.



```
Qihongs-MacBook-Pro:~ qihongkuang$ java -jar EvaluatorTester.jar "(
(11+2^(1+1)^2-3/1*(1+2) = 18
Qihongs-MacBook-Pro:~ qihongkuang$ java -jar EvaluatorTester.jar "1-
1+2*7-2/1+54*(2+(4*2))-2^(1+1)^(1+1) = 537
Qihongs-MacBook-Pro:~ qihongkuang$ java -jar EvaluatorTester.jar "1-
1+2*7-2/1+54*(2+(4*2))-2^(1+1)^(1+1) = 537
1+3 = 4
Qihongs-MacBook-Pro:~ qihongkuang$ java -jar EvaluatorTester.jar "1-                                        3/6"
1+2*7-2/1+54*(2+(4*2))-2^(1+1)^(1+1) = 537
1+3 = 4
3^3^2/(2+2)-4*3/6 = 4918
Qihongs-MacBook-Pro:~ qihongkuang$ java -jar EvaluatorGUI.jar
```



(8+2)*3-6^2+2^(1+1)^2/2

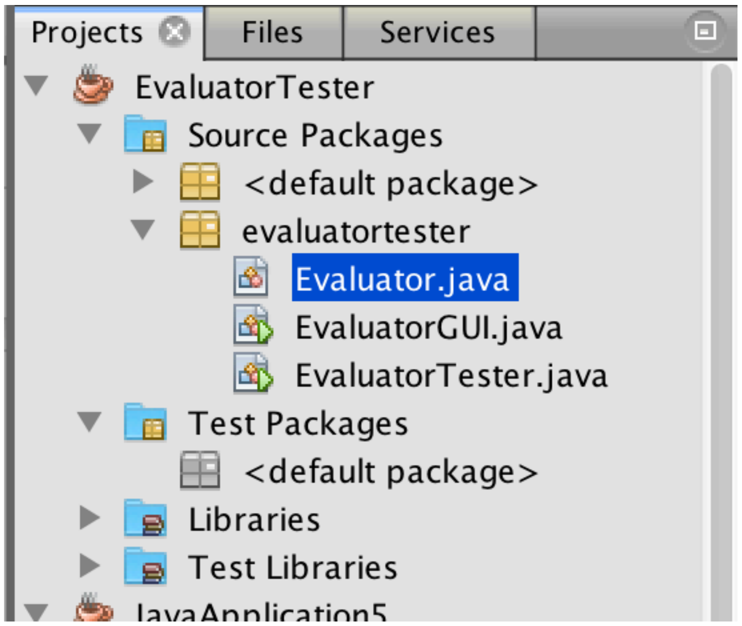


2

Instead of compiling the jar file, users can also compile the source in the terminal, NetBeans or other IDE.

3. Implementation Details

3.1 Class Overview



Both the EvaluatorGUI.java and the EvaluatorTester.java are contained in the same package and both of them use the public class Evaluator.

3.2 Evaluator.java

This file is used to token the input expression and check the left-associated operator priority and right-associated operator priority. Then, push the operands into the operand stack and operators into the operator stack by comparing the priority of each operator. And finally return the result.

Below is the hierarchy of the Evaluator.java:

public class Evaluator

 abstract class Operator

  class checkEmpOperator extends Operator ("#")

  class terminateOperator extends Operator ("!")

  class AdditionOperator extends Operator ("+")

  class SubtractionOperator extends Operator ("-")

  class mulOperator extends Operator ("*")

  class divisionOperator extends Operator ("/")

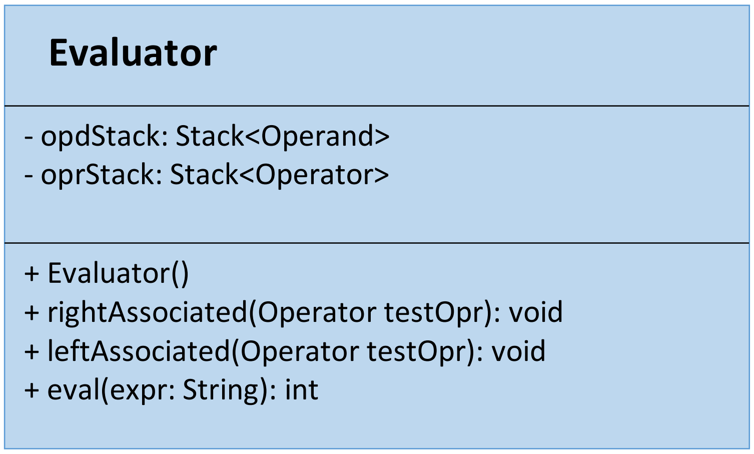
  class powOperator extends Operator ("^"

  class leftParOperator extends Operator ("(")

  class rightParOperator extends Operator (")")

 class Operand

3.3 Evaluator class



Evaluator

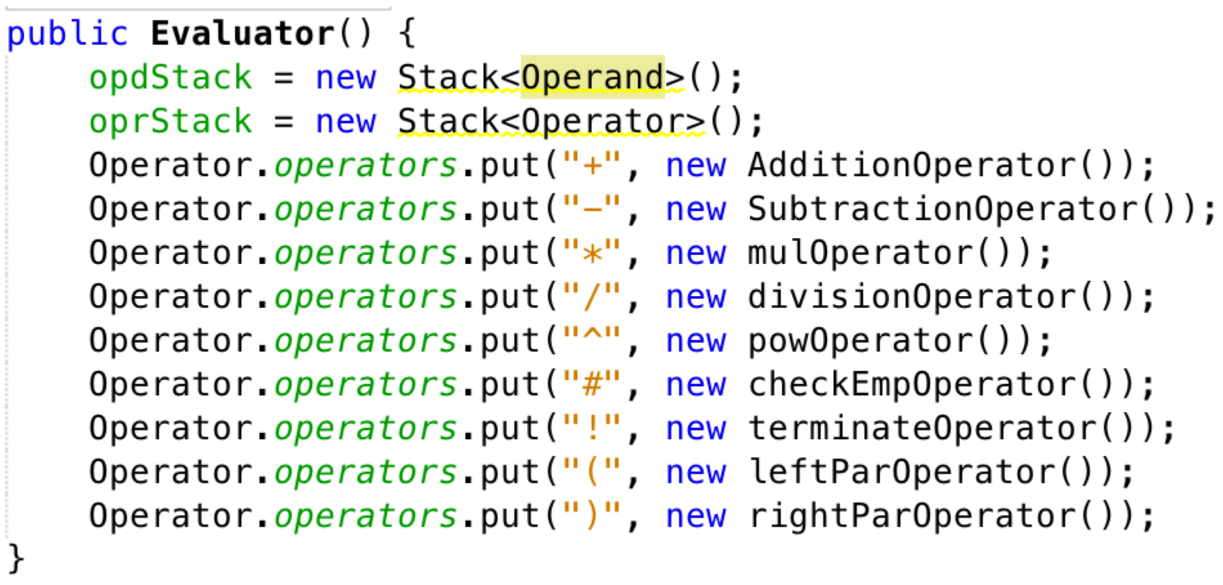
- opdStack: Stack<Operand>
- oprStack: Stack<Operator>

+ Evaluator()
+ rightAssociated(Operator testOpr): void
+ leftAssociated(Operator testOpr): void
+ eval(expr: String): int

The Evaluator constructor instantiates two Stack objects:

1. opdStack of Operand objects. The operand tokens are pushed on the stack and popped to be calculated.
2. oprStack of Operator objects. The operator tokens are pushed on the stack if they have lower priority and popped out of the stack if they have higher priority.

Also, I filled the HashMap with 9 Operator objects so that the operators can be reusable.



```
public Evaluator() {
    opdStack = new Stack<Operand>();
    oprStack = new Stack<Operator>();
    Operator.operators.put("+", new AdditionOperator());
    Operator.operators.put("-", new SubtractionOperator());
    Operator.operators.put("*", new mulOperator());
    Operator.operators.put("/", new divisionOperator());
    Operator.operators.put("^", new powOperator());
    Operator.operators.put("#", new checkEmpOperator());
    Operator.operators.put("!", new terminateOperator());
    Operator.operators.put("(", new leftParOperator());
    Operator.operators.put(")", new rightParOperator());
}
```

3.3.1 eval(String expr) method

This method is to do the calculation for the input expression. It will first filter out the operands and operators based on the delimiters given in the code. Then it checks the precedence of each operator. It will return the final result of the expression.

The eval method uses StringTokenizer to separate the input expression into operands and operators. In the method, it first checks the tokens one by one to see whether they are valid operands. And then it checks whether the tokens are valid operators.

Here, I define the power operator, left parenthesis and the right parenthesis as special case. Since "^" operator is right-associated, only when the priority of the new operators are greater than the top operator on the operator Stack.

Also, left parenthesis has different priorities. When we try to push it onto the stack, its priority is higher. But its priority is lower when we try to pop it out. The details about the operator priority will be discussed in later section.

Moreover, when handling the right parenthesis, I use an integer variable to control. After popping the left parenthesis, we need to go to the next token. So I raise the priority of the right parenthesis to highest, which is 7 so that I can push the right parenthesis (")") onto the operator stack and pop it out immediately.

Finally, the eval method will return the result of the input expression.

3.4 Operator Class

3.4.1 Operator Priority

Operator	Inpriority	Outpriority
#	0	0
!	1	1
+/-	3	3
*/ /	4	4
^	5	5
(	2	6
)	2	2

Each operator has two kinds of priority. One is the priority inside the parenthesis and the other is the one outside the parenthesis. In priority means that the priority inside the parenthesis and out priority means the priority outside the parenthesis.

We need to compare the top operator with the new operator. New operators are supposed to push into the operator stack and the top operators are supposed to popped. So we should always compare Inpriority of the old operator with Outpriority of the new operator.

3.4.2 Abstract Operator Class

Operator abstract

- operator: HashMap<String, Operator> static

+ inpriority(): int
+ outpriority(): int
- check(tok: String): boolean static
- execute(opd1: Stack<Operand>, opd2: Stack<Operator>): void abstract

The Operator class is an abstract superclass. Another 9 concrete subclasses are created for each operator type and they extend the Operator class. Also, a HashMap is initiated in the Operator abstract class.

There are three abstract methods declared in the Operator class so these methods need to be overwritten in the nine concrete subclasses. And we need to define the execution of each operator. For example, AdditionOperator method needs two operands and return the sum of opd1 and opd2. However, checkEmpOperator method only return null in execution because "#" is only used to check whether the operator stack is empty. No calculation need to be executed with "#". Likewise, the excalmatory mark "!" does no calculation.

3.5 Operand Class

Operand class is used to return each operands after parse the input String into tokens. Also it checks whether the tokens are valid operands.

4. Conclusion

1. Both the EvaluatorTester.java and EvaluatorGUI.java run properly and return correct result.
2. These two java file are still not very efficient because there are still some if statements inside the while loop
3. These two java file cannot handle the divided by zero exception and not balance exception. Proper exceptions still need to be put into the code.
4. I declared two methods to handle the left associated operators and the right associated operators. Since more right-associated operators and left-associated operators will be encountered in the future. Declaring these two methods will make the code more reusable.