Data Structures
COMP 2100
Fall 2022

Project 2: Hello, Operator?

For this project you will use stacks and queues to evaluate a numerical expression. You will do this in several phases:

1. Read an expression in infix notation
2. Print a standardized version of the parsed infix expression
3. Convert the expression to postfix and print it out
4. Evaluate the postfix expression and print out the answer

Specification

Your main() method should be in a class called InfixToPostfix. It is expected that you will create additional classes to help manage your stacks and queues.

Input

To perform input properly, you will read in operators and operands using Scanner. An infix expression has the following characteristics: It will start and end with an operand. Between each operand is an operator. The exceptions to this rule are parentheses. You will see a left parenthesis '(' when you expect to see an operand. After the left parenthesis, you will still expect an operand. You will see a right parenthesis ')' when you expect to see an operator. After a right parenthesis, you will still expect to see an operator.

Supported operators include: +, -, *, and /.

If you are expecting an operand and get something that is not a correctly formatted operand, print Invalid operand. If you are expecting an operator and get something that is not one of the supported operators, print Invalid operator: followed by the symbol in question.

It is highly recommended that you store each operator and operand in an appropriate object. One possibility is that you create two separate classes, Operator and Operand designed to hold each. They may both be subclasses of a Term superclass or interface. Alternatively, you could create a single Term class capable of holding either an operator or an operand. You will want to read the entire input into a queue that will allow you to review each token of input in order. For output purposes, supporting iteration on the queue makes a lot of sense.

You will have to perform character-by-character input to correctly read in terms. It is probably easiest to read in an entire line with your Scanner object and then cycle through the resulting String one character at a time. Operands can be any legal Java 16 double literal that is not in scientific notation, e.g. 8, +8.4,

-8.4, 0.0000000004, .51, 2. and so on. In other words, an operand may start with an optional + or -. It may optionally contain a single decimal point which has a sequence of digits 0 through 9 in front of it, behind it, or both. A legal operand must contain at least one digit. You may use the Character.isDigit() method to check to see if a char value is a digit. You should use the Double class to convert a String to a double.

If you are familiar with deterministic finite automata, constructing one can be a great aid to parsing operands.

Standardized Printing

Once you have read all the tokens into a queue or similar data structure, it is simple to print out each value with a single space between them.

For sample input:

```
(        ( -9+              8.4)      )
```

The standardized print version is:

```
( ( -9.0 + 8.4 ) )
```

Students are often tempted to keep the input in one large String. Doing so is counter-productive. It is much better to parse the data into operands and operators (and put them in a queue) when they are first read. Then, the ugly process of parsing is done once and for all. This standardized printing task is, in part, to force you to do this parsing step once and then store each operand and operator in an appropriate data structure. Doing so makes standardized printing a snap.

Conversion from Infix to Postfix

Using a stack data structure, the conversion from infix to postfix can be done with the following algorithm. The algorithm begins with a full input queue of terms, an empty stack of operators, and an empty output queue of terms.

Process the input queue, one token at a time, applying the appropriate rules depending on what kind of token you get:

- **Operand:** Add it to the output queue.
- **Left Parenthesis:** Push it onto the operator stack.
- **Right Parenthesis:** Pop off everything from the operator stack and add each operator to the output queue, until you hit a left parenthesis. Then pop off the left parenthesis.

- **Any Other Operator:** If the stack is empty, push the new operator onto the stack. If the stack is not empty, compare the precedence of the current operator with the precedence of the operator on the top of the stack. If our operator's precedence is less than or equal to the top's precedence, pop the stack and put the top element into the output queue. Continue doing so until the stack is empty or we reach an operator with a lower precedence than the new operator. Finally, put the new operator on the stack. The precedences start with (which has the lowest, then + and -, and finally * and / with the highest. The operator) does not have a precedence.

After all the input has been processed, pop whatever is left in the stack and add it to the output queue. Then print the contents of the output queue.

Evaluating the Postfix Expression

Evaluating the postfix expression requires a stack as well, but this time the stack is made up of operands. Process the postfix expression term by term. If the term is an operand, push it onto the stack. If the term is an operator, pop two operands off the stack and apply the operator to them. Then, push the result back onto the stack. When you have exhausted the input, the final answer should be the only thing left on the stack.

Errors

You are required to check for five possible errors:

1. **Invalid Operand:** If the next token of input should be an operand, but it is not a properly formatted float literal, print Invalid operand and terminate processing the current input.
2. **Invalid Operator:** If the next token of input should be an operator, but the symbol you find (ignoring spaces, of course) is an invalid operator character, print Invalid operator: and the invalid character and terminate processing the current input.
3. **Missing Operand:** If the last term in the input sequence is an operator other than the right parenthesis, it must be the case that the last operation has no second operand. In this case, print Missing operand and terminate processing the current input.
4. **Unbalanced Right Parenthesis:** If, while converting the infix expression to a postfix expression, you encounter a right parenthesis and pop operators off your stack but find no corresponding left parenthesis, print Unbalanced right parenthesis ')' and terminate processing the current input.
5. **Unbalanced Left Parenthesis:** If, at the end of the conversion from infix a postfix, when doing the final popping of all the remaining operators, you encounter a left parenthesis (which much not have had a corresponding right parenthesis) print Unbalanced left parenthesis '(' and terminate processing the current input.

Sample Output

The following are a few simple cases of sample output for you to check against. User input is marked in green.

Normal Cases

```
Enter infix expression: (5-6)*7
Standardized infix: ( 5.0 - 6.0 ) * 7.0
Postfix expression: 5.0 6.0 - 7.0 *
Answer: -7.0
```

```
Enter infix expression: 60200000000000000000000 - 14.231 *
+80000000000000000000
Standardized infix: 6.02E23 - 14.231 * 8.0E20
Postfix expression: 6.02E23 14.231 8.0E20 * -
Answer: 5.906152E23
```

```
Enter infix expression: 9
Standardized infix: 9.0
Postfix expression: 9.0
Answer: 9.0
```

```
Enter infix expression: ((((((6 - 42))))))
Standardized infix: ( ( ( ( ( ( 6.0 - 42.0 ) ) ) ) ) )
Postfix expression: 6.0 42.0 -
Answer: -36.0
```

Error Cases

```
Enter infix expression: egg + 2
Invalid operand
```

```
Enter infix expression: 4 & 5
Invalid operator: &
```

```
Enter infix expression: 10 +
Missing operand
```

```
Enter infix expression: (2 + 2
Standardized infix: ( 2.0 + 2.0
Unbalanced left parenthesis '('
```

```
Enter infix expression: 2 + 2)
Standardized infix: 2.0 + 2.0 )
Unbalanced right parenthesis ')'
```

Testing

In a problem like this, testing is of crucial importance. There are many different ways that your program can fail, and you want to check them all.

A short, but not nearly exhaustive list of test cases you want to cover includes:

- Each of the four operations separately
- Combinations of the four operations
- Use of parentheses to disambiguate
- More parentheses than are necessary
- Double values preceded by a minus sign
- Double values preceded by an optional plus sign
- Double values using scientific notation
- A single value with no operators
- Expressions with no spaces
- Expressions with unusual spacing
- Expressions long enough to require your stack or queue to perform a reallocation (assuming you are using array based stacks or queues)
- Error cases with a single bad operand
- Error cases with multiple bad operands
- Error cases with a single bad operator
- Error cases with multiple bad operators
- Error cases with unbalanced left parentheses
- Error cases with unbalanced right parentheses

For this project, you are required to create a large number of test cases, record them, and record the output of your program with each test case. A document containing this information is one of the items you must turn in.

Hints

- Start early. This project is tough. Do everything you can to work smoothly with your partner.
- This project is supposed to cover stacks and queues. You are required to implement at least one stack class. You'll need stacks for both operators and operands. You can get around the problem of coding multiple stacks by making a generic stack. Or, you can make a stack able to hold a class that can contain either an operand or an operator. Or you can make operands and operators subclasses of a common superclass and hold references to the superclass in your stack. In my solution, I used a generic stack and also made operands and operators subclasses of a superclass.
- A pure stack has a very small number of methods: push(), pop(), peek(), and maybe size() or isEmpty(). Data structures are supposed to make our lives easier. If you want to keep everything in a stack, but occasionally access the elements like an array or with an iterator, that's fine. You don't have to have a pure stack. Instead of using queues, I implemented a get() operator on my stack class. Objects were only added or removed with push() and pop(), but I could read any of them I wanted to. Additionally, feel free to implement your stack with an array or a linked list, but I highly recommend an array. It's easier, faster, and any element can be accessed in O(1) time.

- Work incrementally. Compile constantly. This project is nicely laid out into a few steps. Get input parsing working first. Then, implement your stack. Then, get infix to postfix working. Then, compute the answer. Write some kind of printing function early so that it's easy to see the current state of your terms.
- Write clean code.
- No outside classes other than Scanner, String, and the Double and Character wrapper classes should be used. That means no other Java Standard Libraries (including JCF) and no classes from Sedgewick and Wayne.
- There are lots of error cases to detect, and most of them happen while in the middle of doing something else. How can you handle all these errors? Well, exception handling is a great tool. In my implementation, I created five custom exception classes corresponding to each of the errors: InvalidOperandException, InvalidOperatorException (which takes a char value in its constructor to hold the unexpected operator), MissingOperandException, UnbalancedLeftParenthesisException, and UnbalancedRightParenthesisException. You don't have to make your own exceptions, but it makes handling the errors easy, mostly because you can handle them all in one place.

Grading

Your grade will be determined by the following categories:

Category	Weight
Parsing the infix expression and printing out the standardized version	20%
Converting the expression to postfix and printing it out	25%
Evaluating the postfix expression and printing the answer	20%
Handling error cases	15%
Quality of test cases	10%
Style and comments	10%

Note: Submissions which do not compile will automatically score zero points.

Submit!

Zip up all the .java files you use to solve this problem from your src directory. In this zip file, also include a file called tests.txt that contains all your test cases and their output.

Only the team leader should turn in the final program. I must be able to compile your program with the command javac InfixToPostfix.java and run with the command java InfixToPostfix.

All work must be done within assigned teams. You may discuss general concepts with your classmates, but it is never acceptable for you to look at another team's code. Please refer to the course **policies** if you have any questions about academic integrity. If you have trouble with the assignment, I am always available for assistance.

All work is due before Friday, Oct. 21, 2020 at 11:59 p.m.

To Submit: Zip your project and email it as an attachment to dstucki@otterbein.edu