Com S 228 Spring 2019

Project 4: Infix and Postfix Expressions (170 pts)

Due at **11:59pm**

Sunday, Apr 14

1. Problem Description

In this project, you are asked to implement the following three utilities for integer expressions using stacks:

- Postfix evaluation
- Infix to postfix conversion
- Infix evaluation

The two types (infix and postfix) of expressions are implemented by the classes `InfixExpression` and `PostfixExpression`, which extend the abstract class `Expression`. You are ***only*** allowed to use the stack implementation provided in the files `Purestack.java` and `ArrayBasedStack.java`.

You may add new instance variables and methods to the classes `Expression`, `InfixExpression` and `PostfixExpression`, but you ***cannot*** rename or remove any existing ones, or change any of them from public to private or vice versa.

2. Operands and Operators

To simplify the parsing effort, all the operands in an infix or postfix expression are either ***single lower case*** English letters (`a – z`) or ***non-negative*** integers. A single letter is treated as a variable whose value needs to be provided in the input. A ***hash map*** shall be constructed to store all variables appearing in the expression and their values. Evaluation of this expression may produce negative (intermediate) operands which are the values of subexpressions.

Parentheses '`(`' and '`)`' are allowed to appear in an infix expression but ***not*** in a postfix expression. They may be viewed as 'special operators'.

There are seven standard operators, six of which are ***binary***:

`+, -, *, /, %, ^`

In the above, '`+`' is for addition, '`-`' for subtraction, '`*`' for multiplication, '`/`' for integer division, '`%`' for the remainder operation, and '`^`' for exponentiation.

2.1 Unary Minus

The ***sole unary*** operator is unary minus, which has the same notation '`-`' as subtraction in an infix expression. A unary minus appears in the following three situations ***only***:

1. first in an expression, as in `- 3 * 5 - 7` (where the first '`-`' is a unary minus operator and the second '`-`' is a binary minus operator);

2. after another operator (either binary or unary), as in `8 - - - 10 * - 2` (where the first '`-`' is binary, and the next three '`-`'s are unary);

3. after a left parenthesis (i.e., first in a subexpression), as in `x / (- y + 2)`.

All other occurrences of '`-`' in an infix expression must be of the binary minus operator. The unary minus operator is ***right associative*** and has a higher precedence than all the binary operators. For example, the expression

```
2 * - - 5 ^ 3
```

becomes, after parenthesization,

```
2 * ((- (- 5)) ^ 3)
```

In a postfix expression, ***the unary minus operator is represented by a tilde ('~')*** because there would be no way to tell it apart from the binary minus operator if they continue to share '`-`'.

2.2 Rank and Precedence

An operand has a rank of 1, a unary operator 0, a binary operator -1, and a parenthesis 0. To be a valid infix expression, a necessary condition is that the cumulative rank in a left-to-right scan of the expression must vary between 0 and 1, and has a final value of 1.

In an infix expression, every operator has an input precedence and a stack precedence. Operator precedences and ranks are summarized in the table below.

operator	input precedence	stack precedence	rank
- (unary)	6	5	0
+ - (binary)	1	1	-1
* / %	2	2	-1
^	4	3	-1
(	7	-1	0
)	0	0	0

3. Expression Formats

In an input, operands, operators, and parentheses are separated by one or more blanks or tabs. For example, a valid infix string is

```
( x - 15 )   * 4  / 2  ^     2
```

It has value 13 if x assumes the value 28 from the input. All integers in the input are ***non-negative***. A valid postfix string is

```
8 ~ 7 * y 3 + /
```

An input infix string expects ***no*** characters other than those six for the seven unary and binary operators, the two parentheses, the ten digits, and the 26 English letters (in lower case). An input postfix string may also include one more character '~' for the unary minus operator but it does not contain parentheses. Your code need to check for possible violations.

4. Hashing of Variables

All variables from an input expression should have their values provided. These variables, each represented by one lower case English letter, have their values stored in a hash map declared as

```
private HashMap<Character, Integer> varTable;
```

The hash map is supposed to be constructed as a Java `HashMap` object in the `main()` method, and passed to the postfix or infix expression. The hash map is empty if no variables occur in the input expression.

The demo code below illustrates the use of Java `HashMap`. A hash map named `hashMap` is constructed to store the values of three single-letter variables, 'x', 'y', and 'z'. Then, the value of 'x' is retrieved.

```
class HashMapDemo {
	public static void main(String args[]) {

		HashMap<Character, Integer> hashMap = new HashMap<Character, Integer>();

		hashMap.put('x', 4);
		hashMap.put('y', 2);
		hashMap.put('z', 5);

		char c = 'x';
		if (hashMap.containsKey(c))
		{
			int x = (int) hashMap.get(c);
			System.out.println(x);
		}
	}
}
```

Check out http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html for a summary on HashMap methods, though you may only need put() and get() for the project.

5. Correctness of Expressions and Exceptions

Your code should detect whether an input infix or postfix expression is valid. A quick rejection of an invalid infix expression is when the accumulative rank goes below 0, or above 1 ***during*** a scan, or has a final value that is not 1. Because parentheses have rank 0, passing the rank test does not guarantee the infix expression to be a valid one. An incorrect infix string may also be converted into a postfix, which is later detected during the evaluation. For example, the infix string `(2 *) 3` is converted into the postfix `2 * 3` by the algorithm described in class, but evaluation of the postfix will detect a missing operand of the multiplication operator '*'.

The correctness of a postfix expression is verified during its evaluation. For instance, if a parenthesis appears in an input for a postfix expression, then that's an error.

An exception is thrown in one of the following situations:

a) An expression is found to be incorrect.

b) An illegal arithmetic operation (such as division by 0 or the zeroth power of 0) is being carried out.

c) A variable in the infix or postfix expression to be evaluated was not provided a value from the input.

Possible errors (and their corresponding exceptions) are itemized in the Javadoc comments in `infixExpression.java` and `postfixExpression.java`. So read those comments carefully.

6. Input/Output Format

Assume that an input file already meets the following requirements (***no need for checking*** in your code):

a) Every expression occupies a ***separate*** line.

b) Adjacent expressions may be separated by ***one or more blank lines***.

c) Adjacent operators and/or operands must be separated by ***one or more blanks***.

d) An infix expression starts with the letter '`I`', and follows with ***at least one blank***, and then the actual expression.

e) A postfix expression starts with the letter '`P`', and follows with ***at least one blank***, and then the actual expression.

f) If the expression contains one more variable, then their names and values are expected to be listed on the lines immediately following the expression line, with one variable and its value per line.

Below is a sample input file consisting of an infix expression and a postfix expression.

```
I ( 2 + x ) -   (  33  *  y )
  x     =   1
y    =    2

P  a   a ^  b     c  /  +
a   =  2
b     =    8
c    =   4
```

The output must leave

a) ***exact one*** blank separates two operands, two operators, or an operand and an operator.

b) ***no*** blank between a parenthesis and an operand, or between two parentheses.

7. Execution Scenario

The class `InfixPostfix` repeatedly accepts infix and postfix expressions via standard input or file input. On standard input, enter the letter '**`I`**' followed by one or more blanks before an infix expression; and the letter '**`P`**' followed by one or more blanks before a postfix expression. If the expression contains some variables, then prompts the user to enter their values one by one (see Trial 2 for an example). This may be implemented in a helper method called by `main()` in `InfixPostfix.java`.

On an input infix expression, your code first outputs it in the required format, followed by variables and values, if any from a file input, the equivalent postfix expression, and finally the calculated value of the expression. On an input postfix expression, your code outputs it, followed by variables and values, if any from a file input, and the expression value. If some variables are not provided values, the infix and postfix expressions ***must be output*** before an `UnassignedVariableException` is thrown.

Below is a sample execution scenario. Note that user keystrokes are shown in bold and a larger font.

```
Evaluation of Infix and Postfix Expressions
keys: 1 (standard input)  2 (file input)  3 (exit)
(Enter “I” before an infix expression, “P” before a postfix expression”)
```

```
Trial 1: 1
Expression: I   - ( 2 * i +  - 3 )  * 5
Infix form: - (2 * i + - 3) * 5
Postfix form: 2 i * 3 ~ + ~ 5 *
where
i = 1
Expression value: 5


Trial 2: 1
Expression: P   a  6    + b 3  ^ -
Postfix form: a 6 + b 3 ^ -
where
a = 50
b = 2
Expression value: 48


Trial 3: 2
Input from a file
Enter file name: expr.txt

Infix form: (x + y) * z
Postfix form: x y + z *
where
x = 21
y = 13
z = 5
Expression value: 170

Infix form: 8 / (1 + 3) - 6 ^ 2
Postfix form: 8 1 3 + / 6 2 ^ -
Expression value: -34

Postfix form: 2 2 +
Expression value: 4

Postfix form: 2 x ~ *
where
x = 2
Expression value: -4


Trial 4: 3
```

In trial 1, note that the capital letter '`I`' refers to the input being an infix expression while the smaller case '`i`' refers to a variable in the expression. In trial 3, the file `expr.txt` has the content below.

```
I ( x + y ) * z
```

```
x = 21
y = 13
z = 5
I 8 / ( 1 + 3 ) - 6 ^ 2
P 2 2 +
P 2 x ~ *
x = 2
```

Your code needs to print out the same text messages for user interactions.

8. Submission

Write your classes in the `edu.iastate.cs228.hw4` package. ***Turn in the zip file not your class files.*** Please follow the guideline posted under Documents & Links on Canvas. Also, follow the discussion forum Project 4 Clarifications there.

You are not required to submit any JUnit test cases. Nevertheless, you are encouraged to write JUnit tests for your code. Since these tests will not be submitted, feel free to share them with other students.

Include the Javadoc tag `@author` in each class source file. Your zip file should be named `Firstname_Lastname_HW4.zip`.