HCS3 – Binary Trees Lab — 80-, 90-, 100-point Version

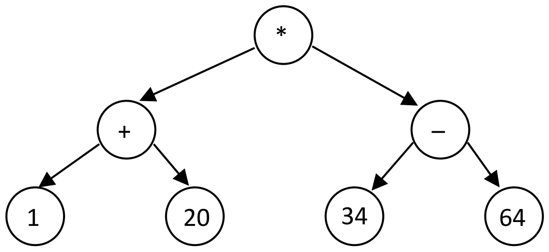
Expression Trees

Problem Description:

Although arithmetic expressions are written in linear form, they are treated as tree-like structures when evaluated. For example, when evaluating the following arithmetic expression:

((1 + 20) * (34 – 64))

The 1 and 20 are first summed, the difference between 34 and 64 is calculated, and then the two values are multiplied together. In performing these calculations, a hierarchy is implicitly formed such that the multiply operator is built upon a foundation consisting of the results of the addition and subtraction operators. This can be accomplished using a binary tree. These trees are known as expression trees.



Implement the ExpressionTree class to accept an arithmetic expression in prefix form, with no parenthesis, and blank space(s) between operators and operands. When processing the prefix form of an arithmetic expression from left to right, you will encounter each operator followed by its operands. You can use the following recursive process to construct the corresponding expression tree.

- Read the next arithmetic operator or numeric value
- If the next value is an operator (+ - * / ^ %), create a node for the operator and build its subtrees
- If the next value is an operand, create and return a new node for the operand

Applying this process to the arithmetic expression **`* + 1 20 – 34 64`** produces the tree above. The Scanner class is an excellent tool for parsing the operands and operators from the prefix expression string. A Scanner object can be made an instance variable so that multiple methods may have access. You may assume the input expression has been properly constructed.

Instead of using the TreeNode class, we will be using the ExpressionNode class to represent a node in an expression tree. The ExpressionNode utilizes an enumerated data type, NodeType, to store the kind of node (operand or operator). The public interface for each of these is shown below. Note the different constructors for operand v. operator.

```
public class ExpressionNode
ExpressionNode(int value)
ExpressionNode(NodeType operatorType, ExpressionNode left, ExpressionNode right)

NodeType getType()
int getValue()
ExpressionNode getLeft()
ExpressionNode getRight()

void setValue(int resetValue)
void setLeft(ExpressionNode left)
void setRight(ExpressionNode right)
```

```
enum NodeType
NUMBER(""),
ADD("+"),
SUBTRACT("-"),
MULTIPLY("*"),
DIVIDE("/"),
REMAINDER("%"),
EXPONENT("^");

public String getSymbol()
```

Here are some examples of utilizing the enumerated type:

```
if (str.equals(NodeType.MULTIPLY.getSymbol())
   // it's the * operator

if (myNode.getType() == NodeType.NUMBER)
   // it's an operand

new ExpressionNode(NodeType.ADD, leftC, rightC);
```

```
switch (myNode.getType())
{
   case SUBTRACT:
      return a - b;

   // etc.
}
```

```
// to iterate thru all values
for(NodeType t : NodeType.values())
{
   // etc.
}
```

Implement the ExpressionTree class as described below. You may NOT use Lambda Expressions or any other collection (like maps, sets, lists, etc.). Copy the following folder to your directory:

```
S:\Templates\Computer Science\CS 3 Honors\Labs\Unit 11 – Expression Tree Lab
```

80-point version – Implement the following:

- ExpressionTree() – creates an empty expression tree
- ExpressionTree(String prefix) – initializes the tree to be empty, then calls on setExpression(String) to build the tree with the given prefix expression
- void setExpression(String prefix) – builds the expression tree with the given prefix expression as outlined on the previous page; assume the parameter contains a valid prefix expression with one or more spaces separating the operators and operands, and no parenthesis are present. To break apart the prefix expression, you could use the Scanner class (instance variable provided) or you could use the String split method. An empty string creates an empty tree.

90-point version – Implement the following:

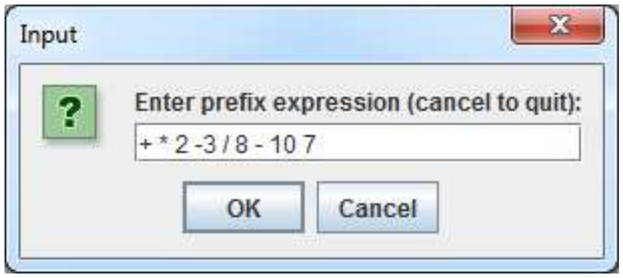
- String toString() – returns a string containing the expression using infix notation, with parenthesis around *each* subtree. An empty tree must return "0". There should be a space on both sides of each operator. For example,
  / * 4 + 3 21 37 yields the string `((4 * (3 + 21)) / 37)`

100-point version – Complete the following:

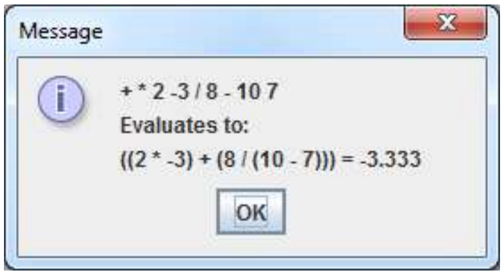
- double evaluate() – returns a real number containing the result of evaluating the expression tree using an in-order traversal

Implement private helper methods as needed. Some sample executions are shown below.

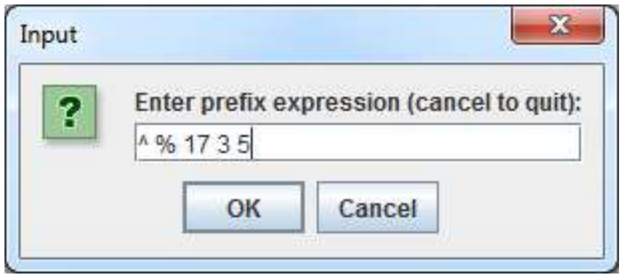
100-Point Sample Executions:



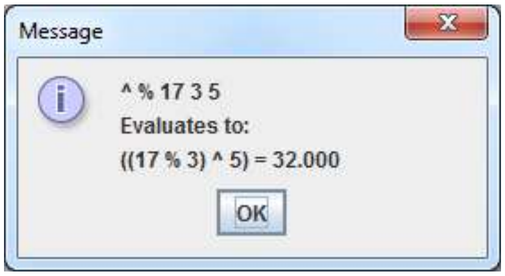
Enter prefix expression (cancel to quit):
+ * 2 -3 / 8 - 10 7



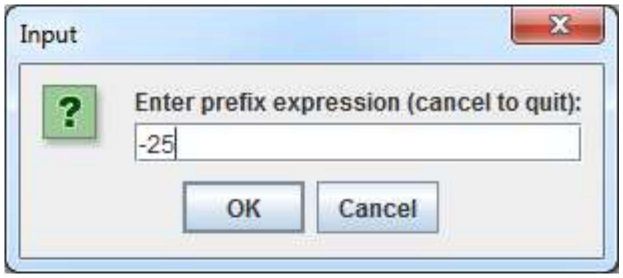
```
+ * 2 -3 / 8 - 10 7
Evaluates to:
((2 * -3) + (8 / (10 - 7))) = -3.333
```



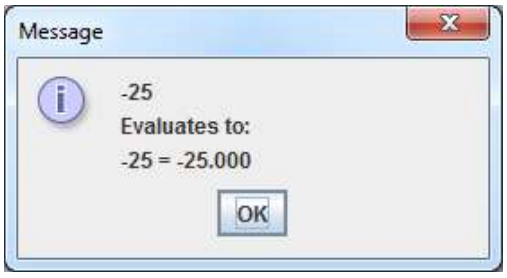
Enter prefix expression (cancel to quit):
^ % 17 3 5



```
^ % 17 3 5
Evaluates to:
((17 % 3) ^ 5) = 32.000
```



Enter prefix expression (cancel to quit):
-25



```
-25
Evaluates to:
-25 = -25.000
```