Binary Expression Tree: Prefix Expression Calculator

ExpressionTree.java

```
/**
 * Expression Tree of TreeNode objects, built from a prefix expression.
 *
 * NOTE:  Specific to Java version 5.0 --- Scanner
 *
 * The nodes are built by the recursive method "build", which calls
 * itself for internal nodes; e.g.: node.setRight ( build ( input ) );
 *
 * Beyond construction, this supports display as prefix expression,
 * postfix expression, and as parenthesized infix expression, as well
 * as evaluation of the expression, returning the value;
 *
 * @author  Timothy Rolfe
 */
import java.util.Scanner;    // Specific to Java 1.5.x

public class ExpressionTree
{
   /**
    * One node in an expression tree, allowing double values.
    *
    * @author  Timothy Rolfe
    */
   private static class TreeNode
   {
      private final boolean  leaf;   // ?Is this a leaf? else internal
      private final char     op;     // For an internal node, the operator
      private       double   value;  // For a leaf, the value
      private       TreeNode left,   // Left subexpression (internal node)
                             right;  // Right subexpression

      // Bare-bones constructor
      private TreeNode ( boolean leaf, char op, double value )
      {
         this.leaf    = leaf;
         this.op      = op;
         this.value   = value;
         this.left    = null;   // Empty to start
         this.right   = null;
      }

      // For leaf nodes, show the value; for internal, the operator.
      public String toString()// Overrides Object.toString, must be public.
      {  return leaf ? Double.toString(value) : Character.toString(op);  }
   }

   TreeNode root = null;

   public ExpressionTree ( Scanner input )
   {  root = build(input);  }

/**
 * Based on a white-space delimited prefix expression, build the
 * corresponding binary expression tree.
 * @param  input  The scanner with the expression
 * @return reference to the corresponding binary expression tree
 */
   private TreeNode build ( Scanner input )
   {
      boolean  leaf;
      String   token;
      double   value;
      TreeNode node;

      leaf = input.hasNextDouble();
      if ( leaf )
      {
         value = input.nextDouble();
         node = new TreeNode ( leaf, '\0', value );
      }
      else
      {
         token = input.next();
         node  = new TreeNode ( leaf, token.charAt(0), 0.0 );
         node.left  = build ( input );
         node.right = build ( input );
      }
      return node;
   }

/**
 * Show the expression tree as a postfix expression.
 * All the work is done in the private recursive method.
 */
   public void showPostFix ()
   {
      showPostFix ( root );
      System.out.println();
   }

   // Postfix expression is the result of a post-order traversal
   private void showPostFix ( TreeNode node )
   {
      if ( node != null )
      {
         showPostFix ( node.left );
         showPostFix ( node.right );
```

```
         System.out.print ( node + " " );
      }
   }

/**
 * Show the expression tree as a prefix expression.
 * All the work is done in the private recursive method.
 */
   public void showPreFix ()
   {
      showPreFix ( root );
      System.out.println();
   }

   // Prefix expression is the result of a pre-order traversal
   private void showPreFix ( TreeNode node )
   {  // NOTE:  removing tail recursion
      while ( node != null )
      {
         System.out.print ( node + " " );
         showPreFix ( node.left );
         node = node.right;  // Update parameter for right traversal
      }
   }

/**
 * Show the expression tree as a parenthesized infix expression.
 * All the work is done in the private recursive method.
 */
   public void showInFix ()
   {
      showInFix ( root );
      System.out.println();
   }

   // Parenthesized infix requires parentheses in both the
   // pre-order and post-order positions, plus the node
   // itself in the in-order position.
   private void showInFix ( TreeNode node )
   {
      if ( node != null )
      {
         // Note:  do NOT parenthesize leaf nodes
         if ( ! node.leaf )
            System.out.print ("( ");        // Pre-order position
         showInFix ( node.left );
         System.out.print ( node + " " );   // In-order position
         showInFix ( node.right );
         if ( ! node.leaf )                 // Post-order position
            System.out.print (") ");
      }
   }
```

```
/**
 * Evaluate the expression and return its value.
 * All the work is done in the private recursive method.
 * @return  the value of the expression tree.
 */
   public double evaluate ()
   {  return root == null ? 0.0 : evaluate ( root ) ;  }

   // Evaluate the expression:  for internal nodes, this amounts
   // to a post-order traversal, in which the processing is doing
   // the actual arithmetic.  For leaf nodes, it is simply the
   // value of the node.
   private double evaluate ( TreeNode node )
   {
      double result;    // Value to be returned

      if ( node.leaf )        // Just get the value of the leaf
         result = node.value;
      else
      {
         // We've got work to do, evaluating the expression
         double left, right;
         char   operator = node.op;

         // Capture the values of the left and right subexpressions
         left  = evaluate ( node.left );
         right = evaluate ( node.right );

         // Do the arithmetic, based on the operator
         switch ( operator )
         {
            case '-':  result = left - right;  break;
            case '*':  result = left * right;  break;
            case '/':  result = left / right;  break;
            case '^':  result = Math.pow (left, right );  break;
         // NOTE:  allow fall-through from default to case '+'
            default:   System.out.println ("Unrecognized operator " +
                          operator + " treated as +.");
            case '+':  result = left + right;  break;
         }
      }
      // Return either the leaf's value or the one we just calculated.
      return result;
   }
}
```

PrefixCalc.java

```
/**
 * Prefix calculator:  generate the expression tree, then display it
 * in the various supported means and finally show the result of the
 * calculation.
 *
 * NOTE:  Specific to Java version 5.0 --- Scanner
 *
 * @author  Timothy Rolfe
 */
import java.util.Scanner;

public class PrefixCalc
{
   public static void main ( String[] args )
   {
      ExpressionTree calc;

   // Allow for a command-line argument (which would be double-quoted).
      if ( args.length > 0 )
      {
         System.out.println ("Processing string " + args[0]);

         calc = new ExpressionTree(new Scanner(args[0]));
      }
      else
      {
         System.out.println
           ( "Prefix expression, with all elements separated by blanks");

         calc = new ExpressionTree(new Scanner(console.nextLine()));
      }

      System.out.println ("\nInput as prefix expression:");
      calc.showPreFix();

      System.out.println ("\nInput as postfix expression:");
      calc.showPostFix();

      System.out.println ("\nInput as parenthesized infix expression:");
      calc.showInFix();

      System.out.println ("\nValue:  " + calc.evaluate());
   }
}
```

Specimen Runs

```
Prefix expression, with all elements separated by blanks
+ * ^ 9 0.5 2 / 5 2

Input as prefix expression:
+ * ^ 9.0 0.5 2.0 / 5.0 2.0

Input as postfix expression:
9.0 0.5 ^ 2.0 * 5.0 2.0 / +

Input as parenthesized infix expression:
( ( ( 9.0 ^ 0.5 ) * 2.0 ) + ( 5.0 / 2.0 ) )

Value:  8.5
```

Command-line argument example:
java PrefixCalc "^ ^ ^ 2 2 2 2"

```
Processing string  ^ ^ ^ 2 2 2 2

Input as prefix expression:
^ ^ ^ 2.0 2.0 2.0 2.0

Input as postfix expression:
2.0 2.0 ^ 2.0 ^ 2.0 ^

Input as parenthesized infix expression:
( ( ( 2.0 ^ 2.0 ) ^ 2.0 ) ^ 2.0 )

Value:  256.0
```

```
Prefix expression, with all elements separated by blanks
^ 2 ^ 2 ^ 2 2

Input as prefix expression:
^ 2.0 ^ 2.0 ^ 2.0 2.0

Input as postfix expression:
2.0 2.0 2.0 2.0 ^ ^ ^

Input as parenthesized infix expression:
( 2.0 ^ ( 2.0 ^ ( 2.0 ^ 2.0 ) ) )

Value:  65536.0
```