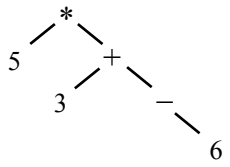
Representing expressions as trees in Java

We show how easy it is to write classes to implement expressions like 5 * (3 + -6) in Java as trees, providing methods to evaluate an expression and to return its preorder, inorder, and postorder forms. OO together with recursion keeps it all short and simple. On the JavaHyperText entry for *trees*, you can download the Java code for expression trees that you see here.



Interface ExpTree

To keep the discussion simple, we implement only expression trees for **int** expressions.

We start with interface `ExpTree`. It requires that any implementing class provide the four basic methods described in the paragraph above.

The spec of method `inorder` requires that binary operands be parenthesized in order to avoid ambiguity and mistakes. For example, the tree to the right above represents the expression 5 * (3 + -6) and *not* the expression 5 * 3 + -6 because mathematical convention requires that 5 * 3 + -6 be evaluated as (5 * 3) + -6.

```
/** An instance is an expression tree. */
public interface ExpTree {
   /** Return the value of this tree. */
   int eval();

   /** Return the preorder of this tree. */
   String preorder();

   /** Return the postorder of this tree. */
   String postorder();

   /** Return the inorder of this tree,
       with binary ops parenthesized. */
   String inorder();
}
```

Class IntLeaf

The simplest class to write is `Value`, which represents an integer constant, or literal. An object of this class will be a leaf of an expression tree.

To save space in writing `Value` to the right, we have omitted most of the comments, since they are obvious. We have also omitted annotation `@Override`, which should be on every method except the constructor

Method `toString` will appear in every node of an expression tree. It will return the inorder of that node.

Finally, note that every method body in `Value` consists of one simple statement.

```
public class Value implements ExpTree {
   private int v; // The value of this leaf

   /** Constr: an instance with value v. */
   public IntLeaf(int v) { this.v= v; }

   public int eval() { return v; }

   public String preorder() { return "" + v; }

   public String postorder() { return "" + v; }

   public String inorder() { return "" + v; }

   public String toString() { return inorder(); }
}
```

Class BinaryOp

An object of class `BinaryOp` represents a binary operator. It needs three fields, for the operator and its two operands. In the partial listing of class `BinaryOp` that appears to the right, the operator will be one of the characters +, -, *, and /.

The class has a constructor, which has as parameters the operator and its two operands.

The other methods of class `BinaryOp` are given at the top of the next page. The longest one, eval, has to determine which operator is in the object and calculate accordingly. Later, we talk about what to do if there are many more operators, like equality, relational operators, boolean operators, and so forth.

```
public class BinaryOp implements ExpTree {
   private String op; // The operator: +, -, *, or /
   private ExpTree leftExp; // The left operand
   private ExpTree riteExp; // The right operand

   /** Constructor: left operand left, operator op,
    *               and right operand right. */
   public BinaryOp(ExpTree left, String op,
                                 ExpTree right) {
      leftExp= left;
      this.op= op;
      riteExp= right;
   }
```

The bodies of the rest of the methods contain only one statement. Writing recursive methods to calculate preorder, inorder, and postorder is easy! Note that method `inorder` places parentheses around the operation, as required

We leave it to you to write class `UnaryOp`, which should represents expressions – exp and + exp.

Methods of class BinaryOp

```
public int eval() {
    if (op.equals("+")) return leftExp.eval() + riteExp.eval();
    if (op.equals("-")) return leftExp.eval() - riteExp.eval();
    if (op.equals("*")) return leftExp.eval() * riteExp.eval();
     return leftExp.eval() / riteExp.eval();
}

public String preorder() { return op + " " + leftExp.preorder() + " " + riteExp.preorder(); }

public String postorder() { return leftExp.postorder() + " " + riteExp.postorder() + " " + op; }

public String inorder() { return "(" + leftExp.inorder() + " " + op + " " + riteExp.inorder() + ")"; }

public String toString() { return inorder(); }
```

What else could you do?

Here are ways that you could extend this set of classes for arithmetic expressions.

1. Add **int** variables. You will need a second kind of leaf, say `Variable`, which contains a variable. It could be initialized in its constructor. After you get that implemented, consider adding a method to assign a value to the variable.
2. Add an additional type, **boolean**. The problem with making this addition is that two different function `eval` are necessary z —one returns an **int**, the other a **boolean**. You could consider doing it this way. It will take some time, but if you haven't refactored and haven't massaged classes this way before, it's a good learning experience.
   a. Delete method `eval` from interface `ExpTree`.
   b. Add two more interfaces, `IntExpTree` and `BoolExpTree`. They should extend interface `ExpTree`. Add to each method `eval`, which in `IntExpTree` will return an **int** and in `BoolExpTree` will return a **boolean**.
   c. Use Eclipse's refactoring tool to change the names of classes `Value`, `UnaryOp`, and `BinaryOp` to `IntValue`, `IntUnaryOp`, and `IntBinaryOp`. Also have them implement `IntExpTree` instead of `ExpTree`.

   If these steps have been done correctly, all previous tests should now work.

   d. Now create classes `BoolValue`, `BoolUnaryOp` (the operation is ! (not)), and `BoolBinaryOp` (the operations are && and ||). They should implement interface `BoolExpTree`. Rather than write them from scratch, it may be simpler to copy and paste the corresponding **int** classes and then modify them. Create appropriate JUnit test cases and test thoroughly.
   e. Now you can add a class `BinaryRelation`, which implements relations ==, <= <, <=, >, and >= with int values. Each such relation produces a boolean value, so the class should implement `BoolExpTree`.
   f. Finally, consider adding a class `Conditional`, which implements a conditional expression

      (<bool-exp> ? <int-exp> : <int-exp>)

      Since it produces an int value, it should extend `IntExpTree`. You get to figure out what inorder, preorder, and postorder mean for such a tree.