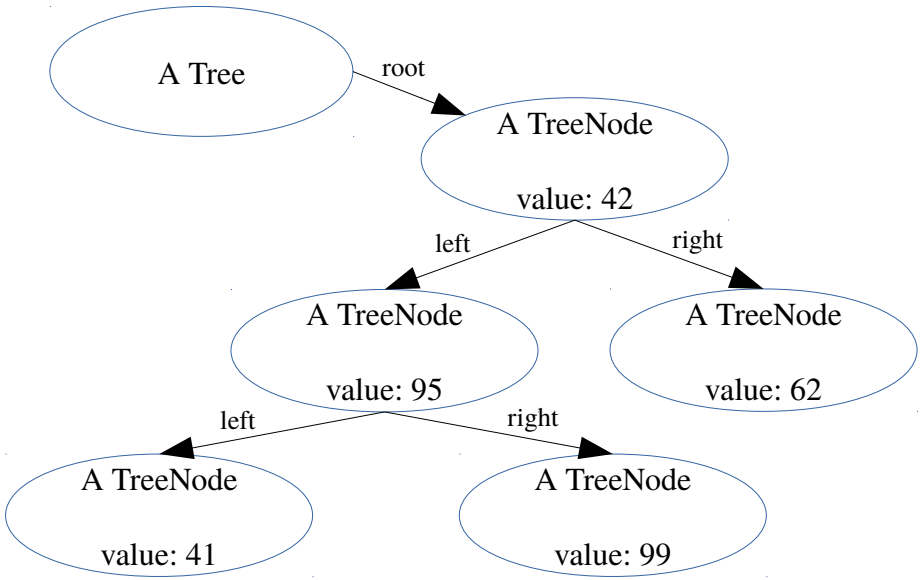
Object-Oriented Applications (CM2201)
Lab 1

The purpose of this lab is to look at a particular situation in which it is possibly better to replace a single class with a number of more specialised classes, implementing type-specific behaviour. The task was introduced in a lecture preceding the lab. class.

A *tree* is a data structure made up of *nodes.* Some nodes have *sub-nodes;* those which do not are called *leaf nodes.* One of the nodes is the *root node.* Typically one might represent a tree using objects like this:



In the above example:

- the *root* is the node with value 42
- the nodes with values 41, 99 and 62 are the *leaf* nodes
- the nodes with values 95 and 62 are the *left* and *right subnodes* of the node with value 42
- the nodes with values 41 and 99 are the left and right subnodes of the node with value 95

A *full binary tree* is a tree where every node is either a leaf node or it has precisely two sub-nodes. So there are no nodes with just a left sub-node and not a right sub-node, for example. The above is an example of a full binary tree.

One way of *traversing* a tree is to perform an *inorder* traversal:

- start at the root
- "traverse" this node

To "traverse" a node, a recursive procedure (one which calls itself) is followed. For a binary tree it can be described thus:

- traverse the left sub-node, if any
- "visit" the node itself
- traverse the right sub-node, if any

The following code implements the above inorder traversal in Java, for a full binary tree. "Visit" is interpreted as meaning "print out the value the node holds" for this particular example. The code can be downloaded from "Lab 1: banishing IF!".

Test.java

```
public class Test {
  public static void main(String args[]) {
    TreeNode r=new TreeNode(3);
    r.setLeftAndRight(new TreeNode(4), new TreeNode(5));

    Tree t=new Tree(r);
    t.traverse();
  }
}
```

Tree.java

```
public class Tree {
  private TreeNode root;
  public void traverse() {root.traverse();}

  public Tree(TreeNode r) {
    root=r;
  }
}
```

TreeNode.java

```
public class TreeNode {
  private int value;
  private TreeNode left;
  private TreeNode right;

  public TreeNode(int val) {
    value=val;
    left=null;
    right=null;
  }
```

[continued overleaf]

```
  public void setValue(int aValue) {
    value=aValue;
  }
  public void setLeftAndRight(TreeNode leftNode, TreeNode
                                                        rightNode) {
    left=leftNode;
    right=rightNode;
  }

  public int getValue() {
    return value;
  }

  public void traverse() {
    if (left!=null) {
      left.traverse();
    }

    System.out.println(value);

    if (right!=null) {
      right.traverse();
    }
  }
}
```

In this implementation, we are using a single class for all the nodes. An alternative is to define an *interface* **TreeNode** and then provide two *implementations* – **LeafNode** and **NonLeafNode**. The traversal method for **LeafNode** would be different from the method for **NonLeafNode** in that it would not attempt to visit sub-nodes.

Your task is to modify the supplied program in this way. You could remove the "getter and setter" methods if you wish; they are not necessarily required for a solution. Your revised program should be capable of being tested using the following revised **Test** class:

```
public class Test {

  public static void main(String args[]) {
    TreeNode r=new NonLeafNode(3, new LeafNode(4), new
LeafNode(5));

    Tree t=new Tree(r);
    t.traverse();
  }
}
```

Now think about whether what you have done has really improved the program …

A suggested outline solution will appear in the "Lab 1: banishing IF!" folder after the last of the Week 4 supervised lab sessions has finished.

Andrew Jones. Last modified 18th October 2017