```
 1 package decaf;
 2
 3 import java.util.Stack;
 4 import java.util.Hashtable;
 5
 6 /**
 7  * Implements the symbol table data abstraction.
 8  *
 9  * <p>
10  * <p>
11  * In addition to strings, compilers must also determine and manage
   the
12  * scope of program names.  A symbol table is a data structure for
13  * managing scope.  Conceptually, a symbol table is just another
   lookup
14  * table.  The key is the symbol (the name) and the result is
   whatever
15  * information has been associated with that symbol (e.g., the
   symbol's
16  * type, line that it occurs, etc.).
17  *
18  * <p>
19  * <p>
20  * In addition to adding and removing symbols, symbol tables also
21  * support operations for entering and exiting scopes and for
   checking
22  * whether an identifier is already defined in the current scope.
   The
23  * lookup operation must also observe the scoping rules of the
   language;
24  * if there are multiple definitions of identifier <code>x</code>,
   the
25  * scoping rules determine which definition a lookup of <code>x</
   code>
26  * returns.  In most languages, including Decaf, inner definitions
   hide
27  * outer definitions.  Thus, a lookup on <code>x</code> returns the
28  * definition of <code>x</code> from the innermost scope with a
29  * definition of <code>x</code>.
30  *
31  * <p>
32  * <p>
33  * This example symbol table is implemented using Java hashtables.
   Each
34  * hashtable represents a scope and associates a symbol with some
35  * data. The ``data'' is whatever data the programmer wishes to
36  * associate with each identifier.
```

```
37  */
38 class SymbolTable {
39     private Stack tbl;
40 
41     /**
42      * Creates an empty symbol table.
43      */
44     public SymbolTable() {
45         tbl = new Stack();
46     }
47 
48     /**
49      * Enters a new scope. A scope must be entered before anything
50      * can be added to the table.
51      */
52     public void enterScope() {
53         tbl.push(new Hashtable());
54     }
55 
56     /**
57      * Exits the most recently entered scope.
58      */
59     public void exitScope() {
60         if (tbl.empty()) {
61             System.err.println("Error --> existScope: can't remove
   scope from an empty symbol table.");
62         }
63         tbl.pop();
64     }
65 
66     /**
67      * Adds a new entry to the symbol table.
68      *
69      * @param id   the name
70      * @param info the data asosciated with id
71      */
72     public void addId(String name, Object info) {
73         if (tbl.empty()) {
74             System.err.println("Error --> addId: can't add a symbol
   without a scope.");
75         }
76         ((Hashtable) tbl.peek()).put(name, info);
77     }
78 
79     /**
80      * Looks up an item through all scopes of the symbol table.  If
81      * found it returns the associated information field, if not it
```

```
 82      * returns <code>null</code>.
 83      *
 84      * @param name the symbol
 85      * @return the info associated with name, or null if not found
 86      */
 87     public Object lookup(String name) {
 88         if (tbl.empty()) {
 89             System.err.println("Error --> lookup: no scope in
    symbol table.");
 90         }
 91         // I break the abstraction here a bit by knowing that stack
     is
 92         // really a vector...
 93         for (int i = tbl.size() - 1; i >= 0; i--) {
 94             Object info = ((Hashtable) tbl.elementAt(i)).get(name);
 95             if (info != null) return info;
 96         }
 97         return null;
 98     }
 99
100     /**
101      * Probes the symbol table.  Check the top scope (only) for the
102      * symbol <code>name</code>.  If found, return the information
    field.
103      * If not return <code>null</code>.
104      *
105      * @param name the symbol
106      * @return the info associated with name, or null if not found
107      */
108     public Object probe(String name) {
109         if (tbl.empty()) {
110             System.err.println("Error --> lookup: no scope in
    symbol table.");
111         }
112         return ((Hashtable) tbl.peek()).get(name);
113     }
114
115     /**
116      * Gets the string representation of the symbol table.
117      *
118      * @return the string rep
119      */
120     public String toString() {
121         String res = "";
122         // I break the abstraction here a bit by knowing that stack
     is
123         // really a vector...
```

```
124         for (int i = tbl.size() - 1, j = 0; i >= 0; i--, j++) {
125             res += "Scope " + j + ": " + tbl.elementAt(i) + "\n";
126         }
127         return res;
128     }
129 }
130
```