```
package decaf;

import java.util.Stack;
import java.util.Hashtable;

/** Implements the symbol table data abstraction.
 *
 * <p>
 *
 * In addition to strings, compilers must also determine and manage the
 * scope of program names.  A symbol table is a data structure for
 * managing scope.  Conceptually, a symbol table is just another lookup
 * table.  The key is the symbol (the name) and the result is whatever
 * information has been associated with that symbol (e.g., the symbol's
 * type, line that it occurs, etc.).
 *
 * <p>
 *
 * In addition to adding and removing symbols, symbol tables also
 * support operations for entering and exiting scopes and for checking
 * whether an identifier is already defined in the current scope.  The
 * lookup operation must also observe the scoping rules of the language;
 * if there are multiple definitions of identifier <code>x</code>, the
 * scoping rules determine which definition a lookup of <code>x</code>
 * returns.  In most languages, including Decaf, inner definitions hide
 * outer definitions.  Thus, a lookup on <code>x</code> returns the
 * definition of <code>x</code> from the innermost scope with a
 * definition of <code>x</code>.
 *
 * <p>
 *
 * This example symbol table is implemented using Java hashtables.  Each
 * hashtable represents a scope and associates a symbol with some
 * data. The ``data'' is whatever data the programmer wishes to
 * associate with each identifier.
 *
 * */
class SymbolTable {
    private Stack tbl;

    /** Creates an empty symbol table. */
    public SymbolTable() {
        tbl = new Stack();
    }

    /** Enters a new scope. A scope must be entered before anything
     * can be added to the table.
     * */
    public void enterScope() {
        tbl.push(new Hashtable());
    }

    /** Exits the most recently entered scope. */
    public void exitScope() {
        if (tbl.empty()) {
            System.err.println("Error --> existScope: can't remove scope from an
                empty symbol table.");
```

```
    }
    tbl.pop();
}

/** Adds a new entry to the symbol table.
 *
 * @param id the name
 * @param info the data asosciated with id
 * */
public void addId(String name, Object info) {
    if (tbl.empty()) {
            System.err.println("Error --> addId: can't add a symbol without
                a scope.");
    }
    ((Hashtable)tbl.peek()).put(name, info);
}

/**
 * Looks up an item through all scopes of the symbol table.  If
 * found it returns the associated information field, if not it
 * returns <code>null</code>.
 *
 * @param name the symbol
 * @return the info associated with name, or null if not found
 * */
public Object lookup(String name) {
    if (tbl.empty()) {
            System.err.println("Error --> lookup: no scope in symbol table."
                );
    }
    // I break the abstraction here a bit by knowing that stack is
    // really a vector...
    for (int i = tbl.size() - 1; i >= 0; i--) {
        Object info = ((Hashtable)tbl.elementAt(i)).get(name);
        if (info != null) return info;
    }
    return null;
}

/**
 * Probes the symbol table.  Check the top scope (only) for the
 * symbol <code>name</code>.  If found, return the information field.
 * If not return <code>null</code>.
 *
 * @param name the symbol
 * @return the info associated with name, or null if not found
 * */
public Object probe(String name) {
    if (tbl.empty()) {
            System.err.println("Error --> lookup: no scope in symbol table."
                );
    }
    return ((Hashtable)tbl.peek()).get(name);
}

/** Gets the string representation of the symbol table.
 *
```

```
     * @return the string rep
     * */
    public String toString() {
        String res = "";
        // I break the abstraction here a bit by knowing that stack is
        // really a vector...
        for (int i = tbl.size() - 1, j = 0; i >= 0; i--, j++) {
            res += "Scope " + j + ": " + tbl.elementAt(i) + "\n";
        }
        return res;
    }
}
```