```java
import java.util.*;

/**
 * SymbolTable is a datatype for a tree structured table, where each node in
 * the tree represents a program scope. The overall tree structure represents
 * the scope nesting of a program. For example, consider the following
 * (Fascal) program:
 */

program
  var p1, p2, p3: integer;
  begin
    a3 := a1 + a2;
  end A;
  procedure B(b1: real; b2: integer);
  var b3: integer;
  begin
    b1 := b2 - b3;
  end B;
  begin
    p1 := p2 - p3;
  end

...
integer p1, p2, p3;

This declaration is represented by three entries with names "p1", "p2", and "p3", respectively. The type for all three entries is integer.

An important instance of other information is that for symbols which define a scope. For example, consider the following procedure declaration from the program above:

procedure B(real b1, integer b2);

An important instance of other information is that for symbols which define ascope. For example, consider the following procedure declaration from the program above:

procedure B(real b1, integer b2);

Asymtab entry for the identifier B has the following values in the header:

name = "B", type = void

The entry also has a scope field, which is a reference to its own local symbol table. The documentation for the FunctionEntry extension of SymbolTableEntry has further discussion.

public class SymbolTable {

/**
 * Allocate a new symtab of the given size. The size is the number of table entries (not bytes). All entries are initialized to null, the parent is initialized to null, and level to 0. Parent and level are only set to non-null/non-zero values when a SymbolTable is constructed with the newLevel method.
 */
 public SymbolTable(int size) {
    entries = new HashMap(size);
    level = 0;
 }

/**
 * Allocate a new symtab and add it as a new level to this symtab. The new level is linked into the existing symtab via the scope field of the given function entry, and the parent entry of this, as illustrated in the class documentation. The level field of the the new symtab is set to this.level+1. The return value is a reference to the new level.
 */
 public SymbolTable newLevel(FunctionEntry fe, int size) {

SymbolTable newst = fe.scope = new SymbolTable(size);

/* Link the parent and parententry fields of the new table to their appropriate parent locations.
 */
 newst.parent = this;

/* Set the level of the new table to one greater than the parent level.
 */
 newst.level = level + 1;

return newst;
 }

/**
 * Lookup an entry by name in this symtab. The symtab entry of the given name is returned, if found, else null is returned. The lookup algorithm is based on the symtab tree structure outlined above. Specifically,
 *<p>
 * (1) Lookup first checks in the given symtab; if an entry of the given name is found there, it is returned.
 *<p>
 * (2) If (1) fails, Lookup ascends through successive parent levels of the given symtab, performing another look up at each level. If an entry of the given name is found at a parent level, it is returned. Note that Lookup will return the entry from the youngest parent level in which it is found, even if one or more older parent levels also contain an entry of the same name.
 *<p>
 * (3) If the top level is reached without finding an entry of the given name, null is returned.
 *<p>
 * This lookup algorithm is intended to model the open scope resolution rule of most block structured programming languages. Viz., a reference to a symbol within an open scope is resolved by looking in the current scope, and if not found there, successive levels of enclosing scopes are searched.
 *<p>
 */
 public SymbolTableEntry lookup(String name) {

int i;
 SymbolTable st;
 SymbolTableEntry se;

/* For this and each parent level, search for an entry of the given name.
 */
 for (st = this; st != null; st = st.parent) {

/* Just use get in the HashMap -- sweet.
 */

if ((se = (SymbolTableEntry) entries.get(name)) != null) {
    return se;
}

/*
 * Return null if symbol is found no where.
 */
return null;

/**
 * Lookup an entry by name in this symtab only. I.e., LookupLocal does not
 * perform the parent-level search that is performed by Lookup. Otherwise,
 * the specification is the same as Lookup.
 * <p>
 * This version of lookup is intended to model the closed scope resolution
 * rule of most block structured programming languages. Viz., a reference
 * to a symbol within a closed scope is resolved by looking in the current
 * scope only, without subsequent checks in enclosing scopes.
 */
public SymbolTableEntry lookupLocal(String name) {
    return (SymbolTableEntry) entries.get(name);
}

/**
 * Enter the given symtab entry into this symtab, if an entry of that name
 * does not already exist. True is returned if the entry was added, false
 * otherwise.
 */
public boolean enter(SymbolTableEntry se) {
    if (lookupLocal(se.name) != null) {
        return false;
    }
    entries.put(se.name, se);
    return true;
}

/**
 * Move up one parent level from this symtab, returning a reference to the
 * new level. If the current level of this symtab has no parent (i.e., it
 * is at level 0), then Ascend has no effect, i.e., it returns a reference
 * to this.
 */
public SymbolTable ascend() {
    return parent != null ? parent : this;
}

/**
 * Move down one level in this symtab, returning a reference to the new
 * level. The level descended to is the one referenced by the symtab entry
 * of the given name, which must have scope field, i.e., it must be a
 * FunctionEntry. If no such entry exists, of if the given name is not
 * that of a FunctionEntry, then descend has no effect, i.e., it returns a
 * reference to this.
 */
public SymbolTable descend(String name) {
    SymbolTableEntry se = lookupLocal(name);
    try {
        return se == null ||
          ((se.getClass() != Class.forName("FunctionEntry")))
            ? this : ((FunctionEntry) se).scope;
    } catch (Exception e) {
      // ClassNotFound exceptin; this is a pain
      System.out.println(e);
      e.printStackTrace();
      return null;
    }

```java
public void dump(SymbolTable st) {
    System.out.println(toString());
}

/**
 * Produce the string value printed by dump.
 */
public String toString() {
    return toString(this.level);
}

/**
 * Work doer for toString. The level parameter is used for indenting.
 */
public String toString(int level) {
    SymbolTableEntry e;
    String indent = "", output = "";
    int nextLevel = level + 1;
    /*
     * Indent per level.
     */
    for (int i = 0; i < level; i++) {
        indent += " ";
    }
    /*
     * Message at top of table.
     */
    output += "Level " + Integer.toString(level) + " Symtab Contents:\n";
    /*
     * Serially traverse the entries and dump each.
     */
    for (Iterator it = entries.values().iterator(); it.hasNext(); ) {
        output += ((SymbolTableEntry)it.next()).toString(nextLevel) +
            (it.hasNext() ? "\n" : "");
    }
    return output;
}

/**
 * The parent table in the tree structure, i.e., the symtab of this'
 * enclosing scope. This is null for the level 0 symtab.
 */
public SymbolTable parent;

/**
 * The hash table of entries
 */
protected HashMap entries;

/**
 * Nesting level of this, starting with 0 at the top.
 */
public int level;

/**
 * Incrementing counter for the memory addresses of data values declared
 * in this symtab's scope. During parsing and symbol table construction,
 * this is used as the memory address offset counter. Once all of the
 */
public int memorySize;
```