```
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
 * (Pascal) program:
 *                                                                      <p><pre>
      program
         var p1,p2,p3: integer;

         procedure A(a1: integer; a2: real);
            var a3: integer;
         begin
            a3 := a1 + a2;
         end A;
         procedure B(b1: real; b2: integer);
           var b3: integer;
           procedure C(c1: integer; c2: real);
             var c3: integer;
           begin
              c1 := c3;
              c2 := c1 * c3 / 10;
           end C;
         begin
           b1 := b2 - b3;
         end B;
      begin
         p1 := p2 - p3;
      end
 *                                                                    </pre><p>
 * An abstract depiction of the symbol table structure for this program is the
 * following:
 *                                                                      <pre><p>
       program symtab
      |---------------|
      |      null     |<--|
      |-------|-------|   |       B's symtab
      |   p3  |       |   |   |---------------|
      |-------|-------|   |---|-----------o       |<--|
      |       |       |       |-------|-------|   |
      |-------|-------|       |   b2  |       |   |       C's symtab
      |   B   |   o-----------|>|-------|-------|   |   |---------------|
      |-------|-------|             . . .         |-----------o       |
      |   p1  |       |       |-------|-------|       |-------|-------|
      |-------|-------|       |   C   |   o-----------|>|   c3  |       |
      |   p2  |       |       |-------|-------|       |-------|-------|
      |-------|-------|                                     . . .
      |       |       |                                   |-------|-------|
      |-------|-------|           A's symtab
      |   A   |   o-----------|>|---------------|
      |-------|-------|       |       o-------------> back to program symtab
                              |-------|-------|
                              |   a1  |       |
                              |-------|-------|
                                      . . .
                                    |-------|-------|
 *                                                                    </pre><p>
 * Note that a number of structural details are omitted from this picture.
 * What the picture depicts is the overall tree structure, and how it
 * represents the nested scope structure of the program.  The details that are
 * shown are the following:
 *                                                                        <p>
 *    (1) Each symtab in the tree has a parent pointer that links it to the
 *        symtab for the enclosing scope in the program.  The symtab for the
 *        outermost scope has no parent.  This topmost symbol table is referred
 *        to as "level 0".
 *                                                                        <p>
 *    (2) The table at each level contains entries for all of the identifiers
 *        defined in the program scope represented by that table.  For example,
 *        the program symtab has entries for the variables p1, p2 and p3, and
 *        for procedures A and B.  In turn, the symtab for procedure B's scope
 *        has entries for parameters b1 and b2, local variable b3, and local
 *        procedure C (not all of which are shown in the picture).
 *                                                                        <p>
 *    (3) Each entry that defines a new scope has a link to its own symbol
 *        table.  For example, procedure B above is entered by name in the
 *        program symbol table.  Since procedure B defines a scope of its own,
 *        the entry for B points to a symbol table that contains the
 *        identifiers declared within B's scope.  Per point (1) above, B's
 *        symtab has a parent pointer back to the program symtab.
 *                                                                        <p>
 *    (4) The entries in the symtabs are depicted in an order other than
 *        alphabetic to indicate that the body of a symbol table is probably
 *        hashed.  I.e., entries are shown in an apparent hashing order, rather
 *        than sequentially or in some lexical order.  Under any circumstances,
 *        users of the symtab abstraction may not assume any order for the
 *        entries within a table.
 *
 * As noted, the picture above omits some structural details.  In particular,
 * all of the publicly accessible fields for a table entry are not shown.  The
 * type SymbolTableEntry*is an abstract type for the entries within a symtab.
 * The general format of a symtab entry is the following:
 *                                                                  <pre><p>
 *              |-------------------------------|
 *              |         symbol name           |
 *              |-------------------------------|
 *              |         symbol type           |
 *              |-------------------------------|
 *              |                               |
 *              |    other information in       |
 *              |      extending classes        |
 *              |                               |
 *              |              ...              |
 *              |                               |
 *              |-------------------------------|
 *                                                                    </pre><p>
 * The name and type fields are common to all symtab entries, the value of the
 * type be null.  As an example, consider the following variable declaration
 * from the program above:
 *                                                                        <p>
```

```
 *      integer p1, p2, p3;
 *                                                                      <p>
 * This declaration is represented by three entries with names "p1", "p2", and
 * "p3", respectively.  The type for all three entries is integer.
 *                                                                      <p>
 * 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:
 *                                                                      <p>
 *      procedure B(real b1, integer b2);
 *                                                                      <p>
 *          ...
 *                                                                      <p>
 * A symtab entry for the identifier B has the following values in the header:
 *                                                                      <p>
 *     name = "B", type = void
 *                                                                      <p>
 * 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 documnentation.  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;

        /*
         * Enter the given entry in the current level.
         */
        enter(fe);

        /*
         * Create a new symtab for the new level, and link it into the
         * structure by pointing the info.proc.symtab field off to it.
         */
        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.
     *
     */
    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.
             */
```

```
225             if ((se = (SymbolTableEntry) entries.get(name)) != null) {
226                 return se;
227             }
228         }
229 
230         /*
231          * Return null if symbol is found no where.
232          */
233         return null;
234 
235     }
236 
237     /**
238      * Lookup an entry by name in this symtab only.  I.e., LookupLocal does not
239      * perform the parent-level search that is performed by Lookup.  Otherwise,
240      * the specification is the same as Lookup.
241      *                                                                      <p>
242      * This version of lookup is intended to model the closed scope resolution
243      * rule of most block structured programming languages.  Viz., a reference
244      * to a symbol within a closed scope is resolved by looking in the current
245      * scope only, without subsequent checks in enclosing scopes.
246      */
247     public SymbolTableEntry lookupLocal(String name) {
248         return (SymbolTableEntry) entries.get(name);
249     }
250 
251     /**
252      * Enter the given symtab entry into this symtab, if an entry of that name
253      * does not already exist.  True is returned if the entry was added, false
254      * otherwise.
255      */
256     public boolean enter(SymbolTableEntry se) {
257         if (lookupLocal(se.name) != null) {
258             return false;
259         }
260         entries.put(se.name, se);
261         return true;
262     }
263 
264     /**
265      * Move up one parent level from this symtab, returning a reference to the
266      * new level.  If the current level of this symtab has no parent (i.e., it
267      * is at level 0), then Ascend has no effect, i.e., it returns a reference
268      * to this.
269      */
270     public SymbolTable ascend() {
271         return parent != null ? parent : this;
272     }
273 
274     /**
275      * Move down one level in this symtab, returning a reference to the new
276      * level.  The level descended to is the one referenced by the symtab entry
277      * of the given name, which must have scope field, i.e., it must be a
278      * FunctionEntry.  If no such entry exists, of if the given name is not
279      * that of a FunctionEntry, then descend has no effect, i.e., it returns a
280      * reference to this.
281      */
282     public SymbolTable descend(String name) {
283         SymbolTableEntry se = lookupLocal(name);
284 
285         try {
286             return
287                 ((se == null) ||
288                     (se.getClass() != Class.forName("FunctionEntry")))
289                 ? this : ((FunctionEntry) se).scope;
290         }
291         catch (Exception e) {   // ClassNotFound exceptin; this is a pain
292             System.out.println(e);
293             e.printStackTrace();
294             return null;
295         }
296     }
297 
298     /**
299      * Dump out the contents of the given symbtab, dumping entries serially,
300      * and recursively traversing into scoping levels.  Empty entries are not
301      * dumped.  The serial order means that entries are dumped in the physical
302      * order they appear in the table.  Hence, if the entries are hashed, they
303      * will appear in the dump at their hashed entry positions, not sorted by
304      * symbol name or other more useful/aesthetic order.
305      *                                                                      <p>
306      * As an example, the following is a symtab dump for the sample program and
307      * picture shown above:
308      *
309      *                                                                      <pre><p>
310         Level 1 Symtab Contents:
311           Entry 7: Symbol: B, Type: 0x0
312             Formals: b1,b2
313             Level 2 Symtab Contents:
314               Entry 9: Symbol: b1, Type: 0x68760
315               Entry 12: Symbol: b2, Type: 0x66312
316               Entry 15: Symbol: b3, Type: 0x66312
317               Entry 18: Symbol: C, Type: 0x0
318                 Formals: c1,c2
319                 Level 3 Symtab Contents:
320                   Entry 20: Symbol: c1, Type: 0x66312
321                   Entry 23: Symbol: c2, Type: 0x68760
322                   Entry 26: Symbol: c3, Type: 0x66312
323           Entry 195: Symbol: p1, Type: 0x66312
324           Entry 200: Symbol: p2, Type: 0x66312
325           Entry 203: Symbol: p3, Type: 0x66312
326           Entry 228: Symbol: A, Type: 0x0
327             Parms: a1,a2
328             Level 2 Symtab Contents:
329               Entry 39: Symbol: a1, Type: 0x66312
330               Entry 42: Symbol: a2, Type: 0x68760
331               Entry 52: Symbol: a3, Type: 0x66312
332      *                                                                   </pre><p>
333      *
334      * The dump format of the type fields is an object memory address, for
335      * brevity.
336      */
```

```
337     public void dump(SymbolTable st) {
338         System.out.println(toString());
339     }
340 
341     /**
342      * Produce the string value printed by dump.
343      */
344     public String toString() {
345         return toString(this.level);
346     }
347 
348     /**
349      * Work doer for toString.  The level parameter is used for indenting.
350      */
351     public String toString(int level) {
352         SymbolTableEntry e;
353         String indent = "", output = "";
354         int nextLevel = level + 1;
355 
356         /*
357          * Indent per level.
358          */
359         for (int i = 0; i < level; i++) {
360             indent += "  ";
361         }
362 
363         /*
364          * Message at top of table.
365          */
366         output += "Level " + Integer.toString(level) + " Symtab Contents:\n";
367 
368         /*
369          * Serially traverse the entries and dump each.
370          */
371         for (Iterator it = entries.values().iterator(); it.hasNext(); ) {
372             output += ((SymbolTableEntry)it.next()).toString(nextLevel) +
373                 (it.hasNext() ? "\n" : "");
374         }
375 
376         return output;
377     }
378 
379     /** The parent table in the tree structure, i.e., the symtab of this'
380      * enclosing scope.  This is null for the level 0 symtab.
381      */
382     public SymbolTable parent;
383 
384     /** The hash table of entries */
385     protected HashMap entries;
386 
387     /** Nesting level of this, starting with 0 at the top. */
388     public int level;
389 
390     /** Incrementing counter for the memory addresses of data values declared
391      * in this symtab's scope.  During parsing and symbol table construction,
392      * this is used as the memory address offset counter.  Once all of the
393      * vars, and if appropriate parameters, have been entered in the this
394      * scope, the resulting value of this counter field is the size of the
395      * memory necesary for this scope.  For for the level 0 symtab, this is the
396      * size of the static pool.  For a function scope, this is the size of its
397      * activation record.
398      */
399     public int memorySize;
400 
401 }
```