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
 * (Fascial) program:
 */

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;
        begin
            c1 := c2;
            c2 := c1 * c3 / 10;
        end C;
    begin
        b1 := b2 - b3;
    end B;
    begin
        p1 := p2 - p3;
    end

/**
 * 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:
 */

(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".

(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).

(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.

(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
* general format of a symtab entry is the following:

<table>
<thead>
<tr>
<th>symbol name</th>
<th>symbol type</th>
</tr>
</thead>
<tbody>
<tr>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
</tr>
</tbody>
</table>

* 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:
*/
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);

As ymtab 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 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,
 * (1) Lookup first checks in the given symtab; if an entry of the given name is found there, it is returned.
 * (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.
 * (3) If the top level is reached without finding an entry of the given name, null is returned.
 * 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;
    SymbolTable newst;
    SymbolTableEntry newse;

    for (st = this; st != null; st = st.parent) {
        newse = st.lookup(name);
        if (newse != null)
            return newse;
    }

    newst = st;
    newse = newst.lookup(name);
    if (newse != null)
        return newse;

    return null;
}

}
SymbolTable.java

```java
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.
 * 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, or if the given name is not
 * that of a FunctionEntry, then descend has no effect, i.e., it returns
 * reference to this.
 */
public SymbolTable descend(String name) {
    SymbolTableEntry se = lookupLocal(name);
    try {
        return se;
    }
    catch (Exception e) { // ClassNotFound exceptin; this is a pain
        System.out.println(e);
        e.printStackTrace();
        return null;
    }
}
```

/*
 * Dump out the contents of the given symtab, dumping entries serially,
 * and recursively traversing into scoping levels. Empty entries are not
 * dumped. The serial order means that entries are dumped in the physical
 * order they appear in the table. Hence, if the entries are hashed, they
 * will appear in the dump at their hashed entry positions, not sorted by
 * symbol name or other more useful/aesthetic order.
 * As an example, the following is a symtab dump for the sample program and
 * picture shown above:
 */

/**
 * Level 1 Symtab Contents:
 * Entry 7: Symbol: B, Type: 0x0
 * Formals: b1,b2
 * Level 2 Symtab Contents:
 * Entry 9: Symbol: b1, Type: 0x68760
 * Entry 12: Symbol: b2, Type: 0x66312
 * Entry 15: Symbol: b3, Type: 0x66312
 * Entry 18: Symbol: C, Type: 0x0
 * Formals: c1,c2
 * Level 3 Symtab Contents:
 * Entry 20: Symbol: c1, Type: 0x66312
 * Entry 23: Symbol: c2, Type: 0x68760
 * Entry 26: Symbol: c3, Type: 0x66312
 * Entry 195: Symbol: p1, Type: 0x66312
 * Entry 230: Symbol: p2, Type: 0x66312
 * Entry 233: Symbol: p3, Type: 0x66312
 * Entry 238: Symbol: A, Type: 0x0
 * Formals: a1,a2
 * Level 2 Symtab Contents:
 * Entry 39: Symbol: a1, Type: 0x66312
 * Entry 42: Symbol: a2, Type: 0x68760
 * Entry 52: Symbol: a3, Type: 0x66312
 * */
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;