

CS 536 — Spring 2008

Programming Assignment 1 Symbol Table Classes

Due: Monday, February 4, 2008

Not accepted after Monday, February 11, 2008

Introduction

You are to write a set of Java classes that implement a **block-structured symbol table**. You must also write a test driver and create test data that thoroughly test your symbol table implementation.

You will implement or use the following six Java classes: `Symb`, `SymbolTable`, `TestSymb`, `DuplicateException`, `EmptySTException` and `P1`.

- Subclasses of the `Symb` class will eventually be used in your compiler to store information about each identifier that appears in a program (i.e., the variable and function names). The only information stored in a `Symb` will be the name of the identifier (a `String`); more information will be added to subclasses of `Symb`. Recall that Java's subclassing rules allow any subclass of `Symb` to be used where a `Symb` object is expected. This means that the symbol table methods we develop in this project will accept all subclasses of `Symb`. `TestSymb` is a subclass of `Symb` that contains a single integer field. It is used to test the operation of the `SymbolTable` class.
- The `SymbolTable` class will implement a block-structured symbol table. It can be built using a linked list of Java `Hashtable` objects, one for each open scope.
- The `DuplicateException` and `EmptySTException` classes are exceptions that can be thrown by methods of the `SymbolTable` class.
- Class `P1` will implement an interactive test driver used to test your `SymbolTable` class.

Class Specifications

1.class `Symb`

<code>Symb(String s)</code>	The class constructor; initialize <code>Symb</code> to have name <code>s</code> .
<code>String name()</code>	Return the name of this <code>Symb</code> .
<code>String toString()</code>	Return a string representation of this <code>Symb</code> object.

2.class TestSym

TestSym(String s, int i)	The class constructor; initialize TestSym to have name s and value i.
int value()	Return the value of this TestSym.
String toString()	Return a string representation of this TestSym object.

3.class SymbolTable

SymbolTable()	The class constructor; initialize SymbolTable to contain a single scope that is initially empty.
void openScope()	Add a new, initially empty scope to the list of scopes contained in this SymbolTable.
void closeScope()	If the list of scopes in this SymbolTable is empty, throw an EmptySTException. Otherwise, remove the current (front) scope from the list of scopes contained in this SymbolTable.
void insert(Symb s)	If the list of scopes in this SymbolTable is empty, throw an EmptySTException. If the current (first) scope contains a Symb whose name is the same as that of s (ignoring case), throw a DuplicateException. Otherwise, insert s into the current (front) scope of this SymbolTable.
Symb localLookup(String n)	If the list of scopes in this SymbolTable is empty, return null. If the current (first) scope contains a Symb whose name is n (ignoring case), return that Symb. Otherwise, return null.
Symb globalLookup(String n)	If any scope contains a Symb whose name is n (ignoring case), return the first matching Symb found (in the scope nearest to the front of the scope list). Otherwise, return null.
void dump(PrintStream p)	This method is for debugging. The contents of this SymbolTable are written to Printstream p (System.out is a Printstream).
String toString()	Return a string representation of this SymbolTable.

4.class P1

void main(String[] args)	The test driver used to test your SymbolTable implementation.
--------------------------	---

5.classes DuplicateException and EmptySTException

These two classes are empty. They are used to signal duplicate insertion and empty symbol tables errors.

Getting Started

The Systems Lab supports two Java compilers, `javac` and `jikes`. `javac` is the Sun Microsystems version 1.5 JDK. This compiles the most recent version of Java including polymorphic classes. `jikes`, developed by IBM, is usually faster to run than `javac`. It also often gives better error messages. However, it is a bit out-of-date and implements Java 1.4.2 (no version 1.5 features or classes). If you plan to use Java 1.5 features, you can't use `jikes`. If you don't need (or want) classes and features beyond Java 1.4.2, you may use either `javac` or `jikes`. You may also use other quality Java compilers or development systems. (*Eclipse* is free and highly regarded).

Later we will use the *JLex* scanner generator and the *JavaCup* parser generator. To make sure these Java-based tools operate properly, put the following two lines in your `.cshrc.local` file (which can be found in your home directory):

```
setenv CLASSPATH ".../classes:/s/java/jre/lib/rt.jar:/p/course/
cs536-fischer/public/JAVA"

setenv VPATH "./classes"
```

(These are two lines, not three. Ignore the line break after the `/.`)

We have placed partial implementations of the required classes, along with a `Makefile` and sample test data in `~cs536-1/public/proj1/startup`. You will certainly need to edit and extend the `SymbolTable` and `P1` classes. You may leave the other classes (which are quite simple) as they are. The `Makefile` allows you to easily compile and test your solution to this assignment. You should use `make` to speed and simplify program development. The command

```
make
```

will recompile classes as needed after any changes you make. The command

```
make test
```

will do necessary recompilations and then test your solution by calling `P1.main` with the commands in `testInput`. (You should edit this file to more thoroughly test your implementation). The command

```
make clean
```

will remove all classfiles created by the compiler. We place all class files in the `classes` subdirectory. This is done to avoid cluttering your top-level project directory.

You will probably want to use the standard Java utility class `java.util.Hashtable` in implementing your block-structured symbol table. This class comes in two versions.

The original version is detailed at

<http://java.sun.com/j2se/1.4.2/docs/api/index.html>.

It assumes all keys and values in the hash table have class `Object`. This means objects retrieved from a hash table must be explicitly cast into their correct class.

Since Java 1.5 supports polymorphism, an updated definition of the `Hashtable` class has been created. `Hashtable<K,V>` defines a hash table in which all keys have class `K` and all table entries have class `V`. Explicit casting is not required. This newer version of hash table is only available in compilers that support version 1.5 (or later). It is detailed at <http://java.sun.com/j2se/1.5.0/docs/api/index.html>.

The Test Driver

You'll need to create an interactive test driver, in method `main` of class `P1`, to test the operation of your block structured symbol table. Your test driver should accept the following commands. One letter abbreviations of the commands should be allowed.

Command	Operation
Open	Open a new scope
Close	Close the top (innermost) scope.
Dump	Dump the contents of symbol table.
Insert	Read a string and an integer and insert the (string,integer) pair into the innermost scope.
Lookup	Read a string and lookup (in the top scope) the symbol table entry associated with the string. Print the integer in the symbol table entry found.
Global	Read a string and lookup (in the nearest scope that contains an entry) the symbol table entry associated with the string. Print the integer in the symbol table entry found.
Quit	Exit the test driver.

The following illustrates the operation of the test driver (text entered by the user is printed in bold face). Note that this example is only meant to illustrate our testing interface; it does not by itself represent an exhaustive test set. To facilitate automatic grading, please make your wording of responses to commands similar to that shown below.

insert

```
Enter symbol:wisconsin
Enter associated integer:1848
(wisconsin:1848) entered into symbol table.
```

insert

```
Enter symbol:florida
Enter associated integer:1845
(florida:1845) entered into symbol table.
```

lookup

```
Enter symbol:Wisconsin
(wisconsin:1848) found in top scope
```

lookup

```
Enter symbol:Florida
(florida:1845) found in top scope
```

lookup

```
Enter symbol:Hawaii
Hawaii not found in top scope
```

```

insert
Enter symbol:wisconsin
Enter associated integer:1836
wisconsin already entered into top scope.
open
New scope opened.
insert
Enter symbol:wisconsin
Enter associated integer:1836
(wisconsin:1836) entered into symbol table.
lookup
Enter symbol:Wisconsin
(wisconsin:1836) found in top scope
dump
Contents of symbol table:
{wisconsin=(wisconsin:1836)}
{florida=(florida:1845), wisconsin=(wisconsin:1848)}
lookup
Enter symbol:Florida
Florida not found in top scope
global
Enter symbol:Florida
(florida:1845) found in symbol table
close
Top scope closed.
lookup
Enter symbol:Wisconsin
(wisconsin:1848) found in top scope
lookup
Enter symbol:Florida
(florida:1845) found in top scope
close
Top scope closed.
lookup
Enter symbol:Wisconsin
Wisconsin not found in top scope
quit
Testing done

```

What To Hand In

We will create a directory for you based on your login in `~cs536-1/public/proj1/handin`. Copy into your handin directory your versions of `SymbolTable.java`, `P1.java`, and any other Java source files you changed or added. If you changed the Makefile we provided, include your version (so that we can properly build and test your symbol table routines). Include your version of `testInput` that comprises the tests you used to verify the operation of your symbol table routines. Include `testOutput`, which is the output generated by your program in response to your `testInput` file. You should include a `README` file to hold external documentation. We'll run your program on a variety of our own test programs. Do not hand in any class files; we'll create them as needed using your source files and `Makefile`.

When your program begins execution it should print out your full name and student ID number. We will grade your program on the basis of the completeness of your testing (as shown in the `testInput` and `testOutput` files) as well as the correct operation of your symbol table routines.

The quality of your documentation is also important. Make sure that you provide both external documentation (in the `README` file) and internal documentation (in the source files). It should be easy for the grader to understand the organization and structure of your program. We may exact significant penalties if we find your program poorly documented or difficult to understand.