CS 541 — Spring 2014

Programming Assignment 1
Symbol-Table Classes

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 should implement or use the following six Java classes: `Symb`, `SymbolTable`, `TestSym`, `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 (such as the variable and function names). The only information stored in a `Symb` is the name of the identifier (a `String`); more information appears in subclasses of `Symb`. Java's subclassing rules allow any subclass of `Symb` to be used where a `Symb` object is expected. The symbol table methods we develop in this project accept all subclasses of `Symb`. `TestSym` 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 implements a block-structured symbol table. It can be built using a linked list of Java `HashMap` 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` implements an interactive test driver used to test your `SymbolTable` class.

Class Specifications

`class Symb`

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

`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` object.
`String toString()`	Return a string representation of this `TestSym` object.

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`.

class P1

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

classes DuplicateException and EmptySTException

These two classes, which extend `java.lang.Exception`, are empty. They are used to signal duplicate-insertion and empty symbol-table errors.

Getting Started

The MultiLab supports the `javac` Java compiler, Oracle version 1.6 JDK, which compiles a recent version of Java, including polymorphic classes. You may also use other quality Java compilers or integrated development environments. (*Eclipse* is free and highly regarded).

You can find partial implementations of the required classes, along with a `Makefile` and sample test data, in `~raphael/courses/cs541/public/proj1/-startup` (or use the tarball http://www.cs.uky.edu/~raphael/courses/-CS541/startup1.tar.gz). You will certainly need to edit and extend the `SymbolTable` and `P1` classes. You may leave the other classes (which are quite simple) mostly 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
```

recompiles classes as needed after any changes you make. The command

```
make test
```

recompiles as necessary and then tests 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
```

removes all class files created by the compiler. All class files reside in the `classes` subdirectory to avoid cluttering your top-level project directory. The command

```
make style
```

runs a style checker on your code and suggests improvements. Take them seriously!

Use the standard Java utility class `java.util.HashMap` in implementing your block-structured symbol table. `HashMap<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. You might also find `java.util.Scanner` useful. You can find details of all Java library routines at http://download.oracle.com/javase/6/docs/api/.

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.

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.

One-letter abbreviations of the commands should be allowed.

The following illustrates the operation of the test driver (text entered by the user is printed in **bold face**). 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: kentucky
Enter associated integer: 1848
(kentucky:1848) entered into symbol table.
insert
Enter symbol: florida
Enter associated integer: 1845
(florida:1845) entered into symbol table.
lookup
Enter symbol: Kentucky
(kentucky: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: kentucky
Enter associated integer: 1836
kentucky already entered into top scope.
open
New scope opened.
```

```
insert
Enter symbol: kentucky
Enter associated integer: 1836
(kentucky:1836) entered into symbol table.
lookup
Enter symbol: Kentucky
(kentucky:1836) found in top scope.
dump
Contents of symbol table:
{kentucky=(kentucky:1836)}
{florida=(florida:1845), kentucky=(kentucky: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: Kentucky
(kentucky:1848) found in top scope.
lookup
Enter symbol: Florida
(florida:1845) found in top scope.
close
Top scope closed.
lookup
Enter symbol: Kentucky
Kentucky not found in top scope.
quit
Testing done
```

What To Hand In

Submit your project electronically by mailing it to raphael@cs.uky.edu. Please run `make clean` first to remove all `class` files. 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. Include a `README` file to hold external documentation. We'll run your program on our own test data.

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.