Project #2

Computer Science 2334
Fall 2008

User Request:

"Create a Word Search System."

Milestones:

1.	Use keyboard input to get information from the user.	*5 points*
2.	Use text file I/O to read and write text files.	*10 points*
3.	Create a class to store each word code; create a class to store a collection of words in a dictionary.	*10 points*
4.	Implement the **Comparator** interface for the item class.	*10 points*
5.	Use a list to store, retrieve, and display information related to words as described below.	*15 points*
6.	Use the `sort()` and `binarySearch()` methods from the **Collections** class to search for information related to the description below.	*20 points*
►	Develop and use a proper design.	*15 points*
►	Use proper documentation and formatting.	*15 points*

Description:

For this project, as with Project #1, you will put together several techniques and concepts learned in CS 1323 and some new techniques to make an application that searches a large database of words which we will call a "dictionary" even though it is only a list of words, without definitions. Note that much of the code you write for this program could be reused in more complex applications, such as a spell-checker or an actual dictionary (with definitions).

Your software will read in a comma-delimited data file and store the information into the dictionary. Each entry in the data file contains the word itself, the language from which the word comes, and a numerical value which we will call "score." The meaning of the score will become apparent in later assignments. For now you may simply treat it as an integer.

Once the file has been read in, your program will enter a loop in which it asks the user for the name of two more files. This should be done using the technique described in the section below on reading input from the keyboard. The first of these two additional files will contain a list of queries for which your program should display information. Each query will be a pair consisting of a word and a language. For each query, your program will find the corresponding score. The result will then be written to the screen and written out to a user-specified output file. This output file will be the second of the two additional files the user specifies. Once the results of running the program have been written to the screen and the output file, the user will be asked if he or she wishes to continue with the program or exit. If the user asks to continue, your program will return to the point in the loop at which it asks the user for the name of two more files (the query and output files).

The name of the input file that contains the dictionary data will be supplied using Eclipse program arguments by the user. Review Project 1 for how to use program arguments.

Learning Objectives:

Sorting and Searching:

Sorting information can be useful to users because the output may be organized in a way that makes it easier to use. It can also be useful to software developers because it can improve the efficiency of their software. Consider finding dictionary entries based on their word and language. If the data structure holding the word data is unsorted, you need to do a linear search through it to find a word. However, if the data structure is sorted based on word and language, you can do a binary search instead. A binary search will, in most cases, take far fewer comparisons to find the desired entry than a linear search.

To observe this efficiency gain, write two versions of the project. In the first version, leave the data structure holding the dictionary entries unsorted and search through it linearly when the user provides the query and output file names. In the second, sort this data structure based on the word and language, then do a binary search on it. You will measure the amount of time the system uses in each case for internally handling dictionary data. Internally handling dictionary data includes both sorting and searching of it. **Do not count the time the system uses for reading in the dictionary data or carrying out other activities, like waiting for the user to provide input.** Run each version of your program 5 times using only the sample provided query file one time for each run, record the values for times used for (1) sorting and (2) searching in each version, and present them in a simple table that includes totals and averages. An example of the table format is shown below.

Version 1	Sort Time	Search Time	Total Time
Run 1			
Run 2			
Run 3			
Run 4			
Run 5			
Average			
Version 2	Sort Time	Search Time	Total Time
Run 1			
Run 2			
Run 3			
Run 4			
Run 5			
Average			

If you only use the sample query file, does the first or second version of your code spend less time internally handling zip code data? Why? If you use many additional query files, do you expect the first or second version of your code to spend less time internally handling zip code data? Why? Rather than creating additional query files, try having your system repeatedly process the same query file. Is there some number of repetitions at which the less efficient version becomes the more efficient version? Is so, approximately what number is that?

Put the table of data and your answers to these questions into MILESTONES.txt under milestone 5.

File Formats:

City Data File:

Each line of the file contains a single word, followed by a comma, followed by a space, followed by the language from which the word comes, followed by a comma, followed by a space, followed by an integer value. For example:

```
valedictorian, English, 27
```

Output Format:

The text written to the screen and the output file for each word matching the search criteria must follow the following output format.

Line 1: Name of the word and language searched for.

Line 2: Score of dictionary entry found

Sample Output:

```
Word and language searched for: valedictorian, English
Score: 27
```

Implementation Issues:

File I/O:

To perform output to a file, use the **FileWriter** class with the **BufferedWriter** class as follows.

```
FileWriter outfile = new FileWriter("output.txt");
BufferedWriter bw = new BufferedWriter(outfile);
bw.write("This is a test -- beep.");
bw.newLine();
bw.close();
```

When you have finished writing to a file, you must remember to close it, or the file won't be saved. If you fail to close the file, it will be empty!

Remember to add '`throws IOException`' to the signature of any method that uses a **FileWriter** or **BufferedWriter** or that directly or indirectly calls a method that performs File I/O.

Reading Input from the Keyboard:

In order to get the word information from the user, you need to read input from the Keyboard. This can be done using the **InputStream** member of the **System** class, that is named "`in`". When this input stream is wrapped with a **BufferedReader** object, the `readLine()` method of the **BufferedReader** class can be used to read and store all of the characters typed by the user into a **String**. Note that `readLine()` will block until the user presses the Enter key, i.e., the method call to `readLine()` will not return until the user presses the Enter key.

The following code shows how to wrap and read strings from `System.in` using an **InputStreamReader** and a **BufferedReader**.

```
BufferedReader inputReader = new BufferedReader(
                                new InputStreamReader( System.in ) );
System.out.print( "Type some input here: " );
String input = inputReader.readLine();
System.out.println( "You typed: " +  input );
```

You need to add '`throws IOException`' to the signature of any method that uses or that directly or indirectly calls a method that uses a **BufferedReader** or **InputStreamReader**.

Due Dates and Notes:

Your revised design and detailed Javadoc documentation are due on **Wednesday, October 1st**. Submit the project archive following the steps given in the submission instructions **by 9:00pm**. Submit your revised UML design on *engineering paper* or a hardcopy using UML layout software, a hardcopy of the index page of your Javadoc documentation, and a hardcopy of the stubbed source code at the **beginning of lab on Thursday, October 2nd**.

The final version of the project is due on **Wednesday, October 8th**. Submit the project archive following the steps given in the submission instructions **by 9:00pm**. Submit your final UML design on *engineering paper* or a hardcopy using UML layout software, a hardcopy of the index page of your Javadoc documentation and a hardcopy of the source code at the **beginning of lab on Thursday, October 9th**.

You are not allowed to use the **StringTokenizer** class. Instead you must use `String.split()` and a regular expression that specifies the delimiters you wish to use to "tokenize" or split each line of the file.

You may write your program from scratch or may start from programs for which the source code is freely available on the web or through other sources (such as friends or student organizations). If you do not start from scratch, you must give a complete and accurate accounting of where all of your code came from and indicate which parts are original or changed, and which you got from which other source. Failure to give credit where credit is due is academic fraud and will be dealt with accordingly.

As noted in the syllabus, you are required to work on this programming assignment in a group of at least two people. It is your responsibility to find other group members and work with them. The group should turn in only one (1) hard copy and one (1) electronic copy of the assignment. Both the electronic and hard copies should contain the names and student ID numbers of **all** group members. If your group composition changes during the course of working on this assignment (for example, a group of five splits into a group of two and a separate group of three), this must be clearly indicated in your write-up, including the names and student ID numbers of everyone involved and details of when the change occurred and who accomplished what before and after the change.

Each group member is required to contribute equally to each project, as far as is possible. You must thoroughly document which group members were involved in each part of the project. For example, if you have three functions in your program and one function was written by group member one, the second was written by group member two, and the third was written jointly and equally by group members three and four, both your write-up and the comments in your code must clearly indicate this division of labor. Giving improper credit to group members is academic misconduct and grounds for penalties in accordance with school policies