Project #3

Computer Science 2334
Spring 2009

User Request:

"Create a Fast Personal Information Search System
with Import/Export and Load/Save Features and a Graphical Display."

Milestones:

1.	Export contact information from the address book to a text file.	*5 points*
2.	Implement **Serializable** for the contact info and address book classes.	*10 points*
3.	Utilize Object Serialization to save the address book to a binary file. Utilize Object Serialization to load the address book from a binary file.	*15 points*
4.	Implement a simple graphical display for showing the relative frequency of data.	*25 points*
5.	Use the **HashMap** class to save to and retrieve from the address book in memory.	*15 points*
►	Develop and use a proper design.	*15 points*
►	Use proper documentation and formatting.	*15 points*

Description:

An import skill in software design is extending the work you have done in a previous project. For this project you will rework Project 2, using the Java **HashMap** class and add a graphical display. For this project, as with Projects 1 and 2, 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 contact information which we will call an *address book*. Note that much of the code you write for this program could be reused in more complex applications, such as address label printers or bulk emailers.

Importing/Exporting and Object Serialization:

Your software will read in a comma-delimited data file and store the information into the address book. The name of the import file that contains the contact info will be supplied by the user via program arguments. (Review Project 1 for how to use program arguments.) Each entry in the import file contains a person's last name, first name, email address, and physical address. The physical address itself will have several components to it which are a street number, a street name, a city, a state, and a zip code. This process is known as *importing* the address book since this file is not a file of contact info objects but a text file that needs to be parsed to construct contact info objects.

Once the file has been read in (imported), your program will enter a loop in which it asks the user for the name of four more files, one at a time as it does its work. This should be done using the technique described in the section below on reading input from the keyboard.

The first of these four additional files will contain a list of queries for which your program should display information. Each query will be a pair consisting of a first name and a last name. For each query, your program will find the corresponding personal information. (You may assume that the combination of first name and last name is unique in the address book.) The result will then be written to the screen and written out to a user-specified result output file. This output file will be the second of the four additional files the user specifies.

Once the results of running the program have been written to the screen and the result output file, the user will be asked for the name of the file to which he or she wishes to *export* the address book. This will be the third of the four additional files the user will specify. Once the user has specified this file name, he or she will be asked for address information on which to filter the exported address book. If the user enters no address information (simply hits return), the address book will not be filtered – the entire address book will be exported to the specified file. If the user enters one or more two letter state codes (such as OK or MN), your program should only export the dictionary entries for people who live in these states. If the user enters a state and a city name (such as Norman, OK or Minneapolis, MN), your program should export only those address book entries containing the city and state combination specified. The format for the export file will be exactly the same as the format of the import file.

After the file has been exported, the user will be asked for the name of a file to which the address book should be saved. *Saving* here means to use object serialization to write the data to the file. This is the fourth and final additional filename for which the user will be asked. Once the user has specified this file name, he or she will be asked for address information on which to filter the saved address book. Filtering for saving should be handled in the same way as filtering for exporting.

After saving the filtered data, 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 use object serialization to read from the saved dictionary file (the one created using object serialization), replacing the current dictionary in memory, and return to the point in the loop at which it asks the user for the name of four more files (the query and output files).

Graphical Display:

Producing graphical displays of information can be very useful to users. Therefore, in addition to reading and writing the files as described above, each time the dictionary is loaded (the first import and the subsequent replacements), your program will produce a graphical display of the states of the addresses in the address book. In particular, your program will display the state flag of each state that is present in the address book. Each state flag will be displayed no more than once, regardless of how many addresses in the address book list that state. However, the size of each flag displayed will be proportional to the number of times the state is listed in the address book. So, for example, if OK is listed 10 times, MN once, and ND not at all, then the Oklahoma state flag should be displayed 10 times as large as the Minnesota state flag and the North Dakota state flag should not be displayed at all. The details of the flag display are up to you (and you are responsible for finding flag images under a license that permits their use for this project). However, you should make good use of the space available in the display window. That is, the flags should take up a large portion of the space available in the window and should grow larger and shrink down as the window size is increased and decreased, respectively.

HashMaps, 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 personal information based on first and last name. If the data structure holding the personal data is unsorted, you need to do a linear search through it to find an entry. However, if the data structure is sorted based on name, 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.

Hash tables are an alternate way to quickly store and retrieve data. Java provides the **HashMap** class (among others) which provides this functionality.

To observe this efficiency gain, write three versions of the project. In the first version, leave the data structure holding the personal 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 first and last names, then do a binary search on it. In the third, you must first add the address book entries to a `HashMap` and then retrieve them from the `HashMap`. You will measure the amount of time the system uses in each case for internally handling personal data. Internally handling personal data includes adding the data to the data structure (list or `HashMap`) and retrieving it. **Do not count the time the system uses for reading in the personal 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		Insert Time	Sort Time	Search Time	Total Time
	Run 1				
	Run 2				
	Run 3				
	Run 4				
	Run 5				
	Average				
Version 2		Insert Time	Sort Time	Search Time	Total Time
	Run 1				
	Run 2				
	Run 3				
	Run 4				
	Run 5				
	Average				
Version 3		Insert Time	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, second, or third version of your code spend less time internally handling personal data? Why? If you use many additional query files, do you expect the first, second, or third version of your code to spend less time internally handling personal 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:

Personal Information File:

Each entry in the personal data file contains a person's last name, first name, email address, and physical address. The physical address itself will have several components to it which are a street number, a street name, a city, a state, and a zip code. Note that the components of the physical address should be stored in objects of their own class, which should then belong to objects of the individual person class. Each component of each entry will be separated from the others by a comma and a space. For example:

```
Gus, Bo, bogus@bogus.edu.invalid, 700, Bogus Lane, Bogus City, UT, 65879
```

Result Output Format:

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

Line 1: First and last name searched for.

Line 2: The email address found.

Line 3: Street number and name.

Line 4: City, state and zip code.

Sample Output:

```
Bo Gus
bogus@bogus.edu.invalid
700 Bogus Lane
Bogus City, UT 65879
```

Implementation Issues:

File I/O:

To export 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.

Object Serialization:

In lab, you did an exercise where you stored a list of objects to a data file using **ObjectOutputStream** and then read them back into the program using **ObjectInputStream**. You are to use this method for saving objects to a file to save the dictionary and to read them back in to the dictionary in memory.

Reading Input from the Keyboard:

In order to get the personal 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**.

How to Complete this Project:

1. Revise your UML design from the lab session. Be sure to clearly write your name and the names of your group members and "Project 3" on the cover sheet. **Remember to not include any personally identifying information on your project**. Make sure to keep a copy of this when you turn it in.
2. Create the classes and methods specified in your design, but do not put code in the methods. Add the required documentation to your classes and methods as specified in the documentation requirements posted on the class website. This is called "stubbing" your classes and methods.
3. Run your stubbed Java files through Javadoc as described in the Lab #2 slides. This will create a set of HTML files in a directory named "docs" under your project directory.
4. Implement the design you have developed by coding each method you have defined as well as any others you have left out of your design. As you do this, make sure to modify and annotate the changes to your design on your UML and properly document all new code. A good approach to the implementation of your project is to follow the project's milestones in the order they have been supplied.
5. Test your program and fix any bugs.
6. Once you have completed the project and are ready to submit it for grading, create a new set of Javadoc files using Eclipse and inspect them to make sure your detailed design is properly documented in your source code.

Extra Credit Features:

You may extend this project with more search features for an extra 5 points of credit. Think of ways to enable a wider range of data fields, such as multiple email addresses for the same person, while ensuring that only a single entry in the address book exists for a single combination of first and last name.

To receive the full five points of extra credit, your extended features must be novel (unique) and must involve effort in the design of the integration into the project and the actual coding of the feature. Also, you must indicate on your final UML design, the portions of the design that support the extra feature(s); and you must include a write-up of the feature(s) in your milestones file. The write-up must indicate what each feature is, how it works, how it is unique, and the write-up must cite any outside resources used.

Due Dates and Notes:

Your revised design and detailed Javadoc documentation are due on **Thursday, March 12th**. 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, March 12th**.

The final version of the project is due on **Thursday, April 2nd**. 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, April 2nd**.

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, your write-up cover sheet 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