Project #2

Computer Science 2334
Spring 2011

User Request:

"Create a sortable and searchable place data 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 annual demographic data, one to store data on metropolitan areas, one to store a collection of metropolitan areas, one to store data on counties[1], and one to store a collection of counties. Note that you should create any additional classes (abstract and/or concrete) and/or interfaces you deem necessary to arrive at a good design.	*10 points*
4.	Implement both the **Comparable** and **Comparator** interfaces to compare one place's data to another's.	*10 points*
5.	Use a **List** to store, retrieve, and display data related to places as described below.	*15 points*
6.	Use the `sort()` and `binarySearch()` methods from the **Collections** class to sort and search for data related to the description below.	*20 points*
►	Develop and use a proper design. (See Milestone 3, above.)	*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 collection of annual demographic data on places. We will call this application a "sortable and searchable place data system" or "SSPDS," since the primary features of this system are its sortability and searchability. Note that much of the code you write for this program could be reused in more complex applications, as we will see in later assignments.

Your software will first ask the user for the name of the data file where the place data is stored. This should be done using the technique described in the section below on reading input from the keyboard. It will then read in the specified data file and store the data into the SSDPS. Each line in the data file consists of a place name, a state[2] name[3], an indication of whether the place is a metropolitan area or a county, an STCOU number if the place is a county, and zero or more years of annual demographic data. The annual demographic data itself will consist of pairs of year and population. The format of this file is described below under Input Format.

1. We will use the term "counties" throughout this assignment to refer to the administrative divisions that within the USA are typically called "counties" but are known as "boroughs" (in AL) or "parishes" (in LA).
2. We will use the term "state" throughout this assignment to refer to the 50 states of the USA, four of which are officially known as commonwealths, plus its territories and the District of Columbia.
3. Actually, as shown in the examples, the file will contain two-letter postal abbreviations for states, rather than state names. Nonetheless, we will refer to them as state names in this assignment.

Once the data is loaded from the file, your program will enter a loop where it asks the user for criteria on which to sort the data. As with the file name, this information should come from the keyboard using the technique described below. The possible sorting options the user can enter are 'P' for place name, 'S' for state name, 'PS' for place name then state name, 'SP' for state name then place name, 'N' for STCOU number, and 'R' for random. Alternately, the user may choose to enter 'PC' for print to console, 'PF' to print to a file, or 'S' for search. If the user chooses either print option, the place data will be printed in its current order (whatever that may be, given the last sort option) and in the format described below under Output Format. (If the print option is 'PF' the user will also be prompted for an output file name.) If the user chooses search, he or she will be prompted for the place name and state name of an place on which to search and your program will search for and display the data on that place. (You may assume that the combination of place name and state name is unique in the SSPDS.) The final option available to the user is 'Q' for quit. If the user chooses quit, your program will thank him or her for running the program and exit without errors.

Learning Objectives:

Sorting and Searching:

Sorting data 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 place data based on place name and state name. If the data structure holding the place 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 these names, 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, you will measure the amount of time the system uses to find a place based on place and state name, given the ordering of the data in the SSPDS. **Do not count the time the system uses for reading in the data or carrying out other activities, like waiting for the user to provide input.** Sort the data using one ordering (such as 'P') then search for five different place and state name combinations, record the values for the resulting search times, repeat this for each possible ordering of the data, then present them in a simple table using the format shown below. (You may need to adjust spacing for long names.)

	1.	2.	3.	4.	5.
Place Name:					
State Name:					
Order	Search Time	Search Time	Search Time	Search Time	Search Time
P					
S					
PS					
SP					
N					
R					

Note that some of the sort options are not related to the search terms that the user will be providing or do not define a unique ordering. For example, random (one of the possible sort options) is not related to the name of the place searched for (the search term). Moreover, the ordering defined by this sort option is not unique. (Each time the random option is selected, the same list may be placed in a different order, so

the ordering of these places based on this criterion is arbitrary.) For these situations, you will have no choice but to search the collection linearly. On the other hand, when the sort option is related to the search terms and does define a unique ordering, a binary search is preferred and should be used. For which sort option(s) is it appropriate to use a binary search (P, S, PS, SP, N, or R)? *Explain your answers.* For which sort option(s) is it *not* appropriate to use a binary search? *Explain your answers.*

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

Note that the 'PS' sort option sorts based on place name then state name which means that the names Fort Dodge, IA; Fort Dodge, KS; and Iowa City, AI would be ordered: (1) Fort Dodge, IA; (2) Fort Dodge, KS; and (3) Iowa City, IA because the place name "Fort Dodge" comes before the place name "Iowa City" and the state name "IA" comes before the state name "KS." Ordering these same places based on the 'SP' sort option would give the order (1) Fort Dodge, IA; (2) Iowa City, IA; and (3) Fort Dodge, KS because the state name "IA" comes before the state name "KS" and the place name "Fort Dodge" comes before the state name "Iowa City." (Note that other data fields have no effect in either ordering.)

Note that each collection can have at most one *natural ordering*. You should determine an appropriate natural ordering for places (which must implement **Comparable**) and define the `compareTo()` method(s) to use that ordering. The other sort options will need to be implemented using `compare()` methods that come from implementing **Comparator**.

Input/Output Formats:

Input Format

Each line in the place data file consists of a place name, state name, an indication of whether the place is a metropolitan area or a county, an STCOU number if the place is a county, and zero or more years of annual demographic data. The annual demographic data itself will consist of pairs of year and population. Note that the annual demographic data pair (year and population) should be stored in an object of the annual demographic data class, and each annual demographic data object should then belong to a place. Also, note that each type of place should be able to have an arbitrary number of annual demographic data entries associated with it. Further, note that we will use two *very similar but not identical types of place* in this assignment: metropolitan areas and counties. You should think carefully about how these types of objects (metropolitan areas and counties) will be related to one another in your design.

In the place data file, each line contains all of the data on a single place. Within each line, the data is ordered place name, state name, then either "Metropolitan Statistical Area" (to indicate that the place is a metropolitan area) or "County or equivalent" (to indicate that the place is a county). If the place is a county, the next entry will be a five digit STCOU number. The STCOU number is a number that combines state (ST) and county (COU) identification numbers. The first two digits of the number specify the state (e.g., 01 for "AL") and the remaining three specify the county. The remaining fields are pairs of numbers in which the first number of each pair is a year and the second number is a population. All fields of a single line are separated from each other by a comma and a space. For example:

```
Abilene, TX, Metropolitan Statistical Area, 2000, 160108
Callahan County, TX, County or equivalent, 48059, 2000, 12917, 2002, 12800
```

The first of these lines gives data for Abilene, TX, which is a metropolitan area that had a population of 160,108 in 2000. The second of these lines gives data on Callahan County, TX (STCOU number 48059), which is a county that had a population of 12,917 in 2000 and a population of 12,800 in 2002.

Output Format:

The text written out for each place must conform to the following output format.

Line 1: Place name, state name [STCOU number (if applicable)].
Line 2: Year, population (for the earliest year found in the data, if any).
Line 3: Year, population (for the second earliest year found in the data, if any).
Line 4: Year, population (for the third earliest year found in the data, if any).
…
Line n: Year, population (for the last year found in the data, if any).
Line n+1: Blank line.

Sample Output:

```
Abilene, TX
2000, 160108

Callahan County, TX [48059]
2000, 48059
2002, 12800
```

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 -- did it work?");
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 place data 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, that is, 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:

Preliminary Design:

1 During the lab session and in the week following, you should work with your partner(s) to determine the classes, variables, and methods needed for this project and their relationship to one another. This will be your preliminary design for your software.

1.1 Be sure to look for nouns in the project description. More important nouns describing the items of interest to the "customer" should probably be incorporated into your project as classes and objects of those classes. Less important nouns should probably be incorporated as variables of the classes/objects just described.

1.2 Be sure to look for verbs in the project description. Verbs describing behaviors of the desired objects and the systems as a whole should probably be incorporated into your project as methods.

1.3 Be sure to use UML class diagrams as tools to help you with the design process.

2 Once you have completed your UML design, create Java "stub code" for the classes and methods specified in your design. Stub code is the translation of UML class diagrams into code. It will contain code elements for class, variable, and method names; relationships between classes such as inheritance, composition, and aggregation as appropriate; variable types; and method signatures. Stub code does not, however, contain method bodies. Because we are using object-oriented design, in which the inner workings of one class are largely irrelevant to the classes with which it interfaces (that is, we are using encapsulation), we do not need to complete the implementation of the classes until the design is completed.

3 Add comments to your stubbed code as specified in the documentation requirements posted on the class website. Run your commented stubbed code 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 At the end of the first week, you will turn in your preliminary design documents (see ***Due Dates and Notes***, below), which the TA will grade and return to you with helpful feedback on your preliminary design. **Please note:** You are encouraged to work with the instructor and the TAs during their office hours during the design week to get feedback throughout the design process as needed.

Final Design and Completed Project

5 Using feedback from the instructor and TAs as well as your own (continually improving) understanding of OO design, revise your preliminary UML design.

6 Make corresponding changes to your stub code, including its comments.

7 Implement the design you have developed by coding each method you have defined. A good approach to the implementation of your project is to follow the project's milestones in the order they have been supplied. If you find that your design does not allow for the implementation of all methods, repeat steps 5 and 6.

8 Test your program and fix any bugs.

9 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 final 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 searches to be used, such as searching based on state or year or population range or regular expression or wild cards. Alternatively, think of ways to decompose the class for place data into logical subclasses. You could also revise user interface elements. If you revise the user interface, you **must** still read the file name from the program arguments and the place list from the text file.

To receive the full five points of extra credit, your extended feature must be novel (unique) and it must involve effort in the design and integration of the feature 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 the feature is, how it works, how it is unique, and the write-up must cite any outside resources used.

Due Dates and Notes:

The electronic copy of your preliminary design (UML, stub code, and detailed Javadoc documentation) is due on **Wednesday, February 23rd**. 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 and a hardcopy of the stubbed source code at the **beginning of lab on Thursday, February 24th**.

The electronic copy of the final version of the project is due on **Wednesday, March 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 and a hardcopy of the source code at the **beginning of lab on Thursday, March 3rd**.

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 on the cover sheet. 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 cover sheet, 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 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.