Project #3

Computer Science 2334

Spring 2014

This project is group work. Each group must have at least two members.

User Request:

"Create a sortable and searchable geographic data system with places of interest that uses text and binary input and output and a graphical data display."

Milestones:

1.	Implement **Serializable** for the classes necessary to save and load all geographic data.	*10 points*
2.	Use object serialization to save and load the geographic to and from a binary file.	*15 points*
3.	Implement a simple graphical display for showing counts of data.	*25 points*
4.	Create appropriate classes to store information on places of interest.	*10 points*
5.	Use **LinkedHashMaps** to save to and retrieve information on places of interest.	*10 points*
►	Develop and use a proper design. (See Milestone 3, above.)	*15 points*
►	Use proper documentation and formatting.	*15 points*

Description:

An important skill in software design is extending the work you have done previously. For this project you will rework Project #2, implementing object serialization for input and output, using the Java **LinkedHashMap** class, and adding a graphical display. To help you locate the modifications from Project #2 to Project #3, the changes between the instructions of the two projects are highlighted here in yellow.

People routinely delineate parts the world into different kinds of geographic regions, such as cities, countries, and continents. Various kinds of regions have similar measurable characteristics such as population size and area. However, different kinds of regions also have different characteristics – for example, continents contain countries but cities do not contain countries. For this project, you will create a system to keep track of a few common types of geographic regions. In particular, it will keep track of cities, countries, and continents. For all of the geographic regions, it will keep track of population size and area. In addition, it will keep track of latitude, longitude, and elevation for cities.

Besides the types geographic regions described above, this project will include another geographic region – places of interest. A *place of interest* is a geographic feature that has an area greater than ½ square mile (i.e., that would round off to 1 square mile or more) but that does not have a substantial permanent human population (so it would not be considered a city). Example places of interest include lakes, forests, mountains, and large parks (such as national parks). For each place of interest, your system will keep track of its name, a brief description of the type of place it is (e.g., "lake" or "forest"), its area, and the country (or countries) where it is located.

As with Project #1, you will put together several techniques and concepts learned in CS 1323 and some new techniques to make an application. This application will allow users to search through data on geographic regions. We will call this application *GeoGrapher*. 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 names of either four text data files or one binary data file where geographic data is stored. (For the text data files, there will be one for continents, one for countries, one for cities, and one for places of interest. For the binary data file, all geographic data will be placed in the same file.) This should be done using the technique described in the section below entitled Reading Input from the Keyboard. It will then read in the specified text or binary data file(s) and store the data into GeoGrapher. Each line in each text data file describes a particular geographic region. The formats of these text files are described below under Text Input and Output Format.

Once the data is loaded from the file(s), your program will enter a loop where it repeatedly asks users several questions: As with the file names, user input to answer these questions should come from the keyboard using the technique described in the section below entitled Reading Input from the Keyboard.

The first question all users will be asked is “What type of geographic data are you interested in?” The user will select a numeric value between 1 and 6 with the following meaning for each: ‘1’ all continents, ‘2’ all countries, ‘3’ all cities, ‘4’ all places of interest, ‘5’ all countries within a given continent, and ‘6’ all cities within a given country. If the user chooses 5, your program will ask for the name of a continent. If the user chooses 6, your program will ask for the name of a country.

The next question all users will be asked is “How should the data be sorted?” The user will enter an ‘AR’ for area, ‘PO’ for population (not valid for places of interest), ‘LA’ for latitude (only a valid option for all cities or all cities within a given country), ‘LO’ for longitude (also only a valid option for all cities or all cities within a given country), ‘EL’ for elevation (also only a valid option for all cities or all cities within a given country), ‘LE’ for lexicographic (“alphabetical”) order, and ‘RA’ for random.

The next question all users will be asked is “How should that data be output?” The user will enter ‘PS’ for print to the screen, ‘PF’ for print to a file, ‘SP’ for search for a particular region, or ‘GD’ for graphical display. If the user chooses either print option, then all of the geographic data of the type selected will be printed in its current order (whatever that may be, given the last sort option) and in the format described below under Text Input and 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 name of a region on which to search, then your program will search for and display the data on that region. (You may assume that the region name is unique in GeoGrapher.) If no region with that name appears in the data searched, the user will be informed of that fact. If the user chooses graphical display, GeoGrapher will display a graphical display of the selected data as described below under Graphical Display.

After the data is output, the final question all users will be asked is “Do you wish to continue?” The user will choose ‘C’ for continue or ‘Q’ for quit. If the user chooses to continue, GeoGrapher will loop back to the first question. If the user chooses to quit, your program will thank him or her for running the program and exit without errors.

For all of the questions, if the user enters an invalid option, your program will inform the user of the error and ask the question again.

So, for example, if the user first selects ‘1’ for continent, then ‘LE’ for lexicographic, then ‘PS’ for print to screen, GeoGrapher would print out Africa, Antarctica, Asia, Europe, North America, Oceania, and South America, in that order (assuming that all of those continents appeared in the input data file). As a second example, if the user first selects ‘4’ for all countries within a given continent, then enters ‘Oceania’ as the continent, then ‘PO’ for population, then ‘PS’ for print to screen, GeoGrapher would print out Nauru, Tuvalu, Palau, Marshall Islands, Kiribati, Tonga, Federated States of Micronesia, Samoa, Vanuatu, Solomon Islands, Fiji, New Zealand, Papua New Guinea, and Australia in that order (assuming that all of those countries appeared in the input data file with populations as given in your sample data file).

Graphical Display:

Producing graphical displays of information can be very useful to users. Therefore, your program will have the ability to display bar charts and maps to the user to display the data. When the user selects option 'GD' for Graphical Display, as described above, a bar chart or map will be generated and presented to the user. The type of bar chart or map will be determined by the type of data chosen by the user in response to the first question and the sorting order of the data chosen by the user in response to the second question.

If the user has chosen option 1, 2, 3, or 4 (all continents, all countries, all cities, or all places of interest) and to sort the data by area, GeoGrapher will display a chart with one bar for each region in the selected data, with the height of each bar proportional to the area of the region and arranged left to right across the chart, such that the tallest bar is on the left and the shortest bar is on the right. Each bar should be labeled below with the name of the corresponding region and immediately above with that region's area. If the user has chosen option 1, 2, or 3 and to sort by population, GeoGrapher will display a similar chart but using population rather than area.

If the user has chosen option 5 or 6 (all countries within a given continent or all cities within a given country) and to sort the data by area, GeoGrapher will display a single bar composed of several smaller segments. There will be one segment in the bar to correspond to each of the subregions in the selected region (e.g., the countries within the selected continent) and the height of each segment will be proportional in height to the area of the corresponding subregion. These segments will be arranged progressively from largest on the bottom to smallest on the top. Each segment will be labeled on the left with the area of the subregion and on the right with the name of the subregion.

If the user has chosen option 3 or 6 (all cities or all cities within a given country) and to sort by latitude or longitude, GeoGrapher will display a plate carrée map projection of the world with points plotted on it for all of the cities in the selected data. Each point should be labeled immediately above it with the name of the corresponding city.

Each graphical display should have a title at the top clearly indicating the data it represents (e.g., "Population of All Cities in Database" or "Location of All Cities in Database"). Other details of the graphical displays are up to you (colors, etc.).

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 information about a city based on its name. If the data structure holding the city 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.

To observe this efficiency gain, you will measure the number of comparisons the system uses to find a city based on name, given the ordering of the data in GeoGrapher. Sort the data using one ordering (such as 'EL') then search for five different city names, record the values for the resulting number of comparisons, 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 titles.)

```
                  1.                2.                3.                4.                5.
Name:
```

Order	Search Time	Search Time	Search Time	Search Time	Search Time
-----	-----------	-----------	-----------	-----------	-----------
AR					
PO					
LA					
LO					
EL					
LE					
RA					

Note that some of the sort options do not define a unique ordering. For example, random (one of the possible sort options) is not related to the name of the city 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 the cities 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 (AR, PO, LA, LO, EL, LE, or RA)? *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 a collection in Java can have at most one *natural ordering*. You should determine an appropriate natural ordering for each type of region (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**.

Hash Maps:

You have already dealt with lists as a basic data structure for storing and retrieving data. Hash tables are an alternate way to quickly store and retrieve data. Java provides the **LinkedHashMap** class (among others) which has this functionality. In this project you will use a **LinkedHashMap** for saving and retrieving information on collections of places of interest. The keys in this map will be place names. The values will be the places themselves. Note that we are using a **LinkedHashMap** so that it can retain an ordering based on area. You should consider how to get the data into the **LinkedHashMap** in that order.

Once you have the data in the **LinkedHashMap** with the links preserving the order by area, you should conduct five linear searches through this data structure based on areas and a five lookups using keys (place names). Record the system time before and after these searches/lookups to see how quickly they happen. Construct a simple table with this information and explain what this tells you about linear searches and hash table lookups.

Input/Output Formats:

Text Input and Output Format

Each line of each data file describes a particular region. For the continent file, each line contains a continent's name, followed by a continent and a space, followed by the approximate population of the continent, followed by a comma and a space, followed by the approximate area of the continent in square miles.

Examples:

```
Antarctica, 4490, 5300000
North America, 542056000, 9460000
```

For the country file, each line contains a country's name, followed by a comma and a space, followed by the approximate population of the country, followed by a comma and a space, followed by the approximate area of the country in square miles, followed by a comma and a space, followed by the continent where the country is located.

Examples:

```
United States, 313232044, 371891, North America
Mexico, 113724226, 761602, North America
```

For the city file, each line contains a city's name, followed by a comma and a space, followed by the approximate population of the city, followed by a comma and a space, followed by the approximate area of the city in square miles, followed by a comma and a space, followed by the country where the city is located. For some (though not all) cities, the country name will be followed by a comma and a space, followed by a reference latitude, followed by a comma and a space, followed by a reference longitude, followed by a comma and a space, followed by a reference elevation. (I am using the term “reference” with regard to latitude, longitude, and elevation because cities have areas and therefore do not have single latitude, longitude, and elevation values.)

Examples:

```
Los Angeles, 11789000, 1668, United States
Mexico City, 17400000, 800, Mexico, N19.4271, W99.1276, 2216
```

In reading these three files, you may assume that the country file will not contain a reference to any continent not in the continent file and that the city file will not contain a reference to any country not in the country file.

Implementation Issues:

Text File I/O:

To perform output to a text 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 information from the user for this project, 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 Make a list of the nouns you find in the project description that relate to items of interest to the "customer." Mark these nouns as either more important or less important. 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. This list will be turned in with your preliminary and final designs long with your other design documents.

1.2 Make a list of the verbs you find in the project description that relate to items of interest to the "customer." Verbs describing behaviors of the desired objects and the systems as a whole should probably be incorporated into your project as methods. This list will be turned in with your preliminary and final designs long with your other design documents.

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 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 (except for return statements for methods that return values or object references – these should return placeholders such as `null`). 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 any class until the design is complete.

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 Create unit tests using JUnit for all of the non-trivial units (methods) specified in your design.

There should be at least one test per non-trivial unit and units with many possible categories of input and output should test each category. (For example, if you have a method that takes an argument of type `int` and behaves differently based on the value of that `int`, you might consider testing it with a large positive `int`, and small positive `int`, zero, a small negative `int`, and a large negative `int` as these are all likely to test different aspects of the method.)

5 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

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

7 Make corresponding changes to your code, including its comments.

8 Make corresponding changes to your unit tests.

9 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 6, 7, and 8.

10 Test each unit as it is implemented and fix any bugs.

11 Test the overall program and fix any bugs.

12 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.

13 Submit your project (see ***Due Dates and Notes,*** below).

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 population or elevation or regular expressions or wild cards. Alternatively, think of ways to decompose one of the region classes into logical subclasses. You could also revise user interface elements. If you revise the user interface, you **must** still read the file names from the keyboard and the geographic data from the text files.

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. If you create any non-trivial units in your extra credit work, you must create appropriate unit tests for them.

Due Dates and Notes:

The electronic copy of your preliminary design (UML, stub code, detailed Javadoc documentation, and unit tests) is due on **Wednesday, March 5th**. Submit the project archive following the steps given in the submission instructions **by 10:00pm**. Submit your UML design on *engineering paper* or a hardcopy using UML layout software at the **beginning of lab on Thursday, March 6th**.

The electronic copy of the final version of the project is due on **Wednesday, March 26th**. Submit the project archive following the steps given in the submission instructions **by 10:00pm**. Submit your final UML design on *engineering paper* or a hardcopy using UML layout software at the **beginning of lab on Thursday, March 27th**.

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 methods in your program and one method 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.

When zipping your project (or design) for submission, be sure to follow the instructions carefully. In particular, before zipping the project be sure to

- **place additional files (such as UML diagrams, cover sheets, and milestones files) within the "docs" directory inside your Eclipse folder for the given project and be sure that Eclipse sees these files (look in the Package Explorer and hit Refresh if necessary), and**
- **compress all files into a .zip format. The formats .rar and .7z will no longer be accepted. Also, when submitting the initial design make sure that the UML is in one of the following formats: .png .jpg or .pdf. Custom formats such as .uml or .dia are NOT acceptable. If you are unsure how to export the file in that format, take a screen-shot of the diagram and attach that.**