# Project 3

*Computer Science 2334*

*Spring 2011*

## *User Request:*

*"Create a fast sortable and searchable place data system with text and binary input and output and a graphical display."*

## *Milestones:*

| | | |
|---|---|---|
| 1. | Implement **Serializable** for the list of places and for any other classes necessary. | *10 points* |
| 2. | Utilize object serialization to save and load the list of places to and from a binary file. | *15 points* |
| 3. | Implement a simple graphical display for showing the relative frequency of data. | *25 points* |
| 4. | Use the **LinkedHashMap** class to save to and retrieve from the list of places in memory. | *10 points* |
| 5. | Override the `hashCode()` and `equals()` methods of **LinkedHashMap** to allow for searching based on partial information. | *5 points* |
| 6. | Convert the place list between a **LinkedHashMap** and a **List** representation (to allow for sorting). | *5 points* |
| ► | Develop and use a proper design. | *15 points* |
| ► | Use proper documentation and formatting. | *15 points* |

## *Description:*

An important skill in software design is extending the work you have done in a previous project. 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.

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 text or binary 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 text or binary file and store the data into the SSDPS. Each line in the text file consists of a place name, a state[1] name[2], an indication of whether the place is a urban 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 Text Input/Output Format.

---

1. 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.
2. 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. (Because urban areas do not have STCOU numbers, when sorting based on STCOU your comparator for STCOU numbers should always return 0 for two urban areas and always claim that the urban area should come after the county when an urban area and a county are compared.) Alternately, the user may choose to enter 'PC' for print to console, 'PF' to print to a file, or 'F' for find. 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 exactly the same format as that of the input text file as described below under Text Input/Output Format. (If the print option is 'PF' the user will also be prompted for an output file name.) If the user chooses find, he or she will be prompted for the place name and state name of a place to find and your program will search for and display the data on that place, if found. Otherwise, your program will display a message informing the user that the place was not found. (You may assume that the combination of place name and state name is unique in the SSPDS.) The user will also have two options available related to binary input and output. These are 'LD' for load and 'SV' for save. If the user chooses one of these options, he or she will be prompted for the input or output filename as appropriate and the SSPDS will load or save the list of place objects using object serialization as described below under Object Serialization. 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.

Graphical Display:

Producing graphical displays of information can be very useful to users. Therefore, each time a place is found using the find function, your program will offer to produce a graphical display of the population of the place over time. If the user accepts the offer, your program will produce what is known as a bar chart. The horizontal axis of this chart will be time (in years) and the vertical axis will be population. Other details of the bar chart are up to you (width of the bars, etc.). However, to make things easy, you may assume that the minimum number of years in the data will be 1 and the maximum will be 50.

## ***Learning Objectives:***

Hash Maps, 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 an unsorted **List**, you need to do a linear search through it to find an entry. However, if the data structure is a **List** 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.

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 add a **LinkedHashMap** (or a derived class) for saving and retrieving information on places. To retain the ability to sort your place data, you should be able to extract the collection of values from your **LinkedHashMap** and place them into a **List** that can then be sorted various ways for printing to a file or the console.

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.

Note that this time all the searching will be done with the data in the **LinkedHashMap** rather than a **List**, Be sure to use the same data file, the same places, and the same computer for searches in Project 3 as you used in Project 2, so that the data will be (at least somewhat) comparable between the two projects.

```
                  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
```

Put the table of data plus the data from Project 2 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:*

### Text Input/Output Format

Each line in the place data text file consists of a place name, state name, an indication of whether the place is a urban 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: urban areas and counties. You should think carefully about how these types of objects (urban 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 "County or equivalent" (to indicate that the place is a county) or one of a small number of terms to describe the urban area (such as "Metropolitan Statistical Area," "Metropolitan Division," and "Micropolitan Statistical Area"). If the place is an urban area, the descriptive term should be retained in the object, so that it can be written out as needed. If the place is a county, the next entry will be a four or five digit STCOU number. The STCOU number is a number that combines state (ST) and county (COU) identification numbers. The first one or two digits of the number specify the state (e.g., 1 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 urban 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.

Note that this text file format is used for both input and output as text. The Project 2 format for output is superseded by the format described here.

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 approach to save the place list to a binary file and to read it back in from a binary file.

## *Implementation Issues:*

Text File I/O:

To perform output to a text file, use the **FileWriter** class with the **BufferedWriter** class use code based on the following.

```
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, March 9th**. 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, March 10th**.

The electronic copy of the final version of the project is due on **Wednesday, March 30th**. 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 31st**.

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.

**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**
- **rename the project folder to the 4x4 of the team member submitting the project.  Note that renaming the project folder to your 4x4 *before* zipping is not the same thing as naming the zip file with your 4x4.  The latter is fine; the former is *mandatory*.**