Project #2

Computer Science 2334
Spring 2012

User Request:

"Create a sortable and searchable presidential nomination 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 classes to store data on individual electoral contests, data for individual Democratic candidates, data for a collection of Democratic candidates, data for individual Republican candidates, and data for a collection of Republican candidates. Note that you should decide, as part of the design process, whether to use the same or separate classes for Democratic and Republican candidates and their collections. Note also 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 candidate to another. *10 points*
5. Use a **List** to store, retrieve, and display data related to candidates 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:

As some of you may be aware, there will be an election for President of the United States (POTUS) this coming November. Moreover, the Democratic and Republican parties have complex and arcane nomination systems which they use to determine their own party's Presidential nominee for the November election. 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. This application will allow users to search through data on people running for either the Republican or Democratic nomination for President of the United States. We will call this application POTUSnom. 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 candidate 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 POTUSnom. Each line in the data file consists of a candidate name, that candidate's city of birth, state of birth, date of birth, party affiliation (Republican or Democratic), and a list of electoral contests (primaries and/or caucuses) in which that candidate has participated and the number of votes and delegates he or she earned in each contest. The format of this file is described below under Input Format.

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 'FN' for first name then last name, 'LN' for last name then first name, 'DC' for delegate count, 'NV' for number of votes, 'PV' for percent of votes, and 'R' for random. Alternately, the user may choose to enter 'PS' for print to screen, 'PF' to print to a file, or 'S' for search. If the user chooses either print option, the user will be asked to select 'D' for Democratic or 'R' for Republican, then candidate data from that party 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 likewise be asked to select 'D' for Democratic or 'R' for Republican, then be prompted for the first name and last name of a candidate on which to search, then your program will search for and display the data on that candidate. (You may assume that the combination of first name and last name is unique in POTUSnom.) If no candidate with that name appears in the data searched, the user will be informed of that fact. 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 candidate data based on first name and last name. If the data structure holding the candidate 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 candidate based on first and last name, given the ordering of the data in POTUSnom. **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 'DC') then search for five different first and last 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.
First Name:					
Last Name:					
Order	Search Time	Search Time	Search Time	Search Time	Search Time
FN					
LN					
DC					
NV					
PV					
R					

Note that while there seemed to be a very large number of candidates for the Republican nomination earlier in this election season, you may need to use a synthetic data set much larger than the actual field of candidates to get measurable differences in the results above.

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 candidate 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 candidates 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 (FN, LN, DC, NV, PV, 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 'FN' sort option sorts based on first name then last name which means that the names Rick Perry; Rick Santorum; and Mitt Romney would be ordered: (1) Mitt Romney; (2) Rick Perry; and (3) Rick Santorum because the first name "Mitt" comes before the first name "Rick" and the last name "Perry" comes before the last name "Santorum." Ordering these same candidates based on the 'LN' sort option would give the order (1) Rick Perry; (2) Mitt Romney; and (3) Rick Santorum because the last name "Perry" comes before the last name "Romney" which comes before the last name "Santorum." (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 candidates (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 data file consists of a candidate name, that candidate's city of birth, state of birth, date of birth, party affiliation (Republican or Democratic), and a list of electoral contests (primaries and/or caucuses) in which that candidate has participated. The contest data itself will consist of triples of state abbreviation, votes won, and delegates won. All fields of a single line are separated from each other by a comma and a space. For example:

```
Barack Obama, Honolulu, HI, 04/08/1961, Democratic, NH, 48970, 35
Newt Gingrich, Harrisburg, PA, 17/06/1943, Republican, NH, 23421, 0, SC, 243172, 23
```

The first of these lines gives data for Barack Obama who is a candidate for the Democratic nomination who won 48,970 votes in the New Hampshire primary and has secured 35 delegates from that state. The second of these lines gives data on Newt Gingrich who is a candidate for the Republican nomination who won 23,421 votes in the New Hampshire primary and secured 0 delegates from that state and who also won 243,172 votes in the South Carolina primary and secured 23 delegates from that state.

Note that the contest data triple should be stored in an object of the contest data class, and each contest data object should then belong to a candidate. Also, note that each candidate should be able to have an arbitrary number of contest data entries associated with him or her. Further, note that we will use two *very similar but not identical types of candidates* in this assignment: Democratic and Republican. You should think carefully about how these types of objects will be related to one another in your design.

Output Format:

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

Line 1: Last name, first name (one letter party designation in parentheses)
Line 2: City of birth, two letter postal abbreviation for state of birth
Line 3: Birthdate in the form day of month, three letter month abbreviation, year
Line 4: Blank line
Line 5: State of first contest in list (spelled out)
Line 6: Number of votes won in first contest in list
Line 7: Number of delegates won in first contest in list
Line 8: Blank line
Line 9: State of second contest in list (spelled out)
Line 10: Number of votes won in second contest in list
Line 11: Number of delegates won in second contest in list
…
Line 4n: Blank line
Line 4n+1: State of last contest in list (spelled out)
Line 4n+2: Number of votes won in last contest in list
Line 4n+3: Number of delegates won in last contest in list
Line 4n+4: Blank line.

Sample Output:

```
Gingrich, Newt (R)
Harrisburg, PA
17 Jun 1943

New Hampshire
23421 votes
0 delegates

South Carolina
243,172 votes
23 delegates
```

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 candidate 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 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 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 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 age or state of birth or regular expression or wild cards. Alternatively, think of ways to decompose the class for candidate 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 candidates 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 **Thursday, February 23rd**. Submit the project archive following the steps given in the submission instructions **by 10:00am**. 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 23rd**.

The electronic copy of the final version of the project is due on **Thursday, March 1st**. Submit the project archive following the steps given in the submission instructions **by 10:00am**. 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 1st**.

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.