Collections, Files & LINQ

In the past week we learned about C# collections, lambda expressions, and LINQ. We discovered that together with some I/O functionality (input/output – or reading and writing files), these constructs provide for some very elegant, efficient, and powerful code construction. This assignment will explore ways we can manipulate, organize, and store large amounts of data using control loops, Lists, files, and LINQ. Included with this assignment is fully-functioning C# source code as well as a 9mb csv data file (us-50000.csv) containing 50,000 people we will use to construct a *database of contacts. The .csv file will be in the bin folder included in the source zip.*

The program you will be modifying for this assignment is a database-like program. Many of you have not yet had any real experience using a database. On a very basic level, this program will give you a good idea of what a database is and what it does.

To complete this assignment you must use the provided source code! There may be cases where code as-is may not work exactly the way you need it to. In those cases, you may copy code that comes close to what you need into a new method/function and then modify the code so that it will work. I expect you to be able to examine the code here and then understand how to modify it so that it works according to the requirements in this assignment. **You must be *willing* to use some <u>creativity</u> and <u>thoughtfulness</u> to complete this assignment**. Be *resourceful*; all good programmers are. You are *not allowed* to copy code as-is from the Internet or any other source other than what is provided to you. If I see that anywhere in your work, you will receive a ZERO for the entire assignment. Also, *no copying code from each other*.

Included Source Code Files

Program.cs

By now you should be familiar with this class and its Main() method. For this assignment Main is a full-blown program flow that loads 50,000 contacts into memory and controls the main menu for the program. *<u>Study Main carefully</u>*! Run through all of the program's menu selections to understand how the program works. Set a break point in main and in other locations, and then step through the code to learn how it works. I showed you how to set and use breakpoints in class; if you cannot remember, or need more details, then *Google "setting **breakpoints** in Visual Studio"*.

Contact.cs

This class holds data for each contact record we load from us-50000.csv. We are dealing with a list of 50,000 contacts for this assignment (that means that there will be 50,000 contact objects in memory while running this program). This class also contains several static methods used for entering, loading, and saving contacts. You will be modifying this class to also include functions for editing and deleting existing contacts. These functions will allow the user to modify the list of contacts. The class will also need a way to *validate* certain member fields such as *email*, *phone*, and *Internet URLs*.

Display.cs

There are some standard methods provided for you in Display.cs so that you may use the console window to display information. This is also true for displaying program menus as well. Keep in mind that when you display a list of contacts to the console, the user may need to scroll the console window upwards to see all previous contact records that were sent to the console. Use the Display.Pause() method to send messages or instructions to the user while waiting for user input before continuing. The Pause method also returns anything the user pressed into the console window before pressing Enter/Return. You should *create and add* *any* ***new methods*** to this class when you discover you need a new way to display information in the console window. Review this class carefully; it should contain most of the methods you will need for displaying information to the user.

Validation.cs

This class contains several functions for validating string data. It will be used to validate user input. We can test values like numbers, email, urls, and (US) phone numbers. **When the user is adding or modifying a contact, the information he/she enters for certain fields will be first validated before assigning the value to the contact.** Anything having to do with data validation should be in this class – including any new validation functions you might write.

ContactQuery.cs

This class holds all query-based methods. All new queries you create will be placed here.

The Program:

The first time you run the program it will create a new *binary* data file of 50,000 contacts by first reading the contact records from a *text-based* .csv file. Anytime you run the program thereafter, it will not need to create a new file; it will load the existing binary data file into memory. Examine the first few lines in Program.cs Main() method to see how that works.

This program is 100% menu-driven from the console window, and is self-explanatory to just about anyone who has ever used a software program before. This means that from a user's perspective, this program is easy to understand and use. There are four menus in this program: the **main** menu, the **sort** menu, the **edit** menu, and the **query** menu. All of these menus are constructed and controlled through re-usable C# methods already provided for you.

Pay attention to **<u>all</u>** of the methods in this program. Much *of the code you need* can be found or inferred from the program itself. **You should be able discover *everything* you need by studying what this program already does**. Even if the exact code cannot be found, there are hints everywhere in this source code that may help you to discover the solutions you're looking for**. Pay close attention to the code *comments*** as well – they can be *very* telling.

This program is fairly elaborate, and I designed it to be so. As programmers, you will oftentimes start out in your new job by working on existing code to an existing product. Even if you see an approach or certain methods used in the code that you are not familiar with, you should be able to familiarize yourself with what is happening in the code after you've gone through it a few times. When you work on this assignment, you should also try to follow the same "spirit" of how the code is written; *stick with the same naming conventions and techniques that are already in play*. Also, stick with the precedence already set by the program. The new features you add to this program should ***seem*** to the user to be a natural extension of how the program already works.

Your Assignment

Modifying the Contact Class:

We want the Contact class to fully encapsulate its data. This means that we will want to ensure certain fields hold valid values. We want to use the **`Validation`** class to validate phone number, email, and web url fields. First, add a new *bool read-only* property to the Contact function called DataIsValid. It will be true, if **<u>all</u>** the data in the object is valid; otherwise, false.

So, when one of these fields/properties is set, use the Validation class to ensure the value is actually valid. If a property fails validation, set DataIsValid = false. DataIsValid should ONLY be true when all fields (phone, email, and url) are valid. To know which fields are invalid, add the following read-only bool fields to Contacts class: Phone1IsValid, Phone2IsValid, EmailIsValid, UrlIsValid. When all of these fields are true, so, too will DataIsValid be true.

Create Contact Queries:

Look at the **`SwitchToUserQueryMenu`**() method in Program.cs. You need to finish building the queries for each of the options provided in the query menu. All of your queries shall be added to the **`ContactQuery`** class. The first query in the menu is already provided for you. In addition to that, the `ContactQuery` class contains two existing query methods. Add all required query methods to the `ContactQuery` class and then modify the `SwitchToUserQueryMenu` () method to use those methods according to what the user selects from the menu. **Do not forget to test each query to ensure that they each return the proper results.** You will be graded on whether or not they actually work correctly.

Sorting the Contact List

In the **`SwitchToSortMenu`**() method, there are several sorting options available to the user. Finish the code for those sorting options. Following the code comments.

Adding New Contacts:

Most of this functionality is finished; however, there are a few loose ends that you need to tie up. From the main menu, select 2 to modify the contact list. You are then brought to the "Modify Contact List" menu. Select A to add new contacts. Doing so will call the **`EnterUserContacts`**`()` method found in the **`Contact`** class. Study this method so that you fully understand what is happening in it. Once you understand what the method does, and how it works, do the following to complete it (refer to the comments at the bottom of the method):

(1) **Do not allow** a new contact to be added to the list if a contact ***already exists*** with all of these combined attributes:
  a. First Name
  b. Last Name
  c. Phone1
  d. Phone2

  Think about how you can *discover* this (using a **LINQ** query), and how you would inform the user of the problem, if you do discover an existing contact with that combination of attributes. Think carefully, and be resourceful. I want to see *a smooth solution* tied into the existing program.

(2) Be sure that the calculated **ID is *unique***. In other words, if there exists a contact with the same ID, you must determine how to recalculate a new ID. How could you improve this routine with respect to how it determines an ID for a new contact? This is not something the user provides, it is something that the program automatically provides for a new contact. Once a contact is placed in the list with the ID, that ID never changes (even the edit function does not allow the user to change the contact's ID).

(3) Now that the Contacts class has the ability to validate phone number, email, and web url fields, be sure to invoke the validation before adding a new contact (or before accepting a modification to an existing contact). If the user enters invalid values for any of these fields, the user must be informed of which field(s) is invalid, and the contact ***must not be added into the contacts list*** until those values are corrected and **properly validated**.

Editing Existing Contacts

The user is allowed to edit existing contacts. The **`UserEditContacts`**() method in the `Contact` class provides the functionality for this. You need to invoke the *same validation rules* that are required when adding a new contact into the list. You don't need to consider the ID because you can assume the ID was already validated when the user was first added to the list, and you will not be changing the ID when editing a contact.

However, you must be sure that the rest of the *validation* rules as well as the ***uniqueness*** rules in #1 under "Adding New Contacts" are enforced here. If a user edits a contact, and those edits conflict with the rules, the changes must not be allowed. Since these rules are applied in at least two areas of the program (adding and editing contacts), ***consider writing validation routines in the Validation*** class that accept a contact to validate. That way you will not have *redundant* code. **Note**: *I will take away points where I discover redundant code that should be consolidated into reusable methods. ...how would you pass a contact reference to a routine for validation? This is not that difficult to figure out, if you think about everything we've covered in class to date.*

Deleting Contacts

Create the ability to delete a contact. To do this you must first allow the user to type the id of the contact he/she wishes to delete. If you find the contact, display the contact on the screen and ask the user to confirm or cancel the deletion. You can find most of the code you need to finish this task throughout the program's existing source code. *Hint*: start by looking at the `UserEditContacts`() method in the `Contacts` class.

Provide Answers for the Following Questions:

(1) How many contacts are there where last name begins with "st"?
(2) How many contacts are there where last name begin with "z"?
(3) How many contacts live in CA?
(4) List the names of the first 15 contacts with IDs between 21049 and 22000.
(5) How many contacts are there who have emails with aol.com?
(6) How many contacts are there with first names starting with "ba" while their last names start with "f"?
(7) How many contacts are there with first names starting with "t" while their last names start with "f" and who live in "MI"?
(8) Sort your contacts list by State. List the first 10 contacts you see after performing the sort.
(9) Sort contacts by state and city. List the first 10 contacts you see after performing the sort.
(10) Sort contacts by state, city, and zip. List the first 10 contacts you see after performing the sort.

Final Notes:

First, when you submit your completed assignment, **<u>DO NOT INCLUDE</u>** your **bin** or **obj** folders! This implies that *we do NOT want* you to submit *ANY* data files with your code; **NO DATA FILES**. We have our own copies of the data. If you do include anything other than your Visual studio solution, project, and source files, ***points will be reduced by 15%***

As mentioned before, the initial contacts list is built the first time you use the program. The program will use the **us-50000.csv** file under **bin/debug** to build the **MyContacts.dat** file. Whenever the user (Q)uits the program, the program *automatically* saves the contact list into MyContacts.dat. This means that any **deletions**, **additions**, and **edits** are saved with the list. In addition, any sort you performed on the list is also saved.

If you discover that you messed up your contact list, or you simply would like to reset the list to its original state, just delete **bin/debug/MyContacts.dat** *after you exit (Quit) the program*. ***Do not*** *delete bin/debug/us-50000.csv*. After deleting MyContacts.dat, restart the program. It will rebuild MyContacts.dat again. Note that if you do not select **Q** for quit from the main menu, *MyContacts.dat will not be saved*. The program only saves this file when you select quit! Stopping the program from the debugger without quitting will result in all changes to your contact list being lost. *Keep that in mind!*

Debugging is your friend. At times you will have to step through each line of code to figure out why it isn't working. With breakpoints you can pause the program in specific areas so that you can examine the ***call stack*** and ***variables*** at that location. If you missed that class, then Google it C# call stack, visual studio.

One more thing… When you are working on the code to edit a contact, and you query to be sure that your contact does not violate any rules with existing contacts, you must make sure that you *ignore* the contact you are actually working on in your query results. You may want to consider temporarily removing a contact from the list of contacts before editing it. That way, your current contact will not be included in the results from any queries you might perform. If you have no clue about what I'm referring to, maybe you will when you actually dive into the editing method. Consider this a *bonus clue*.

Extra Credit

For those who wish to **replace one of your *lowest* homework grades**, you may do the following as *extra credit*.

LINQ Groups:

(1) Create an ***entirely new menu*** that you can access from the main menu. Label it, **"Contacts: Reports"**. Add the following options to it:
- Group by City and State, Sorted by Last Name
- Report by Email Provider, Sorted by Email Domain
- Group by Zip Code, Sorted by City and Last Name
- Group by Area Code (Phone1), Sorted by Last Name

(2) Make sure your menu is fully functional – including a way to get back to the main menu
(3) When a user makes a selection, run a LINQ query that ***groups*** **and** ***sorts*** results according to the selection made.
(4) Once grouped and sorted, write out the results into a new csv file.
   a. **NOT** a binary file like the MyContacts.dat,
   b. You will create a file similar to us-50000.csv; a human-readable text file where values are separated by commas, and each row represents a different contact.
   c. For each report, create a different file.
   d. For example, for "Group by City and State, Sorted by Last Name" selection, create a file named "CityStateSortedLastName.csv.
   e. Each report must have a unique file it is written to.
(5) In each csv file, include the following fields: **Last Name, First Name, Email, Phone 1, Email, City, State, Zip**

Grouping in **LINQ** is not difficult, but requires a little more effort. Go here for more information: https://msdn.microsoft.com/en-us/library/bb545971.aspx. Remember: Grouping involves (at least) two lists in your result. Basically, grouping returns a ***list of lists***. In the example below, we have a list of people grouped by states. So each state contains a list of contacts who live there. Groups are managed *kinda* like this (pseudocode):

```
MyGroupedContacts = LINQ: {Group List by State, and Sort by Last name}

foreach (state in MyGroupedContacts)
{
        foreach (contact in state)
        {
                string myLine =
                 contact.State + ","
                + contact.Zip + ","
                + contact.LastName + ","
                + contact.FirstName
                + "\r\n"; //(line feed at very end of string)

                // write record to file
                File.AppendAllText("filename",myLine);
        }
}
```

When your routines are complete, I should be able to go to your new menu, make a selection, and then open a text file to view the results of your query. Don't forget to inform the user that the file is ready once the report is finished. Look in other areas of the program code to see how this is done, and use the pseudocode above to help you build the output file using your results.