Description: Working with LINQ

Setup

Open the Lab2 solution provided.

The solution contains the Program.cs file and two text files, LastNames.txt and FirstNames.txt.

The LastNames and FirstNames files will be used to create random *Person* objects.

Within Program.Main we will build LINQ queries to interrogate the random *Person* data.

Person

Even though the focus of this lab is to use LINQ it is always a good idea to practice creating classes.

Open the *Person* class code file.

Make sure *Person* is marked with the **Serializable** attribute. Don't worry if you don't understand how that works yet, just make sure your Person class definition looks like this:

```
[Serializable]
public class Person
```

Within the *Person* class, add the following **enum**:

```
public enum AgeGroupEnum
{
    Unknown = 0,
    Child,
    Teen,
    Adult,
    Senior
}
```

We'll use the *AgeGroupEnum* to identify the age group of a given *Person*. You'll see it in use when we define the properties.

Add the following fields to the class:

```
private long m_ID = 0;
private string m_LastName = string.Empty;
```

```
private string m_FirstName = string.Empty;
private DateTime m_DOB = DateTime.MinValue;
```

Add a **public** property for each of the fields – you can use Right Click ➔ Refactor → Encapsulate.

Next, we'll add two dependent properties. Dependent properties are properties that depend on other properties and typically only have a **get** accessor.

Add a **public int** property called *Age*. Add only a **get** accessor to the property with the following definition:

```
DateTime now = DateTime.Now;
int years = now.Year - DOB.Year;
if ((now.Month < DOB.Month) || ((now.Month == DOB.Month)
       && (now.Day < DOB.Day)))
   years--;
return years;
```

Notice that the above code takes into account whether or not the Person's birthdate has occurred yet for the current year.

Next, add the **public** *AgeGroupEnum* property called *AgeGroup*. Again, this will only have a **get** accessor. This property depends upon the *Age* property we just defined. Using **if**...**else** statements, determine the appropriate *AgeGroupEnum* value to return:

Child: Age < 13
Teen: 13 ≤ Age < 19
Adult: 19 ≤ Age < 65
Senior: Age ≥ 65

Next, add two constructors:

```
public Person()
```

AND

```
public Person(long id, string lastName, string firstName, DateTime dob)
```

Make sure you assign the parameters of the second constructor to the appropriate properties.

Finally, add an override for ToString with the following output:

```
return string.Format("{0:000-00-0000} {1,-15} {2,-15} {3:MM/dd/yyyy}",
       ID, LastName, FirstName, DOB);
```

Make sure the class compiles (do not run, just build) without any problems before moving on to the next step.

LINQ

Go to the Program.cs file. Notice that there are two methods defined within the "IO Methods" region: *OpenPersonFile* and *GeneratePeople*. These two methods depend on the *Person* class you just developed. GeneratePeople creates a file of 50,000 random Person objects. *OpenPersonFile* reads this file into a List<*Person*> object. We will use these methods to create a large dataset of Person objects. Feel free to manipulate the RECORDS constant, just keep it within reason (less than 1,000,000).

Make sure the call to *GeneratePeople*() is **uncommented** in Main and run the project. Exit the program and use Explorer to navigate to the bin\debug directory of the Lab2 project. Check that a file called *people.data* exists. This is the output of *GeneratePeople*().

Make sure the call to *GeneratePeople*() is **commented** out before proceeding so you have a consistent dataset from which to test.

GetYoungestAndOldest()

In this method you will create a LINQ query to retrieve the oldest and youngest *Person*. This query is fairly straight-forward. Create a standard query to select all items from the *people* list.

```
var query = from p in people
            select p;
```

Now modify the query to order by the *DOB* property of *p* by inserting:

```
orderby p.DOB
```

between the **from** and **select** clauses.

The *First()* method of ageQuery returns the oldest *Person*. The *Last()* method of *query* returns the youngest *Person*. Output the results.

GetNancysAndJohns()

This method will find all *Person* objects whose first name is either Nancy or John. Again, start with the simple query:

```
var query = from p in people
            select p;
```

This time you will add a **where** clause and an **orderby** clause. You need to decide the order of the clauses. If you place the **where** clause before the **orderby** clause, the entire list will be filtered prior to sorting. If you do it in the other order then the list will be sorted prior to filtering. Generally speaking, filtering is faster than sorting on large datasets. Filtering, at most, has to iterate through the entire set of records once, known as O(n) or linear time. The most efficient whole set sorting algorithm takes O(n log n) time, or loglinear time. So, for various size datasets, here are the theoretical timings:

Records	O(n) Operations	O(n log n) Operations
128	128	896
512	512	4,608
2,048	2,048	22,528
65,536	65,536	1,048,576
4,194,304	4,194,304	92,274,688

As you can see, sorting a large set of records takes many times longer than a simple filter. So, the best general approach is to filter first, then sort.

The **where** clause needs to check the *Person's FirstName* for Nancy and John:

```
where p.FirstName == "Nancy" || p.FirstName == "John"
```

The **orderby** clause will sort by *LastName*, *FirstName* and *DOB*, in that order:

```
orderby p.LastName, p.FirstName, p.DOB
```

GetAgeGroups()

This query involves grouping the results into sub groups. Using the **group**…**into** clause will create an outer set of records, each of which will contain child records grouped by their parent key. The parent records have a special field called "**Key**" which contains the value used to separate the source records into groups. For this query, we want to group by the *AgeGroup* property or *Person*. Since there are 4 age groups (not counting *Unknown*), the outer/parent result will contain 4 records. Each of these 4 records will have some number of child records from the source data. In fact, each parent record is enumerable so you can iterate through it.

Again, start with the query definition:

```
var query = from p in people
```

Now, add the following **group** clause after the **from** and clause:

```
group p by p.AgeGroup into ageGroup
select ageGroup
```

Notice that we are selecting the group this time, not the *p* record. Also, we can sort using the group's Key rather than *p*'s properties. After the **group** clause you can add an **orderby** clause to sort the resulting groups.

```
orderby (int)ageGroup.Key
```

In this case the item following the "**by**" operator in the **group** clause can referred to in the **orberby** clause as the **Key** property of the *ageGroup* **group**.

Loop through the *query* object using a **var** as the iterator field. This **var** will contain each grouping created by the group clause in the query. For each of these groups, print out the Key and the Count() to see the number of *Person* objects in each **group**.

```
foreach (var age in query)
{
    Console.WriteLine("Age: {0}  Count: {1}", age.Key, age.Count());
}
```

LastInitialDistribution()

This query will also use groups; however, instead of using the *Person*'s *AgeGroup* property, we'll group by the first character of the *Person*'s *LastName*:

```
group p by p.LastName[0] into distGroup
```

You can also sort the group by its key to get the initials in alphabetic order. Print out the group key and count for each item to get the distribution of last names.

GetCombinedNameLength()

This query is going to use three new techniques: the "let" statement, anonymous type projection, and selecting distinct records.

Start with:

```
var query = from p in people
```

Next, add a **let** statement to define a temporary variable, called "*name*" within the query:

```
let name = p.LastName + ", " + p.FirstName
```

Now, we can use *name* to filter and order the results:

```
where name.Length == 20
orderby name
```

Next, we are going to project the results into a new anonymous type. Anonymous types do not have a name that you can see without reflection, but it will contain read-only properties that you define:

```
select new { Name = name }
```

It is for this reason that we will use a **var** to loop through all of the data to print them out:

```
foreach (var name in query)
{
```

```
        Console.WriteLine(name);
    }
```

Notice when you print the results you may get duplicates. Since we are not retaining other properties like *ID* in the results, you may want to limit the results to unique or distinct values. You can do this by surrounding the entire query in parentheses and then performing the *Distinct* method upon the results. Here is the full query:

```
var query = (from p in people
             let name = p.LastName + ", " + p.FirstName
             where name.Length == 20
             orderby name
             select new { Name = name }).Distinct();
```

Follow-on Work

I highly recommend that you retain this example and make modifications/additions to the given queries to practice using the LINQ libraries. What you learned here will also apply to LINQ to XML and LINQ to SQL.