Description: Linq to XML

In this project you will use "Linq to Xml" to generate and query datasets.

Setup

Open the Lab5 solution provided. The files in the Lab5 project are:

- FirstNames.txt – 400 most common American first names
- LastNames.txt – 200 most common American last names
- Program.cs – Main console app class
- Student.cs – Student class

Student

The Student class is a typical data class that has an enum, state fields, accessor properties and member methods.

GradeEnum

Within the *Student* class, define a **public enum** called *GradeEnum* with the following values:

```
PreSchool = -1,
Kindergarten = 0,
First = 1,
Second,
Third,
Fourth,
Fifth,
Sixth,
Seventh,
Eighth,
Freshmen = 9,
Sophomore = 10,
Junior = 11,
Senior = 12,
College = 13
```

Notice that the **enum** values are normalized around *Kindergarten*. This is so the numeric values for most of the grades line up with their common names. This is a common technique if the values of the underlying **enum** are meaningful. Another example could be creating an **enum** of all of the chemical elements where their atomic numbers and **enum** values are the same.

Fields

Add the following private fields to the *Student* class:

```
private long m_ID = 0;
private string m_LastName = string.Empty;
private string m_FirstName = string.Empty;
private DateTime m_DOB = DateTime.MinValue;
private float m_GPA = 0f;
private GradeEnum m_Grade = GradeEnum.PreSchool;
```

Properties

Next, encapsulate all of the fields with properties. Remember, the easiest was to do this is to right click on a field and then select "Refactor" ➔ "Encapsulate Field". The only data validation you need to worry about is making sure ID is never negative.

Next, add a new **public** property of type **int** called *Age*. This property will only have a **get** accessor and it will return the result of calling *GetAge()* passing it the *DOB* property. We will define *GetAge()* in the "methods" section later. The code I have given you has a stub for *GetAge()* that simply returns 0.

Constructors

Add two constructors. The first constructor accepts no parameters and performs no operations. The second will accept all the parameters necessary to set all of the above-defined properties (except Age).

Methods

GetAge

GetAge has the following definition:

```
public static int GetAge(DateTime dob)
```

Notice that this method is static. That means you access the method externally using the "Student." prefix, not a specific instance of Student.

To calculate age given two dates, we need to determine the number of years in between those two dates, then account for the birthday of the current year.

1. Create a **DateTime** called *now* and assign to it the current date/time. **DateTime** has a **static** *Now* property that will give you this information.
2. Create an **int** called *years* and assign to it the difference in years between *now* and *dob*. **DateTime** objects have a *Year* property you can use to calculate the difference.
3. Next, check if *now*'s *Month* is less than *dob*'s *Month* **OR** if the *Months* are equal and now's *Day* is less than *dob*'s *Day*. If either condition is true, subtract 1 from *years*.
4. Return *years*.

GetGrade

This method uses the Student's age to estimate the school grade they are in:

```
public static GradeEnum GetGrade(DateTime dob)
```

1. Create a *GradeEnum* object called *grade*.
2. Get the age of the *Student* by calling *GetAge*, passing *dob*, and store the result in an **int** called *age*.
3. Now, depending on the value of *age*, assign the appropriate *GradeEnum* value to *grade*:
    a. If (age < 5) then the student is in pre-school, assign that value to *grade*
    b. A five year old is in kindergarten, assign it to *grade*
    c. 18 year olds and older are in college, assign it to *grade*
    d. Students between the ages of 5 and 17, then they are in one of the "numeric" grades
        i. You can use the fact that a first grader is 6 years old, a second grader is 7, etc. (The difference between the age and grade number is 5)
4. Return *grade*

ToString

This method returns a formatted string representation of the student:

```
public override string ToString()
```

The following format string can be used to format the Student data:

```
"{0,-12} {1:000-00-0000} {2,-25} {3,-25} {4:MM/dd/yyyy} {5:0.000}"
```

The parameters to the **string**.Format function should be the following:

Grade, ID, LastName, FirstName, DOB, GPA

Return the result of the Format command.

A *HeaderString* property was provided which will return a header string that lines up with this formatted string.

FromXElement

The final method of Student is a public static method that will convert data in an Xml element into a Student object:

```
public static Student FromXElement(XElement element)
```

The structure of the Xml within element looks something like the following:

```
<student id="1333215">
  <lastName>Herrera</lastName>
  <firstName>Eugene</firstName>
  <dob>2007-08-30T15:00:05.848</dob>
  <gpa>1.529</gpa>
  <grade>PreSchool</grade>
</student>
```

To convert this Xml to a Student we need to parse out the individual values that will map to the corresponding properties of Student.

ID

The ID is stored in the parent element's *id* attribute. To access an attribute of an XElement, we use the Attribute function of that XElement. We will also have to convert the attribute's value to an **long** as all data within Xml is a **string**.

```
long id = long.Parse(element.Attribute("id").Value);
```

LastName

The last name of the Student is stored in an element named lastName. Instead of using the Attribute method, we'll use the Elements method of the parent element to get at the lastName child element. Since the last name is a string, we won't need to convert it. However, since there is no guarantee that the child element exists, we'll use the FirstOrDefault method to retrieve the value:

```
string lastName = element.Elements("lastName").FirstOrDefault().Value;
```

FirstName

Use the same technique for firstName as you did with lastName.

DOB

Getting the date of birth for the student will also be very similar to the string methods except we need one final step to convert the string to a **DateTime**:

```
DateTime dob = DateTime.Parse(element.Elements("dob").FirstOrDefault().Value);
```

GPA

Use the same technique for gpa as you did with dob. The difference is that gpa is a **float**, not a **DateTime**.

Grade

Getting the grade is the trickiest field to retrieve. We use the same basic technique; however, parsing an **enum** requires a few extra steps. The **Enum** class contains a *Parse* method like most other types. But, for it to function, you need to pass to it the type of **enum** you wish to convert to. Then, you must cast the resulting value to the same **enum** type to store it.

```
GradeEnum grade = (GradeEnum)Enum.Parse(typeof(GradeEnum),
          element.Elements("grade").FirstOrDefault().Value);
```

Now, using the six temporary fields created above, create and return a new Student object.

Program

We will do most of the grunt work inside the Program class.

CreateXDocument

This method will accept a **List**<> of *Student* objects and convert it to an **XDocument**:

```
private static XDocument CreateXDocument(List<Student> students)
```

To do this we will use advanced **Linq** and functional programming techniques all at the same time. Plus, it will be done basically in one statement. Here is the code:

```
XDocument doc = new XDocument(
    new XElement("students",
        students.Select((item) => new XElement("student",
            new XAttribute("id", item.ID),
            new XElement("lastName", item.LastName),
            new XElement("firstName", item.FirstName),
            new XElement("dob", item.DOB),
            new XElement("gpa", item.GPA),
            new XElement("grade", item.Grade))
        )
    )
);
return doc;
```

Here are the details of what is going on:

First we create a new **XDocument** called doc. This constructor accepts a single parameter – the root element of the document. There are other overloads which allow you to assign the Xml Declaration and comments, but this is the simplest.

The root element of the document is called "students". That is the first parameter to the **XElement**'s constructor. The second parameter is the list of students. However, we don't simply pass *students* to the constructor. We must convert each student to Xml, and we use Linq to do it.

Consider the following statement:

```
students.Select((item) => new XElement("student", ...
```

What this says is for each Student (called *item*) in students, create a new **XElement** and select those new **XElements** as the child elements of the "students" parent element.

The rest of the items passed to the **XElement** constructor are the individual attributes and elements that make up a single "student" element.

GenerateStudents

This method is provided for you. It is called to generate a random set of *Students* and save the values as Xml to a file. Feel free to inspect the code and discover how it works. Note that it calls *CreateXDocument()* with the **List**<> of *Students* it generates.

OpenStudentFile

This method uses the **XDocument**.Load method to read a file saved by *GenerateStudents()*.

PrintHeader

This method is used by the query methods which you will be writing shortly.

Main

This method calls the methods necessary to create a Student Xml file and read it. It also calls the following query methods. Note, all of the queries are defined at the end of this document. Only look there if you cannot figure out the Linq commands.

FindAllGradeSchoolStudents

Using Linq, create a query to do the following:

- Get all descendant elements named "student" in the given **XDocument**
- Create a temporary field of to hold each Student's Grade to be used by the rest of the query
- Filter out students that are not in grade school (1^{st} to 6^{th} grade)
- Order the results by grade, last name and first name
- Select the results as Student objects using the Student.FromXElement method

FindValedictorianAndSalutatorian

Using Linq, create a query to do the following:

- Get all descendant elements named "student" in the given **XDocument**
- Create a temporary field of to hold each Student's Grade to be used by the rest of the query
- Create a temporary field of to hold each Student's GPA to be used by the rest of the query
- Filter out students that are not seniors
- Order the results by GPA in descending order
- Select the top 2 elements as Student objects using the Student.FromXElement method
    - To get top two elements, use the Take method on the result of the query

GpaStatisticsPerGradeLevel

Using Linq, create a query to do the following:

- Get all descendant elements named "student" in the given **XDocument**
- Create a temporary field of to hold each Student's Grade to be used by the rest of the query
- Order the results by GPA in ascending order
- Group the results by grade level
- Order the groups by the group's Key
- Select the groups as the result

Loop through the query results which are group items

- Determine the lowest GPA per grade level by using the Min() function on the group
- Determine the highest GPA per grade level by using the Max() function on the group
- Determine the average GPA per grade level by using the Average() function on the group
- Output the number of students, lowest GPA, highest GPA and average GPA per grade level

The Following Pages Contain the Query Code

```
private static void FindAllGradeSchoolStudents(XDocument doc)
{
    var query = from student in doc.Descendants("student")
                let grade = (Student.GradeEnum)Enum.Parse(typeof(Student.GradeEnum),
                    student.Element("grade").Value)
                where grade >= Student.GradeEnum.Kindergarten && grade <= Student.GradeEnum.Sixth
                orderby grade, student.Element("lastName").Value, student.Element("firstName").Value
                select Student.FromXElement(student);

    PrintHeader("FindAllGradeSchoolStudents");

    foreach (Student s in query)
    {
        Console.WriteLine(s);
    }
}

private static void FindValedictorianAndSalutatorian(XDocument doc)
{
    var query = (from student in doc.Descendants("student")
                let grade = (Student.GradeEnum)Enum.Parse(typeof(Student.GradeEnum),
                    student.Element("grade").Value)
                let gpa = float.Parse(student.Element("gpa").Value)
                where grade == Student.GradeEnum.Senior
                orderby gpa descending
                select Student.FromXElement(student)).Take(2);

    PrintHeader("FindAllGradeSchoolStudents");

    if (query.Count() == 2)
    {
        Console.WriteLine("Valedictorian");
        Console.WriteLine(query.ElementAt(0));
        Console.WriteLine("Salutatorian");
        Console.WriteLine(query.ElementAt(1));
    }
}
```

```
private static void GpaStatisticsPerGradeLevel(XDocument doc)
{
    var query = from student in doc.Descendants("student")
                let grade = (Student.GradeEnum)Enum.Parse(typeof(Student.GradeEnum),
                    student.Element("grade").Value)
                orderby float.Parse(student.Element("gpa").Value)
                group student by grade into g
                orderby g.Key
                select g;

    PrintHeader("GpaStatisticsPerGradeLevel", false);

    foreach (var group in query)
    {
        Console.WriteLine(group.Key);
        Console.WriteLine("------------");

        float min = group.Min(student => float.Parse(student.Element("gpa").Value));
        float max = group.Max(student => float.Parse(student.Element("gpa").Value));
        float avg = group.Average(student => float.Parse(student.Element("gpa").Value));
        Console.WriteLine("  Students: {0,4}   GPA (min/max/avg): {1:0.000} / {2:0.000} / {3:0.000}",
            group.Count(), min, max, avg);
        Console.WriteLine();
    }
}
```