|  |
| --- |
|  |
| LINQ Introduction |
| Getting Started |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

Introducing LINQ

Objectives

After completing this lab, you should understand how to do the following:

* Create some basic LINQ queries to sort and filter objects.
* Take advantage of LINQ’s deferred execution.
* Use LINQ to XML and the XDocument class to create an XML file.

Overview

In our first application with LINQ, we will build a WPF application that examines the fuel efficiency for all cars produced in 2008. The application will load data from a file of comma-separated values and display the data in a grid.

We’ll be using data collated by the U.S. government. The U.S. government measures fuel efficiency in miles per gallon (MPG). There are two measurements for each car model – the city measurement and highway measurement. These rating are determined by driving a vehicle under “city” condition (stop and go traffic), and highway conditions (the car maintains a relatively high speed for an extended period). For metric aficionados, 1 gallon = 3.785 liters, and 1 mile = 1.61 kilometers.

Part 1 –CSV the LINQ way

Double-click the Cars.sln file in your LINQ\_Introduction\Before\Cars directory to load the solution into Visual Studio. This is a starter application for you to work in. A completed version of the application resides inside the LINQ\_Introduction\After directory. You can peek in the After directory if you want to see the final version of the project.

In the Solution Explorer window (CTRL+ALT+L), double-click the CarDisplay.xaml file. Review the user interface we will be working with. Although it’s not visible in the designer window, a ListView in the center of the application will show our data in a grid. You can examine the XAML markup in the bottom half of the split WPF designer.

Press F7 to see the code-behind file for the XAML, which is nearly empty. There is a constructor and some event handlers in place, so we’ll need to provide the rest of the logic during the course of this lab.

Next, open up cars.csv. This is the data file we’ll need to load into memory, and it contains values for each car’s manufacturer, name, engine displacement, and more. The Cars project file is configured to copy the csv file into the bin directory during a build (you can see these settings if you right-click and look at the properties of cars.csv).

Open up Cars.cs. This is the class we’ll use to represent the car data.

Now that we’ve toured the existing files, we can get to work. Our first step is to import the csv data.

# Building a Car Repository

1. Right-click the Cars project and select Add -> Class. Give the new file a name of CarRepository.cs.

The repository pattern tries to hide the source of data and the queries needed to retrieve data. We’ll push as much LINQ inside our repository as possible, leaving the UI free to make method calls and consume the results.

Give the repository class a public constructor that take a string parameter named fileName, and a private field named \_cars that is of type List<Car>.

class CarRepository

{

public CarRepository(string fileName)

{

}

private List<Car> \_cars;

}

1. Add a using statement at the top of the file to bring in the System.IO namespace.

In the old days (before LINQ), we’d typically use a foreach or while loop to process each line of text, like the following (don’t use this code):

string[] lines = File.ReadAllLines(fileName);

foreach(string line in lines)

{

string[] columns = line.Split(',');

// ...

}

We’ll use a LINQ query instead of the foreach loop. The first line of the query will introduce a variable (named lines) to range over the sequence of lines in the file.

1. Add the following code to the constructor, which is just the beginning of a LINQ query, into the constructor:

IEnumerable<Car> cars =

from line in File.ReadAllLines(fileName)

1. We’ll need to split each *line* into columns. The *let* clause in LINQ is perfect for introducing a variable that we can use later in the query. Add the third line below to the query:

IEnumerable<Car> cars =

from line in File.ReadAllLines(fileName)

**let columns = line.Split(',')**

1. We can now add our *select* operator and construct a new Car object inside the query. To construct the Car object we’ll use C#’s object initializer syntax to assign the new car’s properties.

IEnumerable<Car> cars =

from line in File.ReadAllLines(fileName)

let columns = line.Split(',')

**select new Car {**

**Manufacturer = columns[0],**

**Name = columns[1],**

**Displacement = Convert.ToDouble(columns[2]),**

**Cylinders = Convert.ToInt32(columns[3]),**

**Transmission = columns[4],**

**CityMPG = Convert.ToDouble(columns[6]),**

**HighwayMPG = Convert.ToDouble(columns[7])**

**};**

1. We have a subtle bug in our query. Can you spot it? Open the cars.csv file and think about how the query will process the file. If you can’t spot the bug, don’t worry, we’ll fix the problem in the next step.

The problem is our query will try to convert the first line of the csv file into a Car object. Although the query will work, we should exclude this header information by filtering out the first row.

1. Since it seems unlikely that we’ll ever have a car manufacturer with the name of “Manufacturer”, let’s add a *where* operator to the query to filter out the header. Add the *where* operator immediately after the *let*.

from line in File.ReadAllLines(fileName)

let columns = line.Split(',')

**where columns[0] != "Manufacturer"**

...

1. Our data load is finished with no foreach loops in sight! Now we need to convert our query into a List<Car> collection. Use the ToList() operator on the query to create the List<Car>.

\_cars = cars.ToList();

1. The final version of the constructor should look like the following:

public CarRepository(string fileName)

{

IEnumerable<Car> cars =

from lines in File.ReadAllLines(fileName)

let columns = lines.Split(',')

where columns[0] != "Manufacturer"

select new Car

{

Manufacturer = columns[0],

Name = columns[1],

Displacement = Convert.ToDouble(columns[2]),

Cylinders = Convert.ToInt32(columns[3]),

Transmission = columns[4],

CityMPG = Convert.ToDouble(columns[6]),

HighwayMPG = Convert.ToDouble(columns[7])

};

\_cars = cars.ToList();

}

1. Finally, we need to give our repository a method that will allow clients of the repository to retrieve cars. Add a public FindAll method to the repository that returns an IEnumerable<Car>.

public IEnumerable<Car> FindAll()

{

return \_cars;

}

1. Right-click CarDisplay.xaml and select View Code to open CarDisplay.xaml.cs.
2. Add a private field to CarDisplay of type CarRepository and with the name \_repository.
3. Inside the CarDisplay constructor, and after the call to InitializeComponent, assign a new instance of CarRepository to the \_repository field. Pass “cars.csv” as the filename parameter for the repository’s constructor.
4. The ListView in our WPF application is named \_carsView. Set the ItemsSource property of \_carsView to the return value of the repository’s FindAll method. The final code should look like the following.

public CarDisplay()

{

InitializeComponent();

\_repository = new CarRepository("cars.csv");

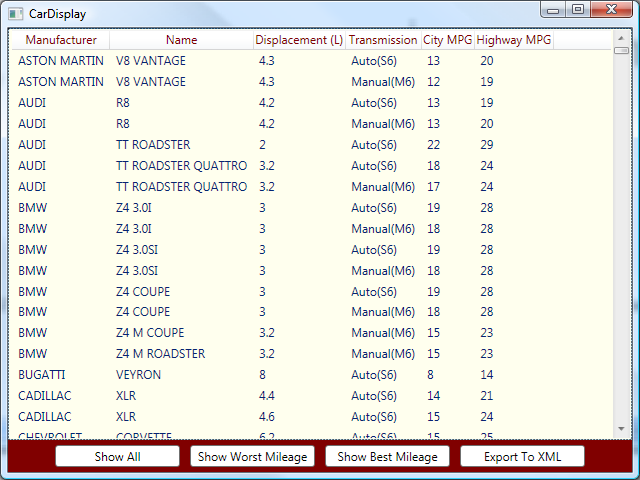
\_carsView.ItemsSource = \_repository.FindAll();

}

private CarRepository \_repository;

...

1. At this point you should be able to press F5 to run the application. Make sure the application is behaving properly before we move to the next section.



# Part II - Ordering

1. We are going to implement the event handlers for the three Show buttons in our application. We’ll first need some additional logic in our repository, so open the CarRepository.cs file again.
2. Add a new public method to the repository named FindWorstMpg. The return type should again be IEnumerable<Car>.
3. Inside FindWorstMpg, build and return a query that will select all cars from the \_cars list. Use an *orderby* operator to sort the cars by the CityMPG property in *ascending* order. The code should look similar to the following.

public IEnumerable<Car> FindWorstMpg()

{

return from c in \_cars

orderby c.CityMPG ascending

select c;

}

1. Open CarDisplay.xaml.cs and find the \_wostMileageButton\_Click event handler. Inside, set the ItemsSource property of the \_carsView to the return value of the FindWorstMPG method.

private void \_wostMileageButton\_Click(object sender,

RoutedEventArgs e)

{

\_carsView.ItemsSource = \_repository.FindWorstMpg();

}

1. Run the project (F5), and look at the results.

The [Bugatti Veyron](http://www.bugatti-configurator.com/bugatti_en.html) should appear at the top of the display when you click the “Show Worst Mileage” button. That’s the price you pay for an 8-liter engine. Of course, if you own a Veyron, chances are you don’t really care about fuel efficiency. However, notice the Veyron’s highway mileage is actually better than the Lamborghinis that follow it in the view.

1. Go back to the CarRepository, and tweak the FindWorstMPG query to use the each car’s CityMPG **and** HighwayMPG properties.

public IEnumerable<Car> FindWorstMpg()

{

return from c in \_cars

orderby (c.CityMPG + c.HighwayMPG) ascending

select c;

}

1. Run the project again (F5) and examine the new results.

Many cars are tied at various levels of efficiency. If you scroll through the results, the list looks unordered because the manufacturer names appear randomly when cars are tied in fuel efficiency.

1. Go back to the repository and tweak the query one more time. Use the Manufacturer property of each car in a *secondary* sort.

public IEnumerable<Car> FindWorstMpg()

{

return from c in \_cars

orderby (c.CityMPG + c.HighwayMPG) ascending,

c.Manufacturer ascending

select c;

}

Let’s also limit the number of cars displayed during this operation in the UI layer. To do this we can use the LINQ Take operator. We’ll be taking a closer look at these operators in a later module.

1. Add a *using* statement in CarDisplay.xaml.cs for the System.Linq namespace.
2. Change the \_worstMileageButton click event handler to only *Take* the first 50 results from the repository.

\_carsView.ItemsSource = \_repository.FindWorstMpg().Take(50);

1. Add logic to the \_showAllButton\_Click event handler to reload all the cars from the repository.

\_carsView.ItemsSource = \_repository.FindAll();

1. Run the project and ensure everything is in working order.

You should be able to look at the 50 worst cars for fuel efficiency, then click the Show All button to see a list of all cars. If you want, you can change the query in the FindAll method of the repository to provide a default ordering to the “all cars” view.

# Part III – Deferred Execution

1. Now we need to implement the logic to show the **most** fuel efficient vehicles. Open up CarRepository.cs and add a public method named FindBestMpg that return an IEnumerable<Car>.
2. Write a query that will order the cars by their fuel efficiency in *descending* order (you could copy the code from your previous method and change just one keyword).

public IEnumerable<Car> FindBestMpg()

{

return from c in \_cars

orderby (c.CityMPG + c.HighwayMPG) descending,

c.Manufacturer ascending

select c;

}

1. Let’s also give the repository an *Add* method that will take an incoming Car parameter. Simply add the incoming car to the repository’s private list.

public void Add(Car newCar)

{

\_cars.Add(newCar);

}

1. Open up CarDisplay.xaml.cs and find the \_bestMileageButton\_Click event handler.
2. To see deferred execution in action, create a new Car object inside this method and *Add* it to the car repository before any other code executes. You can assign any attributes to the car that you want, just make sure the car is the most fuel-efficient car ever made (so that it appears at the top of the most efficient list).
3. After adding the car to the repository, use the FindBestMPG method we implemented earlier to populate the \_carsView list. The code should look similar to the following.

private void \_bestMileageButton\_Click(object sender,

RoutedEventArgs e)

{

Car fakeCar = new Car()

{

Manufacturer = "Pluralsight",

Name = "Pluralsight Party Van",

Displacement = 4,

Cylinders = 8,

Transmission = "R",

CityMPG = 100,

HighwayMPG = 150

};

\_repository.Add(fakeCar);

\_carsView.ItemsSource = \_repository.FindBestMpg();

}

1. Run the project under the debugger (F5) and make sure the new event handler is working correctly. Your fake car should appear in the list of the most fuel-efficient vehicles.

You probably were not surprised to see your fake car appear in the list, after all, we added the car before we queried the repository, but let’s change the implementation of this method to add the car after we’ve called into the repository.

1. Add a using statement for the System.Collections.Generic namespace to the top of the file.
2. Move the Add call to after the FindBestMpg call but *before* we apply a new *Take* operator to grab only the top 50 cars. In between, we can store the return value of the repository in an IEnumerable<Car>. Do you think your fake car will still appear? Your implementation should look something like the following:

private void \_bestMileageButton\_Click(object sender,

RoutedEventArgs e)

{

Car fakeCar = new Car()

{

// ... initializer here

};

**IEnumerable<Car> cars = \_repository.FindBestMpg();**

**\_repository.Add(fakeCar);**

**cars = cars.Take(50);**

**\_carsView.ItemsSource = cars;**

}

1. Run your project and find out if your answer was correct.

The fake car should still appear in the list. This is the magic of LINQ’s deferred execution. We added the car after we executed the FindBestMpg() method, but the query did not execute until the ListView enumerated the query to display the cars. When the ListView executed the query, the fake car was present in the repository and returned by the query.

1. Now let’s change one line of code – add a call to ToList *after* the call to FindBestMpg. Do you think the car will still appear after the following change?

IEnumerable<Car> cars = \_repository.FindBestMpg()**.ToList();**

1. Run the project one more time.

If you’ve made the change correctly, your fake car should **not** appear in the list of the most efficient vehicles. We added the fake car **after** the query executed. The ToList operator had to execute the query in order to create a new List<Car>.

1. If you are not comfortable yet with deferred execution, continue to change around the call to \_repository.Add and the ToList operator to understand what is happening. You can also place a break point inside the method and analyze results with the debugger. Ask your instructor if you need some help.

# Part IV – LINQ to XML

1. Our final step is to implement the logic behind the Export button. We want to export all of the cars in the repository into an XML file. Open CarRepository.cs and add a new public method. The method should be named Export, should return void, and should take one string parameter named fileName.

public void Export(string fileName)

{

}

1. At the top of the CarRepository.cs file, add a using statement for System.Xml.Linq.
2. Create a new instance of the XDocument class and assign the instance to a local variable.
3. Invoke save on the XDocument instance, passing in the fileName parameter. The code should look like the following, which gives us a basic outline to use:

XDocument document = new XDocument();

document.Save(fileName);

1. We are going to use functional construction to build our XML document. We’ll build everything we need *inside the constructor* of the XDocument object. First, well need a root element. Create a new XElement inside the XDocument constructor, passing “cars” into the XElement constructor:

XDocument document = new XDocument(

**new XElement("cars")**

);

1. Next, we’ll need to create an XElement for every car in our repository. We can do this using a LINQ query *inside* the XElement constructor we just wrote. Write a query that will construct a new XElement(“car”) for every object in the \_cars collection. The first parameter to XElement represents the name of the XML element.

Be sure to mind your parentheses and commas to ensure the car element is created inside the root element and not as part of the XDocument. In other words, the query should be the second parameter to the <cars> constructor.

XDocument document =

new XDocument(

new XElement("cars",

**from c in \_cars**

**select new XElement("car")**

)

);

1. Let’s test what we have by adding some code to the button click event. Open the CarDisplay.xaml.cs file and locate the \_exportButton\_Click event handler. Add the following code to invoke the repository’s export method, passing cars.xml as the file name.

private void \_exportButton\_Click(object sender,

RoutedEventArgs e)

{

**\_repository.Export("cars.xml");**

}

1. Run the project and go find the cars.xml file. It should be in the bin\Debug folder of the project and look like the following XML. We should see an empty XML element for each car. Once you’ve verified everything is working so far, continue to the next step.

<?xml version="1.0" encoding="utf-8"?>

<cars>

<car />

<car />

<car />

...

1. Let’s add in some details about each car. After the first parameter to the <car> XElement constructor, add an XElement for every property of a Car. The code should look like the following:

XDocument document =

new XDocument(

new XElement("cars",

from c in \_cars

select new XElement("car",

**new XElement("Manufacturer",**

**c.Manufacturer),**

**new XElement("Name", c.Name),**

**...**

**new XElement("CityMPG", c.CityMPG),**

**new XElement("HighwayMPG", c.HighwayMPG)**

)

)

);

document.Save(fileName);

1. Run your project and verify the contents of cars.xml again. Each <car> element should now have some descendants:

<?xml version="1.0" encoding="utf-8"?>

<cars>

<car>

<Manufacturer>ASTON MARTIN</Manufacturer>

<Name>V8 VANTAGE</Name>

<CityMPG>13</CityMPG>

<HighwayMPG>20</HighwayMPG>

</car>

<car>

…

1. Let’s make one slight change to our XML. Instead of using “element centric” XML, let’s add XML attributes to each <car> element. Change the appropriate statements that use XElement to use XAttribute instead.

XDocument document =

new XDocument(

new XElement("cars",

from c in \_cars

select new XElement("car",

**new XAttribute("Manufacturer",**

**c.Manufacturer),**

**new XAttribute("Name", c.Name),**

**// ... additional properties to save**

**new XAttribute("CityMPG", c.CityMPG),**

**new XAttribute("HighwayMPG", c.HighwayMPG)**

)

)

);

1. Run your project one more time, and look at the output. The XML should now look like the following.

<?xml version="1.0" encoding="utf-8"?>

<cars>

<car Manufacturer="ASTON MARTIN" Name="V8 VANTAGE"

CityMPG="13" HighwayMPG="20" />

...

# Conclusion

Congratulations! You’ve finished an introductory lab to LINQ. You should now feel comfortable with the basic query syntax in LINQ, and be able to read and write basic queries. You’ve seen how deferred execution works, and how to compose queries by adding additional operators in higher layers. Finally, you’ve written some LINQ to XML and produced an XML document using less code than you would have needed using the older XmlDocument API.