|  |
| --- |
|  |
| LINQ & C# |
| C# Features for Language Integrated Query |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

# 

LINQ and C#

# Objectives

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

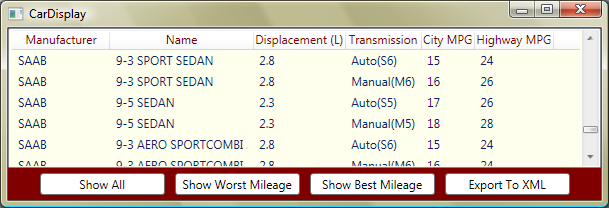
* Create and consume an extension method.
* Use lamba expressions and recognize the difference between lambdas and expression trees.
* Take advantage of type inference and anonymous types.

# Overview

In this lab we will continue building a WPF application that examines the fuel efficiency for all cars produced in 2008. You can continue working on your final solution from the Introduction to LINQ lab, or you can start with the project in the *LINQ\_and\_C#\before* directory. If you finished the previous lab (Introduction to LINQ), your project and the “before” project for this lab should be functionally identical.

# Part 1 – Extending System.String

Press F5 to build and run the project to ensure all the bits are in working order.



Notice how the manufacturer and car names are capitalized. We’ll try to fix this using some string processing. In the days of .NET 2.0 we might put our string processing into a class with a name like StringUtils, however, we are going to use an extension method to make it look like we’ve built a new method into the String type.

1. Right-click the Cars project and select “Add” -> “Class”.
2. In the add class dialog, enter a name of StringExtensions.cs
3. In the new StringExtensions.cs file, make the StringExtensions class a public, static class.
4. Change the namespace for the file to Utility.
5. Define an extension method named PrettyCase that will work with any string object. Remember the first parameter will need to be a string type and use the *this* keyword. We won’t need additional parameters, but the method must return a *string*. The code should look like the following.

using System;

namespace Utility

{

public static class StringExtensions

{

public static string PrettyCase(

this string value)

{

}

}

}

1. Implement an algorithm that will “beautify” a string of text by lowercasing all characters in the string’s words *after* the first character. Exclude words with a length of less than three characters, as these are commonly capitalized (TT, GT, etc.). The following is a brute force algorithm, perhaps you could make it better!

public static string PrettyCase(this string value)

{

string[] words = value.Split();

for (int i = 0; i < words.Length; i++)

{

if (words[i].Length > 3)

{

char[] characters = words[i].ToCharArray();

for (int j = 1; j < characters.Length; j++)

{

characters[j] = char.ToLower(characters[j]);

}

words[i] = new string(characters);

}

}

return String.Join(" ", words);

}

1. Open up the file CarRepository.cs
2. Add a using statement for the Utility namespace – this will bring the extension method into scope.
3. Find the constructor for the CarRepository class in CarRepository.cs. Apply the extension method inside the select clause of the LINQ query of the constructor. We only need to use the extension method on the strings assigned to the *Manufacturer* and *Name* properties.

select new Car

{

**Manufacturer = columns[0].PrettyCase(),**

**Name = columns[1].PrettyCase(),**

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

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

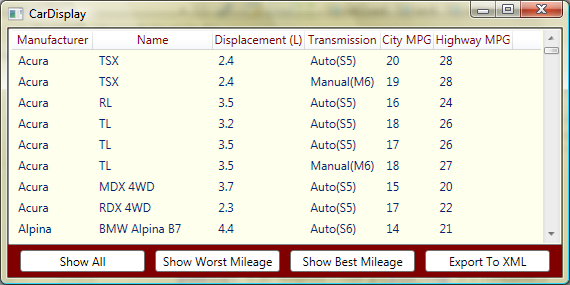
Transmission = columns[4],

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

HighwayMPG = Convert.ToDouble(columns[7])

};

1. Press F5 to run the project and verify your changes.



# Part II – Filtering with Lambdas

In Part II, we are going to give our car repository a Find method where the caller can specify a filter to select a subset of the total car population.

Open up CarRepository.cs.

One approach to filtering would be to add a Find method for each car property (FindByManufacturer, FindByName), however, this could lead to an explosion in the number of methods (we’d probably need FindByManufacturerAndName, too). Let’s use a functional programming style and have the client pass a method that the repository will use to filter the list of cars.

1. Add a Find method that returns IEnumerable<Car> and takes a Func<Car, bool> as a parameter. The type parameters to the function specify that the incoming Func will take one argument of type Car, and return a bool.

public IEnumerable<Car> Find(Func<Car, bool>specification)

{

}

1. Inside the Find method, we need to apply the specification to our repository of cars. The easiest approach is to use the specification in a LINQ query:

**return from c in \_cars**

**where specification(c)**

**orderby c.Manufacturer ascending**

**select c;**

1. To try the new method, open up CarDisplay.xaml. Add the following XAML directly below the \_exportButton markup:

<Button x:Name="\_filter" Click="\_filter\_Click"

Margin="5" Width="125"

Background="White">

Filter

</Button>

1. In the design window, you might need to expand the width of the CarDisplay main window for the button to show.
2. Press F7 to switch to the code view of CarDisplay.xaml. Add the following event handler.

private void \_filter\_Click(object sender,

RoutedEventArgs e)

{

}

1. Now, let’s try filtering the display with a lambda expression. Try the following line of code, which should filter out all cars that get less than 40 miles to the gallon in city driving conditions:

\_carsView.ItemsSource =

\_repository.Find(c => c.CityMPG > 40);

1. Press F5 to run the project. The Toyota Prius should be the only car to display after you click the filter button.

When writing the lambda, the C# compiler can infer the type of the variable c as a Car, because it sees that we are creating a Func<Car, bool>. The lambda is a shorthand notation for the older anonymous method syntax shown below (you don’t need to use this code; just compare this syntax to the lambda expression and notice the differences - no type names, no return statement, no curly brackets).

\_carsView.ItemsSource = \_repository.Find(

delegate(Car c) { return c.CityMPG > 40; }

);

1. Try replacing the lambda expression inside the filter click event with lambda expressions that do the following. If you have problems, ask your instructor for help. Run the application to verify each lambda.
   1. Display only cars whose Manufacturer starts with a “B”.
   2. Display only cars whose Manufacturer **and** Name properties contain a “z”.
   3. Display only cars whose CityMPG is greater than their HighwayMPG (as is often the case with hybrids).
2. Return to the CarRepository.cs file. Let’s rewrite the query inside our new Find method using method invocations and lambda expressions. Remember that the *where* and *select* operators are just extension methods for IEnumerable<T>. A strict rewrite of the query would look like the following.

return \_cars.Where(specification)

.Select(c => c);

Technically, the Select method call is not required since we are returning a sequence of Cars. Select is required when we are projecting, that is transforming the original sequence of objects into something different. You might prefer the method call / lambda syntax above when you are thinking in terms of a pipeline instead of a query. It’s easier to see that the sequence of cars passes through a Where method, then a Select method.

# Part III – Exploring Expressions

1. Open CarDisplay.xaml.cs and ensure your \_filter\_Click event handler is using the following lamda expression for filtering:

\_carsView.ItemsSource =

\_repository.Find(**c => c.CityMPG > c.HighwayMPG**);

1. Place a breakpoint (F9) on the opening curly bracket of the Find method we implemented inside CarRepository during this lab.
2. Press F5 to run the application with the debugger, and click the Filter button. Execution should now be paused inside the Find method.
3. Hover the mouse over the specification parameter. You should see a debugger display similar to the following.



1. Notice that the debugger can only tell us that the specification parameter references an anonymous method that returns bool and takes a Car parameter.
2. Stop debugging (Shift+F5).
3. Let’s make a change to our Find method. Change the method to accept an Expression<Func<Car, bool>> instead of just a Func<Car, bool>. You’ll need to add a using statement for System.Linq.Expressions.

public IEnumerable<Car>

Find(Expression<Func<Car, bool>> specification)

{

...

}

If you try to compile the project at this point, you’ll get an error. When the C# compiler sees an Expression<T>, it doesn’t create an anonymous method that we can invoke. Instead, it builds an expression tree. To use the expression tree with the Where operator, we first need to compile the expression at runtime.

1. Make the highlighted change to the Find method below.

public IEnumerable<Car>

Find(Expression<Func<Car, bool>> specification)

{

return \_cars.Where(**specification.Compile()**)

.Select(c => c);

}

1. Press F5 to run the project and click Filter.
2. You should hit your breakpoint inside the Filter method again. Hover the mouse over the specification parameter to see the following.



1. Notice we now see the actual lambda expression. Drill into the debugger view of the expression by expanding + signs. You should discover that the *Body* property of the expression is of type *BinaryExpression* (*GreaterThan*), and that the *BinaryExpression* has *Left* and *Right* properties that respectively access the *CityMPG* and *HighwayMPG* properties of a Car object.

Think about how remote LINQ technologies like LINQ to SQL work - they need to inspect these expressions at runtime to see what operators are applied to different properties and formulate a SQL command. In other words, the lambda expression is not only available to the debugger, but to code that is executing at runtime.

1. Let’s try to reverse the expression (instead of an expression saying c.CityMPG > c.HighwayMPG, we’ll create an expression for c.CityMPG < c.HighwayMPG). Unfortunately, expression trees are immutable, but we can create a new expression tree using pieces of the existing expression tree.
2. Add the following line of code as the first line of code in the Find method. This code saves a reference to the binary expression inside of the specification.

BinaryExpression binary =

specification.Body as BinaryExpression;

1. For the next line of code, we will create a LessThan expression using the same Left and Right properties as the original binary expression.

BinaryExpression newBinary =

Expression.LessThan(binary.Left, binary.Right);

1. Now, we’ll combine the newBinary expression with the parameters from the original specification.

Expression<Func<Car,bool>> newExpression =

Expression.Lambda<Func<Car,bool>>(newBinary,

specification.Parameters.ToArray());

1. What we’ve built is a new expression tree that applies a “less than” operator to the same properties and car parameter as the original specification. All we need to do now is pass newExpression into the Where operator (after invoking Compile, of course).

return \_cars.Where(newExpression.Compile())

.Select(c => c);

1. At this point, press F5 to ensure the new expression is working. We should only see cars whose highway gas mileage is **better** than their city gas mileage (non-hybrids, generally).

To be clear, the code we’ve written will **only** work if the incoming expression is a binary expression. However, this simple demonstration will hopefully open your eyes to the power of expression trees. The ability to inspect expressions at runtime is the enabling factor for LINQ providers. Providers can parse and convert the expression tree into SQL statements, web service requests, and more.

1. To make the Find method work for all expressions, we should comment out the lines we added and restore the original specification.

public IEnumerable<Car>

Find(Expression<Func<Car, bool>> specification)

{

//BinaryExpression binary = specification.Body

// as BinaryExpression;

//BinaryExpression newBinary =

// Expression.LessThan(binary.Left, binary.Right)

//Expression<Func<Car,bool>> newExpression =

// Expression.Lambda<Func<Car,bool>>(newBinary,

// specification.Parameters.ToArray());

return \_cars.Where(specification.Compile())

.Select(c => c);

}

# Part IV – Anonymous Types

In this section, we’ll produce a “Manufacturer’s Summary Report” that shows the count of cars and average gas mileage for each car manufacturer.

Right-click the project file and select “Add Existing Item…”.

1. Browse to the “before” folder for this lab (LINQ\_and\_C#) and select both the Summary.xaml and Summary.xaml.cs files. Click Add.

The SummaryWindow we’ve just added contains a ListView that expects to bind against objects with the following properties: Manufacturer, TotalCars, AvgCityMPG, and AvgHighwayMPG. If you explore the contents of the Summary.xaml file, you’ll find the GridViewColumns inside setup to display these attributes. You might also notice that we never have to specify the exact type of the object we will bind against, however.

1. Open up CarDisplay.xaml and add a MouseDoubleClick event handler to the \_carsView ListView.

You should be able to let Visual Studio auto-generate the code for the event handler when you type MouseDoubleClick into the xaml. The XAML will look like the following:

<ListView Name="\_carsView"

Background="#ffffee"

**MouseDoubleClick="\_carsView\_MouseDoubleClick">**

<ListView.View>

…

1. Open CarDisplay.xaml.cs. Locate the event handler for the double click event (if you let Visual Studio generate the code), or type the following into the CarDisplay class:

private void \_carsView\_MouseDoubleClick(

object sender, MouseButtonEventArgs e)

{

}

1. We are going to write a LINQ query that will summarize our car data. This query will go directly into the event handler, but rely on our CarRepository to fetch the raw data. We need to group our cars by manufacturer.

var result =

from c in \_repository.FindAll()

group c by c.Manufacturer into gc

orderby gc.Count() descending

select new

{

Manufacturer = gc.Key,

TotalCars = gc.Count(),

AvgCityMPG = gc.Average(c => c.CityMPG)

.ToString("N2"),

AvgHighwayMPG = gc.Average(c => c.HighwayMPG)

.ToString("N2")

};

1. After grouping the cars, the above query creates a new anonymous type with just the properties we need for binding in our summary view. Since we do not have a type name for this anonymous type, we are using the var keyword to declare the result variable.
2. Finally, we need to create an instance of the SummaryWindow, bind this result to the window, and display the window.

SummaryWindow summaryWindow = new SummaryWindow();

summaryWindow.DataContext = result;

summaryWindow.ShowDialog();

1. Press F5 to run the project and ensure everything is working. Congratulations! You just used an anonymous type in a useful fashion!

# Conclusion

In this lab we wrote some advanced LINQ queries that summarized data, created new lambda expressions, expressions trees, and extended System.String with a useful extension method. These pieces (lambdas expressions, expression trees, and extension methods) are the cornerstones of LINQ, and this knowledge will give you a solid foundation to build upon.