The Internal Representation of LINQ Query Statements

At this point, you have been introduced to the process of building query expressions using various C#

query operators (such as from, in, where, orderby, and select). Also, you discovered that some

functionality of the LINQ to Objects API can only be accessed when calling extension methods of the

Enumerable class. The truth of the matter, however, is that when compiled, the C# compiler actually

translates all C# LINQ operators into calls on methods of the Enumerable class. In fact, if you were really a

glutton for punishment, you could build all of your LINQ statements using nothing but the underlying

object model.

A great many of the methods of Enumerable have been prototyped to take delegates as arguments. In

particular, many methods require a generic delegate named Func<>, which was introduced to you during

our examination of generic delegates . Consider the Where() method of Enumerable, which is

called on your behalf when you use the C# where LINQ query operator:

// Overloaded versions of the Enumerable.Where<T>() method.

// Note the second parameter is of type System.Func<>.

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,

System.Func<TSource,int,bool> predicate)

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,

System.Func<TSource,bool> predicate)

The Func<> delegate (as the name implies) represents a pattern for a given function with a set of up

to 16 arguments and a return value. If you were to examine this type using the Visual Studio object

browser, you would notice various forms of the Func<> delegate. For example:

// The various formats of the Func<> delegate.

public delegate TResult Func<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)

public delegate TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)

public delegate TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2)

public delegate TResult Func<T1,TResult>(T1 arg1)

public delegate TResult Func<TResult>()

Given that many members of System.Linq.Enumerable demand a delegate as input, when invoking

them, you can either manually create a new delegate type and author the necessary target methods,

make use of a C# anonymous method, or define a proper lambda expression. Regardless of which

approach you take, the end result is identical.