LINQ and Productivity

By **Bill Wagner**

January 2012

Since its introduction in C# 3.0, LINQ has changed the paradigms many C# developers use to create software. However, it seems that there's still a large segment of the C# developer community that hasn't yet fully adopted it. When I talk to developers that haven't yet made LINQ part of their everyday toolbox, it's because they haven't had time to invest in it fully, or their employer hasn't yet adopted LINQ as one of the techniques they intend to use in all their work. That's a shame, because LINQ, both the libraries and the language enhancements, enable me to write algorithms that are easier to read, maintain, and extend. A reasonable investment in learning the toolset pays off heavily in programmer productivity, both for you, and the rest of your team that will read your code and extend it in the future.

To illustrate what I mean, I wrote two versions of a program that can help you learn about the methods that most developers mean when they refer to LINQ. This sample uses reflection to print out the public methods in the System.Linq.Enumerable class, the one that represents much of the LINQ APIs. This program loads the system.core.dll assembly, and then uses the reflection APIs to find all the methods, their parameters, their generic type parameters, return types, and attributes. It formats that information to show you what the API signature looks like, and writes that information to the console. Let's examine both versions, and compare the differences in readability and expressiveness.

The World before LINQ

Figure 1 shows the two methods that make up the algorithm for finding all the LINQ methods without leveraging the LINQ APIs.

Figure 1: Find all the LINQ apis without using LINQ:

```
private static void NoLinqAnalysis(Assembly assembly)

{

   Type[] types = assembly.GetExportedTypes();

   Type enumerable = null;

   foreach (Type t in types)

      if (t.Name == "Enumerable")

         enumerable = t;
```

```
if (enumerable == null)
{
    Console.WriteLine("No data found.");
    return;
}
MethodInfo[] methods = enumerable.GetMethods(BindingFlags.Public |
                BindingFlags.Static);
foreach (var m in methods)
{
    string generics = "";
    if (m.IsGenericMethodDefinition)
    {
        Type[] typeParms = m.GetGenericArguments();
        StringBuilder parmString = new StringBuilder();
        parmString.Append("<");
        for (int i = 0; i < typeParms.Length; i++)
        {
            parmString.Append(typeParms[i].Name);
            if (i < typeParms.Length - 1)
                parmString.Append(", ");
            else
                parmString.Append(">");
        }
        generics = parmString.ToString();
    }
    ParameterInfo returnParm = m.ReturnParameter;
    string returnTypeName = processGenericTypeInformation(
                    returnParm.ParameterType);
    Console.Write(returnTypeName);
    Console.Write(" ");
    Console.Write(m.Name);
    Console.Write(generics);
    Console.Write("(");
    if (m.GetCustomAttributes(typeof(ExtensionAttribute), false)
        .Length == 1)
        Console.Write("this ");
    int count = 1;
    ParameterInfo[] parms = m.GetParameters();
    foreach (var p in parms)
    {
        var typeName = processGenericTypeInformation(p.ParameterType);
        Console.Write("{0} {1}", typeName, p.Name);
        if (count++ < m.GetParameters().Count())
            Console.Write(", ");
        }
        Console.WriteLine(")");
    }
}
private static string processGenericTypeInformation(Type genericType)
{
    if (!genericType.IsGenericType)
        return genericType.Name;
    string parmTypeName = genericType.Name.Remove(
                    genericType.Name.IndexOf('`'));
    Type typeInfo = genericType.GetGenericArguments()[0];
    string typeName = processGenericTypeInformation(typeInfo);
    parmTypeName = string.Format("{0}<{1}", parmTypeName, typeName);
    int i = 1;
    while (i < genericType.GetGenericArguments().Length)
    {
        typeInfo = genericType.GetGenericArguments()[i];
        typeName = processGenericTypeInformation(typeInfo);
        parmTypeName += ", " + typeName;
        i++;
    }
    parmTypeName += ">";
    return parmTypeName;

}
```

There's quite a bit of code here. The first section finds the type System.Linq.Enumerable. If that type is not found, the method returns without further work:

```
Type[] types = assembly.GetExportedTypes();

Type enumerable = null;

foreach (Type t in types)

    if (t.Name == "Enumerable")

        enumerable = t;

if (enumerable == null)

{

    Console.WriteLine("No data found.");

    return;

}
```

Once it finds the Enumerable type, the next step is to find all the public static methods. Those are the methods that make up the LINQ APIs:

```
MethodInfo[] methods = enumerable.GetMethods(BindingFlags.Public |

                BindingFlags.Static);

foreach (var m in methods)

{
```

For each method, the code finds the elements that make up the method signature. That includes the return type, the method name, the type parameters for the method, and all parameters. For each parameter, the code also finds the type information, including generic type parameters and the name for each parameter.

The first section of code finds the list of generic parameters to the method, if there are generic parameters to the method:

```
string generics = "";
if (m.IsGenericMethodDefinition)
{
    Type[] typeParms = m.GetGenericArguments();
    StringBuilder parmString = new StringBuilder();
    parmString.Append("<");
    for (int i = 0; i < typeParms.Length; i++)
    {
        parmString.Append(typeParms[i].Name);
        if (i < typeParms.Length - 1)
            parmString.Append(", ");
        else
            parmString.Append(">");
    }
    generics = parmString.ToString();
}
```

The next few lines retrieve the return type for the method. The return type might be a generic type, so there is a helper method I'll discuss a little later to retrieve generic type information from the return type.

Once the return type is printed, the code prints out the method name, including the generic type parameters found earlier.

Finally, the code determines and prints the parameter information. Once again, those parameters may be generic types, so that same helper method retrieves the generic parameters.

```
Console.Write("(");
if (m.GetCustomAttributes(typeof(ExtensionAttribute), false)
    .Length == 1)

    Console.Write("this ");

int count = 1;

ParameterInfo[] parms = m.GetParameters();

foreach (var p in parms)

{

    var typeName = processGenericTypeInformation(p.ParameterType);

    Console.Write("{0} {1}", typeName, p.Name);

    if (count++ < m.GetParameters().Count())

        Console.Write(", ");

}

Console.WriteLine(")");
```

There's also that helper method to examine a type object that may represent a generic type, and prints out its name, formatted with any generic type parameters. This method is recursive, because generic types may contain generic types as their parameters (consider Expression<Func<IEnumerable<T>>>. The recursion handles that case.

I didn't go through each line in detail, because these methods show familiar C# syntax that should be obvious to the vast majority of C# developers. Rather, I emphasized the fragments of the code that will change the most when the algorithm moves to LINQ. There are lots of loops. There are if statements sprinkled throughout the code. There are loops with if statements and break statements. This complicates the code, and makes it harder to understand the core algorithms.

New Tools, New Expressiveness

Figure 2 shows the same algorithm, but this time leveraging the APIs and syntax changes that are available in modern C#. This version uses many of the LINQ APIs and much of the LINQ query syntax.

Figure 2: Finding all the LINQ APIs using LINQ.

```
private static void LinqAnalysis(Assembly assembly)

{

    var enumerableType = assembly.GetExportedTypes()
    .Single(t => t.Name == "Enumerable");
var methods = from m in enumerableType.GetMethods(
                    BindingFlags.Public | BindingFlags.Static)
                let isExtension = m.GetCustomAttributes(
                    typeof(ExtensionAttribute), false)
                    .Any()
                select new
                {
                    returnParm = m.ReturnType.FormatTypeName(),
                    name = m.Name +
                    (m.GetGenericArguments().Any() ?
                        "<" + m.GetGenericArguments()
                            .Select(t => t.Name)
                            .CommaSeparated() + ">"
                            : ""),
                    parameters = "(" +
                    (isExtension ? "this " : "") +
                    m.GetParameters()
                        .Select(p => p.ParameterType.FormatTypeName()
                            + " " + p.Name)
                        .CommaSeparated()
                        + ")"
                };

    foreach (var m in methods)

       Console.WriteLine("{0} {1}{2}", m.returnParm, m.name, m.parameters);

}

public static class LINQHelpers

{

    public static string FormatTypeName(this Type genericType)

    {

        if (!genericType.IsGenericType)

            return genericType.Name;

        var parmTypeName = genericType.Name.Remove(genericType.Name

            .IndexOf('`'));

        var genericTypes = (from g in genericType.GetGenericArguments()

                        select g.FormatTypeName())

                        .CommaSeparated();

        return string.Format("{0}<{1}>", parmTypeName, genericTypes);

    }

    public static string CommaSeparated(this IEnumerable<string> items)

    {

        return items.Any()

            ? items.Aggregate((partialResult, item) =>

                string.Format("{0}, {1}", partialResult, item))
            : "";

    }

}
```

As we walk through the code, I'll explain the new concepts. By the end, you'll have a greater understanding of LINQ and how you can leverage it in your daily activities.

The first line finds the type object for the System.Linq.Enumerable class:

```
var enumerableType = assembly.GetExportedTypes()

    .Single(t => t.Name == "Enumerable");
```

The Single() method returns the one object in the source sequence that satisfies the test, in this case, where the Name property is "Enumerable". If the number of objects passing the test isn't exactly 1, the Single() method throws an exception.

The next statement builds a query that creates the sequence of objects that contain formatted information about each API. It's quite a long query, but it's understandable if you break it down. First, it defines the source for the query, the sequence of public static methods:

```
var methods = from m in enumerableType.GetMethods(

                BindingFlags.Public | BindingFlags.Static)
```

The next clause is a let clause, which stores a preliminary result. Here, it stores whether or not the method is an extension method. The Any() method returns true if a sequence has any elements, and false for an empty sequence.

```
let isExtension = m.GetCustomAttributes(

    typeof(ExtensionAttribute), false)

    .Any()
```

The last part of the query is the select clause. It creates an instance of an anonymous type that contains 3 strings: the return type, the method name, and the parameter list. It uses the object initializer syntax. The return parameter is found using an extension method that I'll explain a bit later. The name of the method is found using another LINQ query, written using method call syntax:

```
vname = m.Name +

(m.GetGenericArguments().Any() ?

    "<" + m.GetGenericArguments()
```

```
        .Select(t => t.Name)

        .CommaSeparated() + ">"

        : ""),
```

Retrieving the name of the method should be obvious. GetGenericArguments() returns the sequence of Type objects that represent generic type parameters for the method. The Select() call returns the name for each generic type parameter. CommaSeparated() is another extension method we'll look at later. It does what its name implies: convert a sequence of strings into a comma separated list. The string concatenation puts the list of generic type parameters between angle brackets.

The code that initializes the parameters utilizes another LINQ query using the method call syntax:

```
parameters = "(" + (isExtension ? "this " : "") +

    m.GetParameters()

    .Select(p => p.ParameterType.FormatTypeName()

        + " " + p.Name)

    .CommaSeparated()

    + ")"
```

The Select call returns a sequence of strings containing the parameter type and the parameter name. CommaSeparated() formats those items into a comma separated list. The rest of the code adds the 'this' keyword if the method is an extension method, and wraps the entire thing in parentheses.

Finally, let's examine the helper extension methods that format the generic type parameters into a string, and create the comma separated list.

CommaSeparated() is simpler: It uses the Aggregate method to build the list:

```
public static string CommaSeparated(this IEnumerable<string> items)

{

    return items.Any()

        ? items.Aggregate((partialResult, item) =>

            string.Format("{0}, {1}", partialResult, item))

        : "";
}
```

Aggregate() is another member of the Enumerable class. It returns a single result based on some aggregation of the input sequence. Here, each input string is appended to the current result, along with a comma for a separator. Aggregate() is likely the tool of choice when you are building a single result from an input sequence.

FormatTypeName() examines a type to return the type name as you would see it in code. If the type is not generic, the type name is returned. Otherwise, the extra characters tacked on in the meta data are removed, and the generic type parameters are added, surrounded by angle brackets. In production code, I would almost certainly use the method call syntax. For example purpose, I mixed the query syntax with the method call syntax just to show you that they are equivalent. Once again, CommaSeparated() manages concatenation of the strings.

For clarity, I used string concatenation on this sample. I've recommended using StringBuilder, or string.Format in most cases. This is one of those cases where it's ok to break that rule, as the number of calls are rather small, and the extra garbage collection won't have that much of an affect.

```
public static string FormatTypeName(this Type genericType)

{

  if (!genericType.IsGenericType)

    return genericType.Name;

  var parmTypeName = genericType.Name.Remove(genericType.Name

    .IndexOf('`'));

  var genericTypes = (from g in genericType.GetGenericArguments()

            select g.FormatTypeName())

            .CommaSeparated();

  return string.Format("{0}<{1}>", parmTypeName, genericTypes);

}
```

Some closing thoughts

Examine the two figures for readability and understanding. If you're not using LINQ as part of your regular practice, that version may appear a bit more foreign. However, the more you use LINQ or read LINQ code, the easier it is to comprehend. Anders has often said that LINQ and the other functional concepts added to C# make it easier to 'concentrate on the what instead of the how' of programming.

The same is true when you read code written by others. When you read the LINQ version of the code, notice how much more the what is highlighted as opposed to the how. The LINQ code replaces loops and conditionals with query expressions and method calls. Reading the code reveals more about what it's doing than how it is accomplishing it. This is especially true for the algorithms that combine lists of items, either the generic type parameters or the parameter lists. The LINQ version quickly reveals that the code is joining strings into a comma separated list. The pre-LINQ version quickly reveals all the edge cases of looping through the list to perform the join: don't add a comma after the last element, put the right bracket at the end or beginning of the list and so on.

That same emphasis shows up when I wrote the code. I finished the LINQ version in less than half the time it took to write the non-LINQ version. Must of the extra time was dealt with those edge cases. There are separate paths for one parameter vs. more than one parameter. There are separate paths for multiple type parameters and single type parameters. The non-LINQ version needs more checks for empty lists and null return values. LINQ queries and methods behavior reasonably for empty input sequences.

As a final comment on the maintainability of LINQ based code, let's add one feature. Let's sort the output by method name, and then by number of parameters. That adds one line to the LINQ query (highlighted below):

```
var methods = from m in enumerableType.GetMethods(

                   BindingFlags.Public | BindingFlags.Static)

                let isExtension = m.GetCustomAttributes(

                   typeof(ExtensionAttribute), false)

                   .Any()

                orderby m.Name, m.GetParameters().Count()

                select new

                {

                   returnParm = m.ReturnType.FormatTypeName(),

                   name = m.Name +

                   (m.GetGenericArguments().Any() ?

                      "<" + m.GetGenericArguments()

                         .Select(t => t.Name)
```

```
          .CommaSeparated() + ">"
          : ""),
      parameters = "(" +
      (isExtension ? "this " : "") +
      m.GetParameters()
        .Select(p => p.ParameterType.FormatTypeName()
          + " " + p.Name)
        .CommaSeparated()
        + ")"
    };
```

I'm not going to add sorting to the non-LINQ version, as it's just not a high enough priority to justify the work: I'd have to allocate temporary storage. I'd have to write the custom comparison function. It would probably add another page to the output in **Figure 1**. By leveraging LINQ, I can accommodate more new feature requests in the scope of a project.

Ever since its release, I've been explaining LINQ to developers. As developers learn and adapt to the new capabilities, they have found benefits in productivity and maintainability for their code. If you're part of the minority that has not yet adopted LINQ as part of your regular development regimen, I hope this has given you another reason to adopt LINQ.