# Introducing LINQ: Language-Integrated Query

**By Anthony Sneed**

**Problem: Data != Objects**

Developers live in two worlds: The world of objects and the world of data. One of the things that makes developers' lives more difficult is the impedance mismatch between these two worlds and the inordinate amount of plumbing code required to bridge the gap between them. For example, we might want to retrieve customer data from a relational database, display it to the user for updating, then save those changes back to the database. The traditional approach is to rely on an API like ADO.NET for the data-centric operations, potentially sacrificing on type-safety and compiler syntax checking, and utilize object-oriented programming techniques for other parts of our application. Where these two worlds intersect in our application, we are forced to deal with differences between the SQL and CLR type systems, how entity relations work, as well as other incompatibilities. Conversely, we may wish to deal with objects in a data-centric fashion, filtering, sorting and grouping collections of objects, but this means we may end up with code that is difficult to maintain because it obscures what we want to do at a higher level.

**The Missing LINQ**

Not wanting us to languish in this no man's land, the good folks at Microsoft, led by Anders Hejlsberg (chief architect of the C# programming language), have created the LINQ Project. LINQ stands for Language-Integrated Query and seeks to make query a first-class citizen of the programming language. Microsoft plans to incorporate LINQ into C# 3.0 and Visual Basic 9.0, both of which will be released with Visual Studio 2008 (code-named Orcas). By introducing a SQL-like syntax, LINQ allows you to interrogate an in-memory collection (anything implementing IEnumerable<T>), filter the results, group, sort, transform and perform a number of other data-centric operations. Take, for example, the following array of strings:

```csharp
string[] stooges = { "Moe", "Larry", "Curly", "Shemp" };
```

With C# 3.0's new query syntax, you can extract only strings with a length of 5, sort them alphabetically, and change the results to upper case:

```csharp
IEnumerable<string> query =
    from s in stooges
    where s.Length == 5
    orderby s
```

```
    select s.ToUpper();
```

In this snippet, `from s in stooges` indicates that you want to query the `stooges` collection and use `s` to represent each item in the sequence. The `select` clause determines the data you want returned and what shape it should take. Here we want to return each element converted to upper case. The `where` and `orderby` clauses, as you might guess, restrict the results to those with 5 characters and specify an ascending sort. Execution of the query is actually deferred until you iterate the query results, typically with a `foreach` statement, or by calling extension methods, such as **ToArray** or **ToList**, that perform the iteration internally (we'll talk about extension methods in just a bit). The following statement triggers the query execution:

```
foreach (string s in query) Console.WriteLine(s);
```

This produces the following output:

```
CURLY
LARRY
SHEMP
```

**Behind the Magic:  C# Language Enhancements**

Supporting language-integrated query is a set of powerful enhancements to the C# programming language, each of which is useful in its own right, but when taken together provide the basis for LINQ. The most significant of these is *extension methods*, which allow third parties to in essence add methods to a type, thereby augmenting the type's contract, while still allowing the type's author to provide his or her own implementations of these methods. Extension methods are nothing more than static methods defined in static classes, and (in C#) have the operator in front of the first method parameter. The System.Linq namespace contains what are called the *standard query operators*, which are extension methods that provide the basic functionality of LINQ. The C# compiler, in fact, converts query syntax (things like from, in, where, select) to extension method calls. One of these is the simplest query operator, **Where**:

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<T> Where<T>
```

```
            (this IEnumerable<T> source, Func<T, bool> predicate)
    {
            foreach (T item in source)
            {
                if (predicate(item)) yield return item;
            }
        }
    }
}
```

As you can see, the method accepts a source of type IEnumerable<T> and a predicate of the Func generic delegate type.  We can create an instance of the delegate as an anonymous method:

```
Func<string, bool> predicate = delegate(string s)
        { return s.Length == 5; };
```

The **Where** method simply iterates over the source, returning only items that pass the predicate test. Because it's defined as a static method, there's nothing to prevent you from calling it directly, like so:

```
IEnumerable<string> query = Enumerable.Where(stooges, predicate);
```

However, because **Where** is defined as an extension method, we can invoke it using instance syntax (note, however, that it's not really an instance method!):

```
IEnumerable<string> query = stooges.Where(predicate);
```

Extension methods are brought into scope when the namespace in which they are contained is imported with a using directive.  However, they are only used if no matching methods with the same name and signature exist on the target type or its base classes.  This lets you plug in your own query operators which take precedence over those in the System.Linq namespace.

Although you can define the predicate as an anonymous method (or a standalone method for that matter), C# provides a much more compact syntax, called a *lambda expression*.  If we were to re-write our anonymous method as a lambda expression and pass it as an argument, our call to **Where** would look like this (notice how the type of "s" is inferred and "return" is left out):

```
IEnumerable<string> query = stooges.Where(s => s.Length == 5);
```

The power of lambda expressions, however, is not just that they are compact (and hence more readable when passed as method parameters), but they can be compiled as either code or data.  When assigned to the generic type **Expression<T>**, the compiler emits an expression tree representing the code rather than IL that would execute the code.  Frameworks like LINQ to SQL can then analyze the expression tree to formulate SQL statements targeting a relational database.
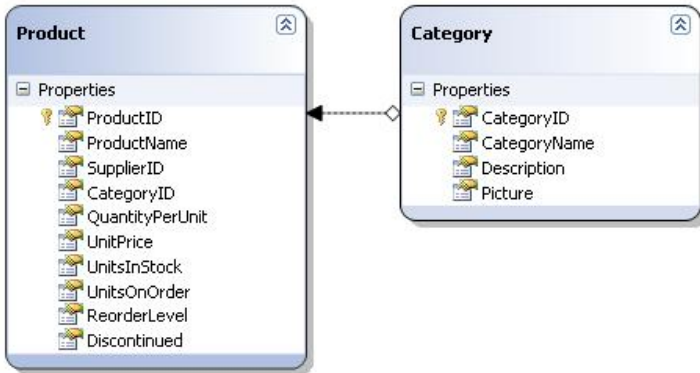
**Bridging the Gap: LINQ to SQL, Linq to XML**

Besides enabling you to query in-memory collections, LINQ is based on an extensible architecture that allows you to plug in a query processing engine to fetch data from any external source.  In fact, the first version of LINQ comes with both these capabilities, called LINQ to SQL and LINQ to XML, respectively.  Combining these LINQ extensions, you could retrieve a list of products from the Northwind database, place the result in an in-memory collection, massage the data, then persist the output to an XML file, all without having to translate "foreign" data constructs into CLR objects, while at the same time benefiting from Intellisense and compile-time syntax checking.  Let's start by querying relational data using LINQ to SQL:

```
NorthwindDataContext db = new NorthwindDataContext();

IEnumerable<Product> prodQuery =
    from p in db.Products
    where p.Category.CategoryName == "Beverages"
    select p;
```

The NorthwindDataContext class derives from DataContext, which does the heavy lifting to convert the query expression into SQL, execute it against the database, and return the results as a sequence of Product objects.  Where, might you ask, are these classes defined?  It just so happens that Visual Studio 2008 comes with special tool support for LINQ to SQL, including a class designer (which you get when adding a "LINQ to SQL Classes" item to a project).  The designer generates the NorthwindDataContext and Product classes when you drag the Products table from a data connection in the Server Explorer onto the class designer.  If you bring over multiple tables, relations between the tables are reflected in the class diagram and become object properties in the designer-generated code.

**Product**

Properties
- ProductID
- ProductName
- SupplierID
- CategoryID
- QuantityPerUnit
- UnitPrice
- UnitsInStock
- UnitsOnOrder
- ReorderLevel
- Discontinued

**Category**

Properties
- CategoryID
- CategoryName
- Description
- Picture

Notice how the where clause in our query references the CategoryName property of the Category class, which is shown to be a property of the Product class. When you execute this query, LINQ to SQL generates a SQL statement that includes a JOIN on the Products and Categories tables. The join is performed by SQL Server, which makes for efficient query execution. You can see this take place by setting the Log property of the data context to dump output to the console window or a text file.

```
SELECT * FROM [dbo].[Products] AS [t0]
LEFT OUTER JOIN [dbo].[Categories] AS [t1]
ON [t1].[CategoryID] = [t0].[CategoryID]
WHERE [t1].[CategoryName] = @p0
-- @p0: Input String (Size = 9; Prec = 0; Scale = 0) [Beverages]
```

LINQ to SQL also includes support for tracking changes to in-memory objects, then posting those changes back to the database, while at the same time maintaining concurrency control. (If you like, you can also specify stored procedures for the updates.) For example, we can store our product list in a local collection, change the unit price of an item, then call **SubmitChanges** to persist the new data back to SQL Server.

```
List<Product> products = prodQuery.ToList();

Product chai = (from p in products
      where p.ProductName == "Chai"
      select p).Single();

chai.UnitPrice = 20;

db.SubmitChanges();
```

The ORM functionality of LINQ to SQL is somewhat limited (for example, many-to-many relationships aren't fully supported and it only works with SQL Server).  But the power of LINQ really shines when combining LINQ to Objects, LINQ to SQL and LINQ to XML.  We can, for instance, take our list of beverages, sort them, and save them as an XML file.

```
IEnumerable<XElement> xmlQuery =
    from p in products
    orderby p.ProductName
    select new XElement("Product",
        new XAttribute("ProductName", p.ProductName),
        new XAttribute("UnitPrice", p.UnitPrice));

XElement bevXml = new XElement("Beverages", xmlQuery);

bevXml.Save("beverages.xml");
```

Notice how this code, unlike the DOM API, is element-centric, as opposed to document-centric, and that there's no need for XPath query strings.  Nice.  The beverages.xml file looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Beverages>
  <Product ProductName="Chai" UnitPrice="20.0000" />
  <Product ProductName="Chang" UnitPrice="19.0000" />
  <Product ProductName="Chartreuse verte" UnitPrice="18.0000" />
  <Product ProductName="Côte de Blaye" UnitPrice="263.5000" />
  [Remaining items elided for clarity ...]
</Beverages>
```

**Time to LINQ Up!**

There is much more to LINQ than what I have room to cover here.  Other C# features driving LINQ are things like object initializers, anonymous types and implicitly typed local variables.  There are standard query operators that perform grouping, joins, partitioning, union, intersection, aggregation and conversion, and other functions as well.  And while LINQ does not completely resolve impedance mismatch between objects and data, it is a step in the right direction, resulting in cleaner code that more clearly reflects the developer's intention.  It also makes you more productive by eliminating much of the plumbing code you would otherwise have to write, which, after all, is what frameworks are all about.