LINQ: The Future of Data Access in C# 3.0

By Joe Hummel

....
Publisher: **O'Reilly**
Pub Date: **October 01, 2006**
Print ISBN-10: **0-596-52841-8**
Print ISBN-13: **978-0-59-652841-6**
Pages: **64**

Table of Contents

Overview

Language Integrated Query (LINQ) is Microsoft's new technology for powerful, general purpose data access. This technology provides a fully-integrated query language, available in both C# 3.0 and VB 9.0, for high-level data access against objects, relational databases, and XML documents. In this Short Cut you'll learn about LINQ and the proposed C# 3.0 extensions that support it. You'll also see how you can use LINQ and C# to accomplish a variety of tasks, from querying objects to accessing relational data and XML. Best of all, you'll be able to test the examples and run your own code using the latest LINQ CTP, available free from Microsoft. This Short Cut includes a complete reference to the standard LINQ query operators.

LINQ: The Future of Data Access in C# 3.0

By Joe Hummel

Publisher: **O'Reilly**
Pub Date: **October 01, 2006**
Print ISBN-10: **0-596-52841-8**
Print ISBN-13: **978-0-59-652841-6**
Pages: **64**

Table of Contents

Copyright

Chapter 1. Learn LINQ and the C# 3.0 Features That Support It

Imagine writing SQL-like queries entirely in C#, with IntelliSense and compile-time type checking. Imagine queries as flexible as dynamically-generated SQL, yet as secure and efficient as calling stored procedures. Now take these same queries and execute them against an XML document or against a collection of data objects. While Microsoft isn't quite there yet, this is the goal of the LINQ project.

Language Integrated Query (LINQ) is a C# 3.0 API centered on data access. Developers focused on data access will be able to leverage the LINQ API to interoperate with a variety of data sources and vendors in a consistent, object-oriented fashion. The LINQ API is also extensible, as demonstrated by Microsoft's most important components of LINQ, LINQ to SQL and LINQ to XML. The former provides LINQ-like manipulation of relational databases and the latter of XML documents. You can expect future componentsboth from Microsoft and developers like yourselfto extend LINQ in new and interesting ways.

The vast majority of LINQ is made possible by language extensions in C# 3.0 and VB 9.0, which will appear in the upcoming 3.0 release of .NET. However, LINQ will also require a new version of the .NET Framework, which will follow the upcoming 3.0 release. We shall tentatively refer to this version as .NET 3.5. The implication is that developers will need to redeploy on .NET 3.5 to use LINQ.

NOTE

LINQ is still in development, and will be released in an upcoming version of .NET (3.5?). In the meantime, you can experiment with LINQ by downloading the latest CTP for Visual Studio 2005 here: http://msdn.microsoft.com/data/ref/linq.

The material for this Short Cut is based on the May 2006 CTP of LINQ. Since LINQ is beta software, I encourage you to install Visual Studio 2005 and LINQ in a virtual environment, such as one created with Virtual PC.

The source code accompanying this Short Cut is available at either of the following URLs: http://www.oreilly.com/catalog/language1 or http://pluralsight.com/drjoe/pdfs/pdfs.aspx. To get started yourself, install the latest LINQ CTP, startup Visual Studio 2005, expand the project types for C#, and select "LINQ Preview." To see lots of examples of LINQ in action, try this site: http://msdn.microsoft.com/vcsharp/future/linqsamples.

Chapter 2. A Quick Introduction to LINQ

Anders Hejlsberg, the chief architect of the C# programming language, summarizes LINQ quite nicely:

"*It's about turning query set operations and transforms into first-class concepts of the language*" (http://channel9.msdn.com/showpost.aspx?postid=114680)

Let's look at an example. Imagine we are writing a scheduling application for a hospital, where every night a doctor must be on call to handle emergencies. The main objects we are interacting with are of type `Doctor`, stored in a `Doctors` collection that inherits from `List<Doctor>`:

```
Doctors doctors = new Doctors();
```

Suppose we need a list of all the doctors living within the Chicago city limits. The obvious approach involves a simple iteration across the collection:

```
List<Doctor> inchicago = new List<Doctor>();
foreach (Doctor d in doctors)
  if (d.City == "Chicago")
    inchicago.Add(d);
```

In LINQ, this search reduces to the following query, with the IDE providing keyword highlighting, IntelliSense, and compile-time type checking:

```
var inchicago = from d in doctors where d.City == "Chicago" select d;
```

Need a list sorted by last name? How about:

```
var byname = from d in doctors
             where d.City == "Chicago"
             orderby d.FamilyLastName
             select d;
```

Also need a list sorted by pager number? In LINQ:

```
var bypager = from d in doctors
              where d.City == "Chicago"
              orderby d.PagerNumber
              select d;
```

Finally, suppose we need a count of how many doctors live in each city (not just Chicago, but every city for which at least one doctor lives). With LINQ, we can compute this result by grouping the doctors by city, much like we would in SQL:

```
var bycity = from d in doctors
             group d by d.City into g
             orderby g.Key
             select new { City = g.Key, Count = g.Count() };
```

These types of queries begin to reveal the real power of LINQ. The results are easily harvested by iterating across the query, for example:

```
foreach (var result in bycity)
  System.Console.WriteLine("{0}: {1}", result.City, result.Count);
```

This yields the following output for our set of doctors:

```
Chicago: 5
Elmhurst: 1
Evanston: 3
Oak Park: 2
Wilmette: 1
```

LINQ does not limit us to a read-only data access strategy. CRUD-like (create, read, update, and delete) operations are easily performed using the traditional object-oriented approach. For example, have the *Larsens* moved to the suburbs? If so, search for the appropriate Doctor objects and update the City:

```
var larsens = from d in doctors where d.FamilyLastName == "Larsen" select d;
foreach (var d in larsens)
  d.City = "Suburb";
```

Need to create a doctor? Add an instance of the Doctor class to the doctors collection. Need to delete a doctor? Query to find the corresponding Doctor object, and remove from the collection. In short, while the LINQ API does not provide direct support for CRUD, these operations are easily accomplished.

NOTE

Here are the definitions of the Doctors and Doctor classes we'll be using throughout this Short Cut.

```
public class Doctors : List<Doctor>
 {
   public Doctors()  // constructor (fills list with Doctors):
   {
     this.Add( new Doctor("mbl", "Marybeth", "Larsen", 52248,
                          "mlarsen@uhospital.edu", "1305 S. Michigan", "Chicago",
                          new DateTime(1998, 12, 1)) );
     this.Add( new Doctor("jl", "Joe", "Larsen", 52249,
                          "jlarsen@uhospital.edu", "1305 S. Michigan", "Chicago",
                          new DateTime(1999, 11, 1)) );
     this.Add( new Doctor("ch", "Carl", "Harding", 53113,
                          "charding@uhospital.edu", "2103 Oak St", "Evanston",
                          new DateTime(2000, 10, 1)) );
     .
     .
     .
   }
 }
 // continued
  public class Doctor : IComparable<Doctor>
// NOTE: IComparable NOT required by LINQ.
 {
        private string   m_Initials, m_GivenFirstName, m_FamilyLastName;
        private int      m_PagerNumber;
        private string   m_EmailAddress, m_StreetAddress, m_City;
        private DateTime m_StartDate;

        public string Initials
        { get { return m_Initials;       } set { m_Initials = value; }       }
        public string GivenFirstName
        { get { return m_GivenFirstName; } set { m_GivenFirstName = value; } }
        public string FamilyLastName
        { get { return m_FamilyLastName; } set { m_FamilyLastName = value; } }
        public int PagerNumber
        { get { return m_PagerNumber;    } set { m_PagerNumber = value; }    }
        public string EmailAddress
        { get { return m_EmailAddress;   } set { m_EmailAddress = value; }   }
        public string StreetAddress
        { get { return m_StreetAddress;  } set { m_StreetAddress = value; }  }
        public string City
        { get { return m_City;           } set { m_City = value; }           }
        public DateTime StartDate
        { get { return m_StartDate;      } set { m_StartDate = value; }      }

        public Doctor(string initials, string givenFirstName, string familyLastName,
              int pagerNumber, string emailAddress, string streetAddress,
string city,
              DateTime startDate)
        {
              this.m_Initials       = initials;
              this.m_GivenFirstName = givenFirstName;
```

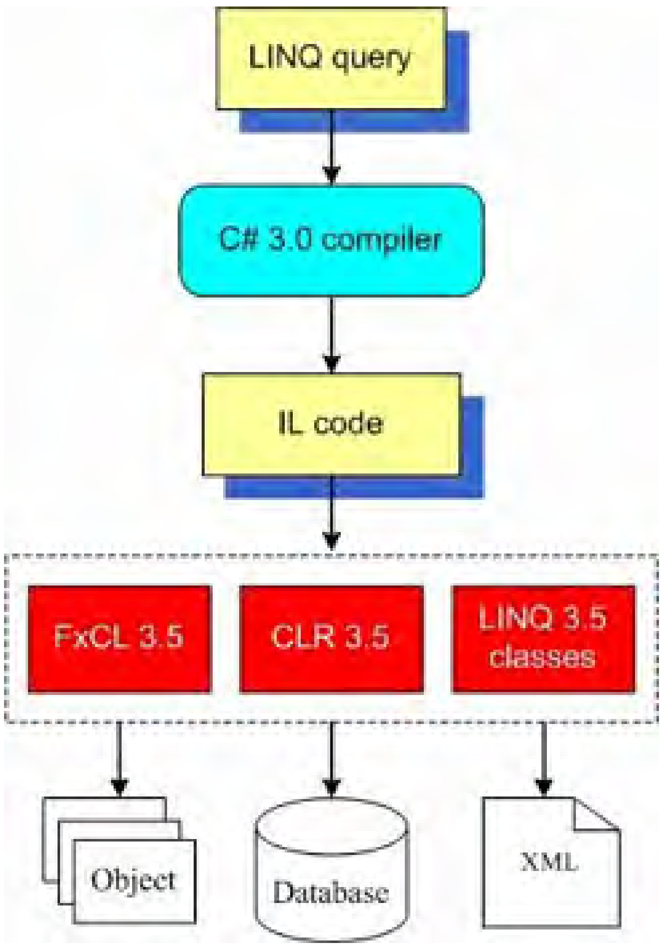
```
            this.m_FamilyLastName = familyLastName;
            this.m_PagerNumber    = pagerNumber;
            this.m_EmailAddress   = emailAddress;
            this.m_StreetAddress  = streetAddress;
            this.m_City           = city;
            this.m_StartDate      = startDate;
        }

        public override bool Equals(object obj)  // NOTE: Equals NOT required by LINQ.
        {
            if (obj == null || this.GetType().Equals(obj.GetType()) == false)
              return false;
            Doctor other = (Doctor) obj;
            return this.m_Initials.Equals(other.m_Initials);
        }

        public override int GetHashCode()  // NOTE: GetHashCode NOT required by LINQ.
        { return this.m_Initials.GetHashCode(); }

        public int CompareTo(Doctor other)
        { return this.m_Initials.CompareTo(other.m_Initials); }
    }
```

PREV NEXT

Chapter 3. The LINQ Architecture

In scanning the examples in the previous section, you probably noticed use of the var keyword, as well as the use of the new keyword without an accompanying class name. What you see are two new features of C# 3.0, namely type inference and anonymous types, added to the language in support of LINQ. Additional language features include query expressions, lambda expressions, expression trees, extension methods, and object initializers, all of which will be explained in the following sections.

Figure 3-1. LINQ_Architecture



LINQ's architecture is designed around a high-level, collection-like abstraction. This means that LINQ queries are written in C# 3.0 against the notion of an object collection, and then translated by the compiler into IL for execution against a particular data source (see Figure 1). This provides many benefits, including:

- Ability to target different data sources, e.g. databases and XML documents.
- Ability to use LINQ with existing .NET 1.* and 2.0 objects.
- Ability to extend LINQ to support new classes and technologies.

As an analogy, consider the foreach loop. The foreach is merely syntactic sugar for a while loop written against the IEnumerable and IEnumerator interfaces. However, foreach represents a convenient and powerful abstraction, allowing intuitive iteration across different types of collections, today and in the future. It's now hard to imagine C# or VB without foreach.

PREV NEXT

Chapter 4. Supporting LINQ in C# 3.0

Generics, delegates, and anonymous methods in C# 2.0 provide insight into how LINQ is supported in C# 3.0. Let's consider once again the problem of finding all doctors living within the Chicago city limits. As shown earlier, here's the obvious approach using foreach:

```
List<Doctor> inchicago = new List<Doctor>();
foreach (Doctor d in doctors)
  if (d.City == "Chicago")
    inchicago.Add(d);
```

A more elegant approach currently available in C# 2.0 is to take advantage of the collection's FindAll method, which can be passed a *delegate* to a function that determines whether the doctor resides in Chicago:

```
public delegate bool Predicate<T>(T obj);  // pre-defined in .NET 2.0

public bool IsInChicago(Doctor obj)  // notice how this matches delegate signature
{
  return obj.City == "Chicago";
}
.
.
.

List<Doctor> inchicago = doctors.FindAll(new Predicate<Doctor>(this.IsInChicago));
```

FindAll iterates through the collection, building a new collection containing those objects for which the delegate-invoked function returns true.

A more succinct version passes an *anonymous method* to FindAll:

```
List<Doctor> inchicago = doctors.FindAll(delegate(Doctor d)
                        {
                          return d.City == "Chicago";
                        } );
```

The {} denote the body of the anonymous method; notice these fall within the scope of the () in the call to FindAll. The signature of the anonymous methodin this case a Boolean function with a single Doctor argumentis type-checked by the compiler to ensure that it matches the definition of the argument to FindAll. The compiler then translates this version into an explicit delegate-based version, assigning a unique name to the underlying method:

```
private static bool b__0(Doctor d)
{
  return d.City == "Chicago";
}

List<Doctor> inchicago = doctors.FindAll( new Predicate<Doctor>(b__0) );
```

While this has nothing to do with LINQ per se, this approach of translating from one abstraction to another exemplifies how LINQ integrates into C# 3.0.

PREV NEXT

4.1. Lambda Expressions

From the developer's perspective, *lambda expressions* in C# 3.0 are a straightforward simplification of anonymous methods. Whereas an anonymous method is an unnamed block of code, a lambda expression is an unnamed expression that evaluates to a single value. Given a value x and an expression f(x) to evaluate, the corresponding lambda expression is written

```
x => f(x)
```

For example, in C# 3.0 the previous call to FindAll with an anonymous method is significantly shortened by using a lambda expression:

```
List<Doctor> inchicago = doctors.FindAll(x => x.City == "Chicago");
```

In this case, x is a doctor, and f(x) is the expression x.City == "Chicago", which evaluates to true or false. For readability, let's substitute d for x:

```
List<Doctor> inchicago = doctors.FindAll(d => d.City == "Chicago");
```

Notice that this lambda expression matches the parameter and body of the anonymous method we saw earlier, minus the syntactic extras like {}.

Lambda expressions are equivalent to anonymous methods that return a value when invoked, and are thus interchangeable. In fact, the compiler translates lambda expressions into delegate-based code, exactly as it does for anonymous methods.

4.2. Type Inference

Interestingly, there is one very subtle difference between lambda expressions and anonymous methods: the latter require type information, while the former do not. Returning to our example, notice that the lambda expression

```
d => d.City == "Chicago"
```

does not specify a type for d. Without a type, the compiler cannot translate the lambda expression into the equivalent anonymous method:

```
delegate(?????? d)  // what type is the argument d?
{
  return d.City == "Chicago";
}
```

To make this work, C# 3.0 is actually *inferring* the type of d in the lambda expression, based on contextual information. For example, since doctors is of type List<Doctor>, the compiler can prove that in the context of calling FindAll

```
doctors.FindAll(d => d.City == "Chicago")
```

d must be of type Doctor.

Type inference is used throughout C# 3.0 to make LINQ more convenient, without any loss of safety or performance. The idea of *type inference* is that the compiler infers the types of your variables based on context, not programmer-supplied declarations. The compiler does this while continuing to enforce strict, compile-time type checking.

As we saw earlier when introducing LINQ, developers can take advantage of type inference by using the var keyword when declaring local variables. For example, the declarations

```
var sum = 0;
var avg = 0.0;
var obj = new Doctor(...);
```

trigger the inference of int for sum, double for avg, and Doctor for obj. The initializer in the declaration is used to drive the inference engine, and is required. The following are thus illegal:

```
var obj2;          // ERROR: must have an initializer
var obj3 = null;   // ERROR: must have a specific type
```

Assuming the inference is successful, the inferred type becomes permanent and compilation proceeds as usual. Apparent misuses of the type are detected and reported by the compiler:

```
var obj4 = "hi!";
.
.
.
obj4.Close();  // Oops, wrong object! (ERROR: 'string' does not contain 'Close').
```

Do not confuse var with the concept of a VB variant (it's not), nor with the concept of var in dynamic languages like JavaScript (where var really means *object*). In these languages the variable's type can change, and so type checking is performed at runtimeincreased flexibility at the cost of safety. In C# 3.0 the type cannot change, and all type checking is done at compile-time. For example, if the inferred type is object (as for obj6 below), in C# 3.0 you end up with an object reference of very little functionality:

```
object obj5 = "hi";  // obj5 references the string "hi!", but type is object
var    obj6 = obj5;  // obj6 also references "hi!", with inferred type object
.
.
.
string s1 = obj6.ToUpper();  // ERROR: 'object' does not contain 'ToUpper'
```

While developers will find type inference useful in isolation, the real motivation is LINQ. Type inference is critical to the success of LINQ since queries can yield complex results. LINQ would be far less attractive if developers had to explicitly type all aspects of their queries. In fact, specifying a type is sometimes impossible, e.g., when projections select new patterns of data:

```
      var query = from d in doctors
      where d.City == "Chicago"
      select new { d.GivenFirstName, d.FamilyLastName };  // type?

foreach (var r in query)
  System.Console.WriteLine("{0}, {1}", r.FamilyLastName, r.GivenFirstName);
```

In these cases, typing is better left to the compiler.

PREV NEXT

4.3. Anonymous Types and Object Initializers

Consider the previous query to find the names of all doctors living in Chicago:

```
var query = from d in doctors
            where d.City == "Chicago"
            select new { d.GivenFirstName, d.FamilyLastName };
```

The query interacts with complete Doctor objects from the doctors collection, but then projects only the doctor's first and last name. Look carefully at the syntax. We see the new keyword without a class name, and { } instead of ( ):

```
new { d.GivenFirstName, d.FamilyLastName }
```

The new operator is in fact instantiating an object, but the class is an *anonymous type*the C# 3.0 compiler is automatically creating an appropriate class definition with a unique name and substituting that name. The only constructor provided by the generated class is a default one (i.e. parameter-less), which favors serialization but complicates the issue of object initialization. This is the motivation for the { }, which denote C# 3.0's new *object initializer* syntax. Object initializers provide an inline mechanism for initializing objects. And in the case of anonymous types, object initializers define both the data members of the type and their initial values.

For example, our earlier object instantiation is translated to

```
new <Projection>f__12 { d.GivenFirstName, d.FamilyLastName }
```

where the compiler has generated

```
public sealed class <Projection>f__12
{
    private string _GivenFirstName;
    private string _FamilyLastName;

    public string GivenFirstName
    { get { return _GivenFirstName; } set { _GivenFirstName = value; } }
    public string FamilyLastName
    { get { return _FamilyLastName; } set { _FamilyLastName = value; } }

    public <Projection>f__12( )              // default constructor: empty
    { }
    public override string ToString()        // dumps contents of all fields
    { ... }
    public override bool Equals(object obj)  // all fields must be Equals
    { ... }
    public override int GetHashCode()
    { ... }
}
```

As you can see, the names and types of the properties are inferred from the initializers. If you prefer, you can assign your own names to the data members of the anonymous type. For example:

```
new { First = d.GivenFirstName, Last = d.FamilyLastName }
```

In this case the compiler generates:

```
public sealed class <Projection>f__13
```

```
{
      private string _First;
      private string _Last;

      public string First
      { get { return _First; } set { _First = value; } }
      public string Last
      { get { return _Last; } set { _Last = value; } }
      .
      .
      .
}
```

Anonymous types are being added to C# 3.0 primarily to support of LINQ, allowing developers to work with just the data they need in a familiar, object-oriented package. And courtesy of type inference, you can work with anonymous types in a safe, type-checked manner.

A current limitation of anonymous types is that they cannot be accessed outside the defining assemblywhat name would you use to refer to the class? In theory you could compile the code in a separate assembly and then use the generated name, but this is an obvious bad practice (and currently prevented by the fact that the generated name is invalid at the source code level). At issue is the design of N-tier applications, whose tiers communicate by passing data. If the data is packaged as an anonymous type, how do the other tiers refer to it? Since they cannot, anonymous types should be viewed as a technology for local use only.

PREV NEXT

4.4. Query Expressions

A LINQ query is called a *query expression*. Query expressions start with the keyword from, and are written using SQL-like *query operators* such as Select, Where, and OrderBy:

```
using System.Query;  // import standard LINQ query operators

// all doctors living in Chicago, sorted by last name, first name:
var chicago = from d in doctors
              where d.City == "Chicago"
              orderby d.FamilyLastName, d.GivenFirstName
              select d;

foreach (var r in chicago)
  System.Console.WriteLine("{0}, {1}", r.FamilyLastName, r.GivenFirstName);
```

By default, you must import the namespace System.Query to gain access to the standard LINQ query operators.

As noted earlier, type inference is used to make query expressions easier to write and consume:

```
      var chicago = from d in doctors ... select d;

foreach (var r in chicago) ... ;
```

But what exactly is a query expression? What type is inferred for the variable chicago above? Consider SQL. In SQL, a *select* query is a declarative statement that operates on one or more tables, producing a table. In LINQ, a query expression is a declarative expression operating on one or more IEnumerable objects, returning an IEnumerable object. Thus, a query expression is an expression of iteration across one or more objects, producing an object over which you iterate to collect the result. For example, let's be type-specific in our previous declaration:

```
      IEnumerable<Doctor> chicago = from d in doctors ... select d;
```

LINQ defines query expressions in terms of IEnumerable<T> to hide implementation details (preserving flexibility!) while conveying in a strongly-typed way the key concept that a query expression can be iterated across:

```
foreach (Doctor d in chicago) ... ;
```

Of course, type inference conveniently hides this level of detail without any loss of safety or performance.

4.5. Extension Methods

So how exactly are query expressions executed? Query expressions are translated into traditional object-oriented method calls by way of *extension methods*. Extension methods, new in C# 3.0, extend a class without actually being members of that class. For example, consider the following query expression:

```
using System.Query;

// initials of all doctors living in Chicago, in no particular order:
var chicago = from d in doctors
              where d.City == "Chicago"
              select d.Initials;
```

Using extension methods, this query can be rewritten as follows:

```
using System.Query;

var chicago = doctors.
              Where(d => d.City == "Chicago").
              Select(d => d.Initials);
```

This statement calls two methods, Where and Select, passing them lambda expressionsyet these methods are not members of the Doctors class. How and why does this compile?

In C# 3.0, extension methods are defined as static methods in a static class, annotated with the System.Runtime.CompilerServices.Extension attribute. By importing the namespace that includes this static class, the compiler treats the extension methods as if they were instance methods. For example, the LINQ standard query operators are defined as extension methods in the class System.Query.Sequence. By importing the namespace System.Query, we gain access to the query operators as if they were members of the Doctors class.

Let's look at an extension method in more detail. Here's the signature for System.Query.Sequence.Where:

```
namespace System.Query
{
    public delegate ReturnT Func<ArgT, ReturnT>(ArgT arg);

    public static class Sequence
    {
        public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
                                              Func<T, bool> predicate)
        { ... }
        .
        .
        .
    }
}
```

The extension method Where returns an IEnumerable object by iterating across an existing IEnumerable object (*source*), applying a delegate-based Boolean function (*predicate*) to determine membership in the result set. Notice that both the first parameter and the return value are defined in terms of IEnumerable<T>, establishing the link between query expressions and the extension methods that drive them.

Given a query expression or a query written using extension methods, the C# 3.0 compiler translates these into calls to the underlying static methods. For example, either form of our query above is conceptually translated into the following:

```
var temp    = System.Query.Sequence.Where(doctors, d => d.City == "Chicago");
var chicago = System.Query.Sequence.Select(temp, d => d.Initials);
```

In reality, this form is skipped and the translation proceeds directly to C# 2.0-compatible code:

```
IEnumerable<Doctor> temp;     // doctors living in chicago
IEnumerable<string> chicago;  // initials of doctors living in chicago

temp    = System.Query.Sequence.Where<Doctor>( doctors,
          new Func<Doctor, bool>(b__0) );
chicago = System.Query.Sequence.Select<Doctor, string>( temp,
          new Func<Doctor, string>(b__3) );
```

with the lambda expressions translated into delegate-invoked methods:

```
private static bool b__0(Doctor d)    // lambda: d => d.City == "Chicago"
{ return d.City == "Chicago"; }

private static string b__3(Doctor d)  // lambda: d => d.Initials
{ return d.Initials; }
```

Notice the important role that type inference plays during this translation. Extension methods and their supporting types (IEnumerable, Func, etc.) are elegantly defined once using generics. Type inference is relied on to determine the type T involved in each aspect of the query, and to then qualify the generic appropriately. In this case we see the inference of both Doctor and string.

In C# 3.0, what differentiates an extension method from an ordinary static method? Observe that a query based on extension methods is converted to a static version by passing the object instance as the first argument:

```
doctors.Where(...) ==> System.Query.Sequence.Where(doctors, ...)
```

This is identical to the standard mechanism for calling instance methods, where the first parameter is a reference to the object itselfproviding a value for this. This logic explains C#'s choice of the this keyword in the signature of extension methods:

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source, ... )
```

In C# 3.0 it is the presence of the this keyword on the first parameter that identifies a static method as an extension method.

What if the class (such as Doctors) or one of its base classes contain methods that conflict with imported extension methods? To avoid unexpected behavior, all other methods take priorityextension methods have the lowest precedence when the compiler performs name resolution, and thus become candidates only after all other possibilities have been exhausted. If two or more imported namespaces yield candidate extension methods, the compiler reports the conflict as a compilation error.

PREV NEXT

4.6. Lazy Evaluation

Interestingly, you may be a bit surprised by the behavior of queries in LINQ. Let's look at an example. First, consider the following Boolean function that as a side-effect outputs the doctor's initials:

```
public static bool dump(Doctor d)
{
  System.Console.WriteLine(d.Initials);
  return true;
}
```

Now let's use dump in a simple query that will end up selecting all the doctors because the function always returns true:

```
var query = from d in doctors
            where dump(d)
            select d;
```

Here's the million dollar question: what does this query output? [*Nothing!*]

It turns out that the declaration of a query expression does just thatdeclares a query, but does *not* evaluate it. Evaluation in most cases is delayed until the results of the query are actually requested, an approach known as *lazy evaluation*. For example, the following statement outputs the initials of the first doctor:

```
query.GetEnumerator().MoveNext();  // outputs ==> "mbl"
```

By "moving" to the first element in the result set, we trigger a call to dump based on the first doctor, which outputs the doctor's initials. Since dump returns true, this doctor satisfies the where condition, the doctor is considered part of the result set, and MoveNext returns because it has successfully moved to the first element in the result set. Hence the initials of the first doctor, and only the first doctor, are output. [*What do you think happens if dump returns false for the first doctor?*]

To output the initials of all the doctors, we iterate across the entire result:

```
foreach (var result in query)  // "mbl", "jl", "ch", ...
  ;
```

Again, it's sufficient to simply request each result in order to trigger evaluation (and in this case the output of the doctor's initials), we do not have to access the resulting value.

You are probably wondering how lazy evaluation is supported in C# 3.0. In fact, the support was introduced in C# 2.0 via the yield construct. For example, here's the complete implementation of the standard LINQ query operator Where that we discussed earlier:

```
public static class Sequence
{
      public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
                                          Func<T, bool> predicate)
      {
        foreach (T element in source)
          if (predicate(element))
            yield return element;
      }
}
```

The *yield return* pattern produces the next value of the iteration and then returns. In response to yield, the C# compiler generates the necessary code (essentially a nested iterator class) to implement IEnumerable and enable iteration to continue where it left off.

NOTE

To learn more about how things work and what the C# compiler is doing, I highly recommend Lutz Roeder's *Reflector* tool as a way to reverse-engineer your compiled code and see what's going on: http://www.aisto.com/roeder/dotnet.

One of the implications of LINQ's lazy evaluation is that query expressions are re-evaluated each time they are processed. In other words, the results of a query are not stored or cached in a collection, but lazily evaluated and returned on each request. The advantage is that changes in the data are automatically reflected in the next processing cycle:

```
foreach (var result in query)    // outputs initials for all doctors
  ;

doctors.Add( new Doctor("xyz", ...) );

foreach (var result in query)    // output now includes new doctor "xyz"
  ;
```

Another advantage is that complex queries can be constructed from simpler ones, and execution is delayed until the final query is ready (and possibly optimized!).

If you want to force immediate query evaluation and cache the results, the simplest approach is to call the `ToArray` or `ToList` methods on the query. For example, the following code fragment outputs the initials of all doctors *three* consecutive times:

```
var cache1 = query.ToArray();  // evaluate query & cache results as an array

doctors.Add( new Doctor("xyz", ...) );

System.Console.WriteLine("###");
var cache2 = query.ToList();   // evaluate query again & cache results as a list

System.Console.WriteLine("###");
System.Console.WriteLine( cache1.GetType() );
System.Console.WriteLine( cache2.GetType() );

foreach (var result in cache1)  // output results of initial query:
  System.Console.WriteLine(result.Initials);

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

The second cache contains the new doctor, but the first does not. Here's the output:

```
mbl
jl
.
.
ks
###
mbl
jl
.
.
ks
xyz
###
Doctor[]
System.Collections.Generic.List`1[Doctor]
mbl
jl
.
.
ks
###
```

Most (but not all) of the standard LINQ query operators are lazily evaluated. The exception are operators that return a single (*scalar*) value, such as `Min` and `Max`, which might as well produce the value instead of a collection containing that value:

```
int minPager = doctors.Min(d => d.PagerNumber);
int maxPager = doctors.Max(d => d.PagerNumber);

System.Console.WriteLine("Pager range: {0}..{1}", minPager, maxPager);
```

The section on standard LINQ query operators will make note of which operators are lazily evaluated, and which are not. But first, let's take a deeper look at what LINQ offers.

Chapter 5. Applying LINQ

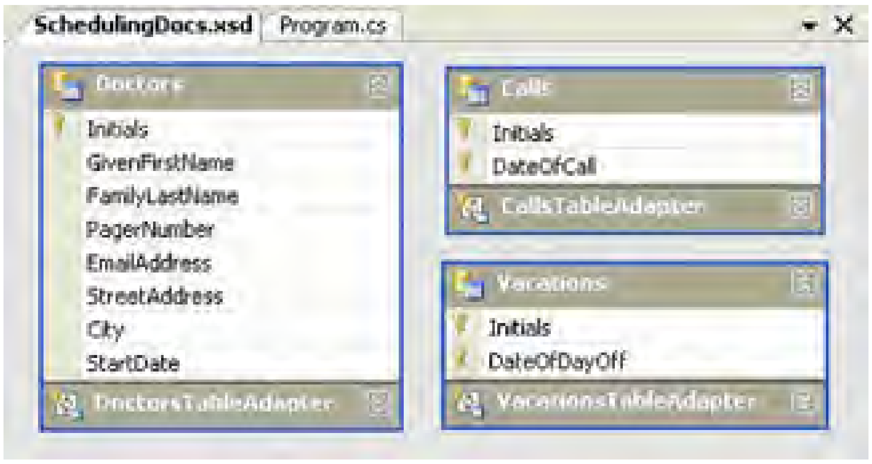
LINQ is really the centerpiece of a wide range of data access technologiesfrom datasets to databases, text files to XML documents, objects to object graphs. The extensibility of LINQ is one of its most elegant features, allowing LINQ to query just about any enumerable data source.

In this section we'll look at some of the more interesting ways you can use LINQ to query DataSets, databases, XML documents, text files, and more.

5.1. LINQ to DataSets

LINQ supports the querying of both typed and untyped DataSets. Expanding on our theme of a hospital scheduling application, suppose we have a namespace DataSets with a typed dataset SchedulingDocs containing three tables: *Doctors*, *Calls*, and *Vacations*. The Doctors table contains one record for each doctor, the Calls table keeps track of which doctor is on call each day, and the Vacations table makes note of vacation requests. The DataSet is summarized in Figure 2.

Figure 5-1. Tables in the *SchedulingDocs* typed DataSet



Let's assume an instance of SchedulingDocs has been created and filled:

```
DataSets.SchedulingDocs ds = new DataSets.SchedulingDocs();  // create dataset
.
.  // open connection to a database and fill each table?
.
```

To find all the doctors living within Chicago, the query is exactly as we've seen before:

```
var chicago = from d in ds.Doctors
              where d.City == "Chicago"
              select d;
```

In this case ds.Doctors denotes a DataTable object, and d represents a DataRow object. Since the DataSet is typed, these objects are strongly-typed as DoctorsTable and DoctorsRow, respectively.

NOTE

For completeness, here's the code to create and fill an instance of SchedulingDocs, declared within the DataSets namespace:

```
using System.Data.Common;
using DataSets;

SchedulingDocsTableAdapters.DoctorsTableAdapter    doctorsAdapter;
SchedulingDocsTableAdapters.CallsTableAdapter      callsAdapter;
SchedulingDocsTableAdapters.VacationsTableAdapter  vacationsAdapter;
SchedulingDocs       ds;
DbProviderFactory  dbFactory;

dbFactory        = DbProviderFactories.GetFactory(providerInfo);
ds               = new SchedulingDocs();
doctorsAdapter   = new SchedulingDocsTableAdapters.DoctorsTableAdapter();
```

```
     callsAdapter     = new SchedulingDocsTableAdapters.CallsTableAdapter();
     vacationsAdapter = new SchedulingDocsTableAdapters.VacationsTableAdapter();

     using (DbConnection dbConn = dbFactory.CreateConnection())
     {
       dbConn.ConnectionString = connectionInfo;
       dbConn.Open();

       doctorsAdapter.Fill(ds.Doctors);
       callsAdapter.Fill(ds.Calls);
       vacationsAdapter.Fill(ds.Vacations);
     } //dbConn.Close();
```

LINQ supports the notion of joins, including inner and outer joins. For example, let's join the Doctors and Calls tables to see which doctors are scheduled in the month of October 2006:

```
var oct2006 = (
                from d in ds.Doctors
                join c in ds.Calls
                on d.Initials equals c.Initials
                where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                      c.DateOfCall <= new DateTime(2006, 10, 31)
                orderby d.Initials
                select d.Initials
               )
              .Distinct();
```

This query expression uses a number of standard LINQ query operators, including `Join`, `OrderBy`, and `Distinct`; `Join` implements an *inner equijoin*.

LINQ is not limited to yielding tabular data, but will produce hierarchical results as appropriate. For example, suppose we want to know not only which doctors are on call in October 2006, but also the dates. In this case, we join the Doctors and Calls tables, now grouping the results:

```
var oct2006 = from d in ds.Doctors
              join c in ds.Calls
              on d.Initials equals c.Initials
              where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                    c.DateOfCall <= new DateTime(2006, 10, 31)
              group c by d.Initials into g
              select g;
```

Notice we group the calls ("c") on a per doctor basis ("d"). For each scheduled doctor, this yields an enumerable collection of calls. Here's how we process the query:

```
            foreach (var group in oct2006)
{
  System.Console.WriteLine("{0}: ", group.Key);
  foreach (var call in group)
     System.Console.WriteLine("  {0}", call.DateOfCall.ToString("dd MMM yyyy"));
  System.Console.WriteLine("  calls = {0}", group.Count());
}
```

The hierarchical output appears as follows:

```
jl:
  02 Oct 2006
  calls = 1
mbl:
  01 Oct 2006
  31 Oct 2006
  calls = 2
.
.
.
```

How about we re-order the results by those working the most, and select just the data we need in the result set:

```
var oct2006 = from d in ds.Doctors
              join c in ds.Calls
              on d.Initials equals c.Initials
              where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                    c.DateOfCall <= new DateTime(2006, 10, 31)
              group c by d.Initials into g
              orderby g.Count() descending
              select new { Initials = g.Key,
                           Count = g.Count(),
                           Dates = from c in g
                                   select c.DateOfCall };

foreach (var result in oct2006)
{
  System.Console.WriteLine("{0}:", result.Initials);
  foreach (var date in result.Dates)
    System.Console.WriteLine("  {0}", date.ToString("dd MMM yyyy"));
  System.Console.WriteLine("  calls = {0}", result.Count);
}
```

By projecting just the needed data, we can do things like data-bind the result for ease of display. The trade-off is that this *potentially* requires another set of custom objects to be instantiated. However, keep in mind that the design of LINQ enables the query operators to optimize away unnecessary object creation, much like compilers routinely eliminate unneeded code. This is especially true when applying LINQ in other situations, such as against a database (i.e., "*LINQ to SQL*").

PREV NEXT

5.2. LINQ to SQL

Instead of executing our queries against a DataSet, suppose we want to execute against the database directly? No problem. With LINQ to SQL, we change only the target of our query:

```
      Databases.SchedulingDocs db = new Databases.SchedulingDocs(connectionInfo);

var oct2006 = ( // find all doctors scheduled for October 2006:
                from d in db.Doctors
                join c in db.Calls
                on d.Initials equals c.Initials
                where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                      c.DateOfCall <= new DateTime(2006, 10, 31)
                orderby d.Initials
                select d.Initials
               )
               .Distinct();
```

In this case, SchedulingDocs is a class denoting a SQL Server 2005 database named *SchedulingDocs*. This class, and its associated entity classesD o ctors, Calls, and Vacationswere automatically generated by LINQ's *SQLMetal* tool to represent the database and its tables. As you would expect, the query expression is lazily evaluated, waiting for the query to be consumed:

```
      foreach (var initials in oct2006)  // execute the query:
  System.Console.WriteLine("{0}", initials);
```

The query is now translated into parameterized SQL, sent to the database for execution, and the result set produced. How efficient is the generated SQL? In this case (and the May CTP of LINQ), a single select statement is executed against the database:

```
SELECT DISTINCT [t0].[Initials]
  FROM [Doctors] AS [t0], [Calls] AS [t1]
  WHERE ([t1].[DateOfCall] >= @p0) AND
        ([t1].[DateOfCall] <= @p1) AND
        ([t0].[Initials] = [t1].[Initials])
  ORDER BY [t0].[Initials]
```

NOTE

In the May CTP of LINQ, SQLMetal is provided as a command-line tool. To run, first open a command window and cd to the install directory for LINQ (most likely *C:\Program Files\LINQ Preview\Bin*). Now ask SQLMetal to read the metadata from your database and generate the necessary class files. For the SQL Server 2005 database named SchedulingDocs, the command is:

```
  C:\...\Bin>sqlmetal /server:. /database:SchedulingDocs
/language:csharp /code:SchedulingDocs.cs /namespace:Databases
```

This will generate a source code file *SchedulingDocs.cs* in the current directory.

Let's look at a more complex query that computes the number of calls for every doctor in the month of October 2006. Recall that an inner join produces results for only those doctors that are working:

```
var oct2006 = from d in db.Doctors
               join c in db.Calls
               on d.Initials equals c.Initials
               where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                     c.DateOfCall <= new DateTime(2006, 10, 31)
               group c by d.Initials into g
               select new { Initials = g.Key, Count = g.Count() };
```

An *outer join* is needed to capture the results for all doctors, whether scheduled or not. Outer joins are based on LINQ's *join ... into* syntax:

```
var allOct2006 = from d1 in db.Doctors       // join all doctors
                 join d2 in oct2006          // with those working in Oct 2006
                 on d1.Initials equals d2.Initials
                 into j
                 from r in j.DefaultIfEmpty()
                 select new { Initials = d1.Initials,
                              Count = (r == null ? 0 : r.Count) };
```

This left outer join produces a result set "*into*" j, which is then enumerated across using the sub-expression "*from r in ...*". If a given doctor is working, then r is the joined result; if the doctor is not working then the result is empty, in which case j.DefaultIfEmpty() returns null. For each doctor, we then project their initials and the number of calls they are workingeither 0 or the count from the inner join. Iterating across the query:

```
foreach (var result in allOct2006)
    System.Console.WriteLine("{0}: {1}", result.Initials, result.Count);
```

Yields:

```
ay: 7
bb: 0
ch: 3
.
.
.
```

When the query is executed, the following SQL is sent to the database (this is a test intended for the SQL wizards in the audience):

```
SELECT [t7].[Initials], [t7].[value] AS [Count]
FROM (
    SELECT
        (CASE
            WHEN [t4].[test] IS NULL THEN 0
              ELSE (
                SELECT COUNT(*)
                FROM [Doctors] AS [t5], [Calls] AS [t6]
                WHERE ([t4].[Initials] = [t5].[Initials]) AND
                      ([t6].[DateOfCall] >= @p0) AND
                      ([t6].[DateOfCall] <= @p1) AND
                      ([t5].[Initials] = [t6].[Initials])
              )
        END) AS [value], [t0].[Initials]
    FROM [Doctors] AS [t0]
    LEFT OUTER JOIN (
        SELECT 1 AS [test], [t3].[Initials]
        FROM (
            SELECT [t1].[Initials]
            FROM [Doctors] AS [t1], [Calls] AS [t2]
            WHERE ([t2].[DateOfCall] >= @p0) AND
                  ([t2].[DateOfCall] <= @p1) AND
                  ([t1].[Initials] = [t2].[Initials])
            GROUP BY [t1].[Initials]
        ) AS [t3]
    ) AS [t4] ON [t0].[Initials] = [t4].[Initials]
) AS [t7]
```

NOTE

In all fairness, it should be noted that the focus of this Short Cut is LINQ, and not particularly LINQ to SQL. For this reason, the picture painted of LINQ to SQL is quite superficial. For example, in LINQ to SQL, queries are translated to SQL and executed with SQL semantics. In comparison, most other LINQ queries are directly executed by .NET Framework objects with CLR semantics. Likewise, LINQ to SQL handles changes to the data quite differently than other flavors of LINQ. For more details, we encourage you to read the forthcoming Part 2 of this Short Cut series on LINQ (expected fall 2006), which focuses exclusively on LINQ to SQL. Watch for an announcement at http://oreilly.com

5.3. Create, Read, Update, and Delete with LINQ

Most of the discussion thus far has focused on data querying, and not data modification. Don't be mistaken, LINQ provides full support for read/write data access, commonly referred to as CRUD.

Interestingly, while data is read using an SQL-like query language, data modification is approached using more traditional, object-oriented mechanisms. For example, to schedule the doctor *mbl* for call on November 30, 2006 in our *SchedulingDocs* database, we do two things. First, we add a new row to the object representing the Calls table:

```
db.Calls.Add( new Databases.Calls{Initials="mbl",
                                  DateOfCall=new DateTime(2006, 11, 30} );
```

Second, we flush the change back to the database:

```
db.SubmitChanges();
```

The first step makes a local, in-memory change only; the second step is what triggers the underlying SQL (or stored procedure) to update the database. LINQ to SQL will automatically generate the appropriate SQL for inserts, updates, and deletes, or interoperate with your custom stored procedures.

To delete a call, we find the corresponding object, remove it from the table, and update the database:

```
var del = from c in db.Calls
          where c.Initials == "mbl" && c.DateOfCall == new DateTime(2006, 11, 30)
          select c;

foreach (var c in del)
  db.Calls.Remove(c);

db.SubmitChanges();
```

Since there is at most one doctor on call for any given day, we know the above query will return exactly one record. In this case we can use the standard query operator `Single`, passing a lambda expression for the search criteria:

```
var call = db.Calls.Single( c => c.Initials == "mbl" &&
                                 c.DateOfCall == new DateTime(2006, 11, 30) );
db.Calls.Remove(call);
db.SubmitChanges();
```

If it's possible that the search may fail, use the query operator `SingleOrDefault`, and check the query result for null.

Finally, to update existing data, the approach is (1) query to find the corresponding objects, (2) update those objects, and (3) flush the changes. For example, if the doctor *ay*'s pager number changes to 53301, we update the database as follows:

```
var ay = db.Doctors.Single( d => d.Initials == "ay" );
ay.PagerNumber = 53301;
db.SubmitChanges();
```

The same logic applies to other LINQ scenarios, such as XML documents and DataSets. For example, with the typed DataSet `SchedulingDocs` (see Figure 2), scheduling a doctor on call is simply a matter of adding a new row to the Calls table:

```
ds.Calls.AddCallsRow( "mbl", new DateTime(2006, 11, 30) );
```

Much like the database objects, DataSets are a local, in-memory collection of objects. To persist your changes, an updated DataSet must be written to some durable medium, such as the file system or a database:

```
dbConn.Open();
callsAdapter.Update( ds.Calls );
dbConn.Close();
```

Here we re-open the connection, update the database to match, and close the connectionthe equivalent of db.SubmitChanges(). The key difference is that in the case of LINQ to SQL, the SQLMetal tool generates the necessary code to update the underlying database. In the case of DataSets and XML documents (and other flavors of LINQ), it's typically our responsibility to load the data, and consequently to persist it back.

PREV NEXT

5.4. LINQ to XML

From its beginnings, LINQ was designed to manipulate XML data as easily as it manipulates relational data. LINQ to XML represents a new API for XML-based development, equivalent in power to *XPath* and *XQuery* yet far simpler for most developers to use.

For example, let's assume the data source for our hospital scheduling application is an XML document stored in the file *SchedulingDocs.xml*. Here's the basic structure of the document:

```
<?xml version="1.0" standalone="yes"?>
<SchedulingDocs>
  <Calls>
    <Call>
      <Initials>mbl</Initials>
      <DateOfCall>2006-10-01T00:00:00-05:00</DateOfCall>
    </Call>
    .
    .
    .
  </Calls>
  <Doctors>
    <Doctor>
      <Initials>ay</Initials>
      <GivenFirstName>Amy</GivenFirstName>
      <FamilyLastName>Yang</FamilyLastName>
      <PagerNumber>53300</PagerNumber>
      <EmailAddress>ayang@uhospital.edu</EmailAddress>
      <StreetAddress>1400 Ridge Ave.</StreetAddress>
      <City>Evanston</City>
    </Doctor>
    .
    .
    .
  </Doctors>
  <Vacations>
    <Vacation>
      <Initials>jl</Initials>
      <DateOfDayOff>2006-10-03T00:00:00-05:00</DateOfDayOff>
    </Vacation>
    .
    .
    .
  </Vacations>
</SchedulingDocs>
```

Using LINQ, we load this document as follows:

```
import System.Xml.XLinq;  // LINQ to XML

XElement root, calls, doctors, vacations;

root = XElement.Load("SchedulingDocs.xml");

calls     = root.Element("Calls");
doctors   = root.Element("Doctors");
vacations = root.Element("Vacations");
```

We now have access to the three main elements of the XML document: *calls*, *doctors*, and *vacations*. To select all the doctors, it's a simple query expression:

```
var docs = from doc in doctors.Elements()
           select doc;
```

And to find just those doctors living in Chicago:

```
var chicago = from doc in doctors.Elements()
```

```
              where doc.Element("City").Value == "Chicago"
              orderby doc.Element("Initials").Value
              select doc;
```

As you can see, querying XML documents with LINQ is conceptually the same as that of relational databases, DataSets, and other objects. The difference is that the structure of the XML document must be taken into account, e.g., in this case the document's hierarchical design and its use of elements over attributes.

An important aspect of LINQ is the ability to easily transform data into other formats. In the world of XML, transformation is commonplace given the need to create XML documents as well as translate from one schema to another. For example, suppose we need to produce a new XML document containing just the names of the doctors, with their initials as an attribute:

```
<?xml version="1.0" standalone="yes"?>
<Doctors>
  <Doctor Initials="bb">Boswell, Bryan</Doctor>
  <Doctor Initials="lg">Goldstein, Luther</Doctor>
  .
  .
  .
</Doctors>
```

This document is easily produced by the following query, which simply projects new *XElements*:

```
var docs = from doc in doctors.Elements()
           orderby doc.Element("FamilyLastName").Value,
                   doc.Element("GivenFirstName").Value
           select new XElement("Doctor",
                    new XAttribute("Initials", doc.Element("Initials").Value),
                    doc.Element("FamilyLastName").Value +
                    ", " +
                    doc.Element("GivenFirstName").Value);

XElement newroot = new XElement("Doctors", docs);
```

The last statement creates the root element `<Doctors>`, using the query to generate the `<Doctor>` sub-elements.

Finally, here's a real-world example of translating a text-based IIS logfile into an XML document. This example comes from a series of posts to the MSDN LINQ Project General Forum, "Transforming a TXT file into XML with LINQ to XML," http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=574140&SiteID=1. The logfile contains 0 or more lines of the form:

```
Time  IP-address  Method  URI  Status
```

For example:

```
#Software: Microsoft Internet Information Services 5.1
#Version: 1.0
#Date: 2006-06-23 12:37:18
#Fields: time c-ip cs-method cs-uri-stem sc-status
12:37:18 127.0.0.1 GET /cobrabca 404
12:37:25 127.0.0.1 GET /cobranca 401
.
.
.
```

Here's the LINQ query to produce an XML document from such a log:

```
var logIIS = new XElement("LogIIS",
               from line in File.ReadAllLines("file.log")
               where !line.StartsWith("#")
               let items = line.Split(' ')
               select new XElement("Entry",
                 new XElement("Time", items[0]),
                 new XElement("IP", items[1]),
                 new XElement("Url", items[3]),
                 new XElement("Status", items[4])
               )
);
```

LINQ over a text file? The next section will discuss this, and other examples, in more detail.

NOTE

The general focus of this Short Cut precludes an in-depth treatment of LINQ to XML. For more details, we encourage you to read the forthcoming Part 3 of this Short Cut series (expected Q4 2006), which focuses exclusively on LINQ to XML. Watch for an announcement at http://oreilly.com

PREV NEXT

5.5. LINQ to IEnumerable

One of the elegant design aspects of LINQ is that queries can be executed against any enumerable data source. If an object implements IEnumerable, then LINQ can access the data behind that object. For example, suppose we need to search the current user's *My Documents* folder (and sub-folders) for all non-system files modified in the last hour. Using LINQ we do this as follows:

```
using SIO = System.IO;

string[] files;
string   mydocs;

mydocs = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
files  = SIO.Directory.GetFiles(mydocs, "*.*", SIO.SearchOption.AllDirectories);

var query = from file in files
            let lasthour = DateTime.Now.Subtract(new TimeSpan(0, 1, 0, 0))
            where SIO.File.GetLastAccessTime(file) >= lasthour &&
                (SIO.File.GetAttributes(file) & SIO.FileAttributes.System) == 0
            select file;
```

Notice the presence of the let statement, which allows for the definition of values local to the query; let is used to improve readability and efficiency by factoring out common operations.

This example should not be very surprising, since what we are really doing is iterating across an array of filenames ("*files*"), not the file system itself. But this is a design artifact of the .NET Framework, not a limitation of LINQ. A similar example is the searching of a file, which is easily done in LINQ by iterating across the lines of the file:

```
string filename = ...;  // file to search

var lines = from line in SIO.File.ReadAllLines(filename)
            where line.Contains("class")
            select line;
```

In this example, we are reading all the lines into an array and then searching the array to select those lines containing the character sequence "class."

Need to search the Windows event log? An event log is a collection of EventLogEntry objects, and is accessed in .NET by creating an instance of the EventLog class. For example, here's how we gain access to the *Application* event log on the current machine:

```
using SD = System.Diagnostics;

SD.EventLog applog = new SD.EventLog("Application", ".");
```

Suppose we need to find all events logged by our application for a particular user. This is easily expressed as a LINQ query:

```
string appname  = ...;   // name of our application, e.g. "SchedulingApp"
string username = ...;   // login name for user, e.g. "DOMAIN\\hummel"

var entries = from entry in applog.Entries
              where entry.Source == appname &&
                    entry.UserName == username
              orderby entry.TimeWritten descending
              select entry;
```

Interestingly, while this query makes perfect sense, it does *not* compile. The issue is that EnTRies is a pre-2.0 collection, which means it implements IEnumerable and not IEnumerable<T>. Since IEnumerable is defined in terms of object and not a specific type T, the C# type inference engine cannot infer the type of objects the query expression is working with. The designers of LINQ provide an easy solution in the form of the standard query operator Cast. The Cast operator wraps a generic enumerable object with a type-specific one suitable for LINQ:

```
var entries = from entry in applog.Entries.Cast<SD.EventLogEntry>()
```

```
       where entry.Source == appname &&
             entry.UserName == username
       orderby entry.TimeWritten descending
       select entry;
```

The Cast operator allows LINQ to support .NET 1.*x* collections.

As a final example, let's apply LINQ to the world of Visual Studio Tools for Office (VSTO). To search a user's Outlook contacts, the basic query is as follows:

```
Outlook.MAPIFolder folder = this.ActiveExplorer().Session.
  GetDefaultFolder( Outlook.OlDefaultFolders.olFolderContacts );

var contacts = from contact in folder.Items.OfType<Outlook.ContactItem>()
               where ...  // search criteria, e.g. contact.Email1Address != null
               select contact;
```

We take advantage of LINQ's OfType query operator, which (a) wraps pre-2.0 collections and (b) filters the collection to return only those objects of the desired type. Here's a query to collect all distinct email addresses from a user's Outlook contacts:

```
var emails = (
              from contact in folder.Items.OfType<Outlook.ContactItem>()
              where contact.Email1Address != null
              select contact.Email1Address
             )
             .Distinct();
```

Finally, given collections of email addresses from different folders or users, we can use LINQ to perform various set operations over these collections:

```
var union        = emails.Union(emails2);
var intersection = emails.Intersect(emails2);
var difference   = emails.Except(emails2);
```

PREV NEXT

PREV NEXT

Chapter 6. Standard LINQ Query Operators

LINQ provides a wide-range of query operators, many of which have been demonstrated in the previous sections. The purpose of this section is to succinctly summarize the complete set of LINQ query operators listed in Table 1. Recall that you must import the System.Query namespace to use these operators.

PREV NEXT

PREV NEXT

6.1.

6.1.1. Aggregate

The Aggregate operator applies a function over a sequence, with or without an initial seed value. The result can be post-processed by another function is desired.

```
int[] ints  = { 1, 2, 3, 4, 5, 6 };
var   query = from ...;

int sum     = ints.Aggregate( (a, b) => a + b );
int product = ints.Aggregate( 1, (a, b) => a * b );
int sump1   = ints.Aggregate( 0, (a, b) => a + b, r => r+1 );
var result  = query.Aggregate(...);

Console.WriteLine(sum);      // 21
Console.WriteLine(product);  // 720
Console.WriteLine(sump1);    // 22
Console.WriteLine(result);
```

The sequence can be of any type *T*.

See also: Average, Count, LongCount, Max, Min, and Sum.

Table 1. List of Standard LINQ Query Operators

Operator	Lazy?	Description
Aggregate	No	Applies a function to a sequence, yielding a single value.
All	No	Applies a function to a sequence to see if all elements satisfy the function.
Any	No	Applies a function to a sequence to see if any element satisfies the function.
Average	No	Computes the average of a numeric sequence.
Cast	Yes	Yields the elements of a sequence type-casted to a given type.
Concat	Yes	Yields the concatenation of two sequences S1 and S2.
Contains	No	Searches a sequence to see if it contains a given element.
Count	No	Counts the number of elements in a sequence, yielding an integer result.
DefaultIfEmpty	Yes	Given a sequence S, yields S or a sequence with the default value if S is empty.
Distinct	Yes	Returns a sequence with duplicates eliminated.
ElementAt	No	Returns the ith element of a sequence.
ElementAtOrDefault	No	Returns the ith element of a sequence, or the default value if sequence is empty.
Empty	Yes	Yields an empty sequence.
EqualAll	No	Compares two sequences for equality.
Except	Yes	Given two sequences S1 and S2, returns the set difference S1 S2.
First	No	Returns the first element of a sequence.
FirstOrDefault	No	Returns the first element of a sequence, or the default value if sequence is empty.
Fold	No	Obsolete, see Aggregate.
GroupBy	Yes	Groups the elements of a sequence by key.
GroupJoin	Yes	Performs a join of two sequences S1 and S2, yielding a hierarchical result.
Intersect	Yes	Given two sequences S1 and S2, returns the set intersection of S1 and S2.
Join	Yes	Performs a traditional inner equijoin of two sequences S1 and S2.
Last	No	Returns the last element of a sequence.
LastOrDefault	No	Returns the last element of a sequence, or the default value if sequence is empty.

LongCount	No	Counts the number of elements in a sequence, yielding a long result.
Max	No	Returns the maximum of a sequence.
Min	No	Returns the minimum of a sequence.
OfType	Yes	Yields the elements of a sequence that match a given type.
OrderBy	Yes	Orders a sequence of elements by key into ascending order.
OrderByDescending	Yes	Orders a sequence of elements by key into descending order.
Range	Yes	Yields a sequence of integers in a given range.
Repeat	Yes	Yields a sequence of values by repeating a given value n times.
Reverse	Yes	Reverses the elements of a sequence.
Select	Yes	Applies a projection function to a sequence, yielding a new sequence.
SelectMany	Yes	Applies a projection function to flatten a sequence of sequences.
Single	No	Returns the lone element of a singleton sequence.
SingleOrDefault	No	Returns the lone element of a singleton sequence, or default if sequence is empty.
Skip	Yes	Skips the first n elements of a sequence, yielding the remaining elements.
SkipWhile	Yes	Given function F and sequence S, skips the initial elements of S where F is true.
Sum	No	Computes the sum of a numeric sequence.
Take	Yes	Yields the first n elements of a sequence.
TakeWhile	Yes	Given function F and sequence S, yields the initial elements of S where F is true.
ThenBy	Yes	Takes an ordered sequence and yields a secondary, ascending ordering.
TheyByDescending	Yes	Takes an ordered sequence and yields a secondary, descending ordering.
ToArray	No	Iterates across a sequence, capturing results in an array.
ToDictionary	No	Iterates across a sequence, capturing results in a *Dictionary<K, V>*.
ToList	No	Iterates across a sequence, capturing results in a *List<T>*.
ToLookup	No	Iterates across a sequence, capturing results in a *Lookup<K, IEnumerable<V>>*.
ToSequence	Yes	Casts a sequence as an *IEnumerable* sequence for use with standard query ops.
Union	Yes	Given two sequences S1 and S2, returns the set union of S1 and S2.
Where	Yes	Applies a Boolean function to a sequence, yielding a sub-sequence.

6.1.2. All

The All operator applies a function over a sequence, checking to see if all of the elements satisfy the function, i.e., cause the function to return true. For example, do all the doctors have pager numbers? How about all the doctors retrieved by a particular query?

```
Doctors doctors = new Doctors();

var query = from doc in doctors ...;

bool allHavePagers   = doctors.All(doc => doc.PagerNumber > 0);
bool theseHavePagers = query.All(doc => doc.PagerNumber > 0);
```

See also: Any, Contains, and EqualAll.

6.1.3. Any

The Any operator applies a function over a sequence, checking to see if any of the elements satisfy the function, i.e., cause the function to return true. For example, are there any doctors living in Lake Forest?

```
Doctors doctors = new Doctors();

bool inLakeForest = doctors.Any(doc => doc.City == "Lake Forest");
```

The function is optional; if omitted, the Any operator returns true if the sequence contains at least one element.

```
var query = from doc in doctors
            where doc.City == "Lake Forest"
            select doc;

bool inLakeForest = query.Any();
```

See also: All, Contains, and EqualAll.

6.1.4. Average

The Average operator computes the average of a sequence of numeric values. The values are either the sequence itself, or selected out of a sequence of objects.

```
int[]      ints    = { 1, 2, 3, 4, 5, 6 };
decimal?[] values  = { 1, null, 2, null, 3, 4 };
Doctors    doctors = new Doctors();
var        query   = from ...;

double   avg1     = ints.Average();
decimal? avg2     = values.Average();
double   avgYears = doctors.Average( doc =>
                      DateTime.Now.Subtract(doc.StartDate).Days / 365.25 );
var      avg      = query.Average();

Console.WriteLine(avg1);   // 3.5
Console.WriteLine(avg2);   // 2.5
Console.WriteLine(avgYears.ToString("0.00"));   // 5.72
Console.WriteLine(avg);
```

The values can be of type int, int?, long, long?, decimal, decimal?, double, or double?. The resulting type is double, double?, double, double?, decimal, decimal?, double, or double?, respectively.

See also: Aggregate, Count, LongCount, Max, Min, and Sum.

6.1.5. Cast

The Cast operator yields the elements of a sequence type-casted to a given type *T*.

```
System.Collections.ArrayList al = new System.Collections.ArrayList();
al.Add("abc");
al.Add("def");
al.Add("ghi");

var strings = al.Cast<string>();

foreach(string s in strings)   // "abc", "def", "ghi"
  Console.WriteLine(s);
```

The Cast operator is commonly used to wrap pre-2.0 collections (such as ArrayLists) for use with LINQ. Here's a more interesting example of searching the Windows event log for all events logged by an application:

```
var entries = from entry in applog.Entries.Cast<System.Diagnostics.EventLogEntry>()
              where entry.Source == "ApplicationName"
              orderby entry.TimeWritten descending
              select entry;
```

See also: OfType.

6.1.6. Concat

The Concat operator concatenates two sequences S1 and S2, yielding the elements of S1 followed by the elements of S2.

```
int[] ints1  = { 1, 2, 3 };
int[] ints2  = { 4, 5, 6 };
var   query1 = from ...;
var   query2 = from ...;

var all     = ints1.Concat(ints2);
var results = query1.Concat(query2);

foreach(var x in all)     // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(var r in results)
  Console.WriteLine(r);
```

The sequences may contain elements of any type *T*. This element type T must be the same at compile-time (e.g. object), but may differ at run-time:

```
object[] objects1 = { "abc", "def" };
object[] objects2 = { 1, 2, 3 };

var result = objects1.Concat(objects2);
```

See also: Union.

6.1.7. Contains

The Contains operator searches a sequence to see if it contains a given element. For example, is there a doctor with initials "*gg*" still working at University Hospital? One approach is to create a Doctor object with the initials we are looking for, and see if the collection contains this object:

```
Doctors doctors = new Doctors();

bool docExists = doctors.Contains( new Doctor("gg", ...) );
```

This assumes the Doctor class defines Equals based on a doctor's initials. Another approach (without this assumption) is to select all the initials and then see if the result contains the string "gg":

```
var query = from doc in doctors
            select doc.Initials

bool docExists = query.Contains("gg");
```

See also: All, Any, EqualAll, and Where.

6.1.8. Count

The Count operator counts the number of elements in a sequence, yielding an integer result. The elements are either the sequence itself, or selected from a sequence of objects.

```
int[]      ints   = { 1, 2, 3, 4, 5, 6 };
decimal?[] values = { 1, null, 2, null, 3 };
IEnumerable<Doctor> doctors = new Doctors();
var        query  = from ...;

int count1 = ints.Count();
int count2 = values.Count();
int count3 = doctors.Count();
int count4 = doctors.Count( doc => doc.City == "Chicago" );
int count  = query.Count();
```

```
Console.WriteLine(count1);   // 6
Console.WriteLine(count2);   // 5
Console.WriteLine(count3);   // 12
Console.WriteLine(count4);   // 5
Console.WriteLine(count);
```

The sequence can be of any type T.

See also: Aggregate, Average, LongCount, Max, Min, and Sum.

6.1.9. DefaultIfEmpty

Given a non-empty sequence, the DefaultIfEmpty operator yields this same sequence. If the sequence is empty, DefaultIfEmpty yields a sequence containing a single default value.

```
int[]   ints1   = { 1, 2, 3, 4, 5, 6 };
int[]   ints2   = { };
var     query   = from ...;

var  ints   = ints1.DefaultIfEmpty();
var  zero   = ints2.DefaultIfEmpty();
var  minus1 = ints2.DefaultIfEmpty(-1);
var  result = query.DefaultIfEmpty();

foreach(int x in ints)     // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(int x in zero)     // 0
  Console.WriteLine(x);
foreach(int x in minus1)   // -1
  Console.WriteLine(x);
foreach(var r in result)
  Console.WriteLine(r);
```

The sequence can be of any type T; if the sequence is empty and a default value is not provided, the default value for type T is used.

See also: FirstOrDefault, GroupJoin, LastOrDefault, SingleOrDefault, and ElementAtOrDefault.

6.1.10. Distinct

Given a sequence of elements, the Distinct operator returns the same sequence without duplicates.

```
int[] ints  = { 1, 2, 2, 3, 2, 3, 4 };
var   query = from ...;

var distinctInts     = ints.Distinct();
var distinctResults = query.Distinct();

foreach(var x in distinctInts)  // 1, 2, 3, 4
  Console.WriteLine(x);
foreach(var r in distinctResults)
  Console.WriteLine(r);
```

The sequence can be of any type T.

See also: Except, Intersect, and Union.

6.1.11. ElementAt

The ElementAt operator returns the i^{th} element of a sequence; the sequence must be non-empty, and *i* is 0-based.

```
int[]   ints    = { 1, 2, 3, 4, 5, 6 };
Doctors doctors = new Doctors();
var     query   = from ...;

int    third  = ints.ElementAt(2);
Doctor doctor = doctors.ElementAt(2);
var    result = query.ElementAt(i);

Console.WriteLine(third);             // 3
Console.WriteLine(doctor.Initials);   // 3rd doc in Chicago: ch
Console.WriteLine(result);
```

The sequence can be of any type *T*.

See also: ElementAtOrDefault.

6.1.12. ElementAtOrDefault

The ElementAtOrDefault operator returns the *i*th element of a possibly empty sequence; *i* is 0-based.

```
int[]   ints1   = { 1, 2, 3, 4, 5, 6 };
int[]   ints2   = { };
Doctors doctors = new Doctors();
var     query   = from ...;

int    x1     = ints1.ElementAtOrDefault(2);
int    x2     = ints1.ElementAtOrDefault(6);
int    x3     = ints2.ElementAtOrDefault(0);
Doctor doc1   = doctors.ElementAtOrDefault(2);
Doctor doc2   = doctors.ElementAtOrDefault(-1);
var    result = query.ElementAtOrDefault(i);

Console.WriteLine(x1);             // 3
Console.WriteLine(x2);             // 0
Console.WriteLine(x3);             // 0
Console.WriteLine(doc1 == null);   // False
Console.WriteLine(doc2 == null);   // True
Console.WriteLine(result);
```

The sequence can be of any type T; if the sequence is empty or *i* is invalid, the default value for type T is returned.

See also: ElementAt.

6.1.13. Empty

The Empty operator yields an empty sequence of the given type.

```
var emptyDocs = System.Query.Sequence.Empty<Doctor>();

foreach(var doc in emptyDocs)   // <none>
  Console.WriteLine(doc);
```

This operator is helpful when you need an empty sequence for another operator or a method argument.

See also: Range and Repeat.

6.1.14. EqualAll

The EqualAll operator compares two sequences for equality. Two sequences are equal if they are of the same length, and contain the same sequence of elements.

```
var query1 = from ...;
var query2 = from ...;

bool equal = query1.EqualAll(query2);
```

The sequences may contain elements of any type *T*. This element type T must be the same at compile-time (e.g. object), but may differ at run-time:

```
object[] objects1 = { "abc", "def" };
object[] objects2 = { 1, 2, 3 };

bool isFalse = objects1.EqualAll(objects2);  // false
```

The result in such cases is false.

See also: All, Any, and Contains.

6.1.15. Except

Given two sequences of elements S1 and S2, the Except operator returns the distinct elements of S1 not in S2. In other words, Except computes the set difference S1 S2.

```
int[] intsS1 = { 1, 2, 2, 3, 2, 3, 4, 5, 6 };
int[] intsS2 = { 1, 3, 6, 7 };
var   query1 = from ...;
var   query2 = from ...;

var diffInts    = intsS1.Except(intsS2);
var diffResults = query1.Except(query2);

foreach(var x in diffInts)  // 2, 4, 5
  Console.WriteLine(x);
foreach(var r in diffResults)
  Console.WriteLine(r);
```

The sequences may contain elements of any type *T*. This element type T must be the same at compile-time (e.g. object), but may differ at run-time:

```
object[] objects1 = { "abc", "def" };
object[] objects2 = { 1, 2, 3 };

var result = objects1.Except(objects2);
```

See also: Distinct, Intersect, and Union.

6.1.16. First

The First operator returns the first element of a sequence; the sequence must be non-empty.

```
int[]   ints    = { 1, 2, 3, 4, 5, 6 };
Doctors doctors = new Doctors();
var     query   = from ...;

int    first  = ints.First();
Doctor doctor = doctors.First(doc => doc.City == "Chicago");
var    result = query.First();

Console.WriteLine(first);              // 1
Console.WriteLine(doctor.Initials);   // mbl
Console.WriteLine(result);
```

The sequence can be of any type T.

See also: FirstOrDefault.

6.1.17. FirstOrDefault

The FirstOrDefault operator returns the first element of a possibly empty sequence.

```
int[]   ints1   = { 1, 2, 3, 4, 5, 6 };
int[]   ints2   = { };
Doctors doctors = new Doctors();
var     query   = from ...;

int    x1      = ints1.FirstOrDefault();
int    x2      = ints2.FirstOrDefault();
Doctor doc1    = doctors.FirstOrDefault(doc => doc.City == "Chicago");
Doctor doc2    = doctors.FirstOrDefault(doc => doc.City == "Planet Mars");
var    result = query.FirstOrDefault();

Console.WriteLine(x1);             // 1
Console.WriteLine(x2);             // 0
Console.WriteLine(doc1 == null);   // False
Console.WriteLine(doc2 == null);   // True
Console.WriteLine(result);
```

The sequence can be of any type T; if the sequence is empty, the default value for type T is returned.

See also: First.

6.1.18. Fold

The Fold operator is considered obsolete, see Aggregate.

6.1.19. GroupBy

The GroupBy operator groups the elements of a sequence by key; the keys are yielded by a function applied to each element. Each resulting group is an *IEnumerable* sequence of elements S with a key K.

```
Doctors doctors = new Doctors();

var groups = doctors.GroupBy(doc => doc.City);

foreach(var group in groups)
{
  Console.WriteLine("{0}:", group.Key);   // Chicago, Evanston, ...
  foreach(var doc in group)               // {mbl,jl,...}, {ch,cm,...}, ...
    Console.WriteLine("  {0}", doc.Initials);
}
```

A second version allows you to project exactly what data to store in the group, such as only the doctor's initials:

```
Doctors doctors = new Doctors();

var groups2 = doctors.GroupBy(doc => doc.City, doc => doc.Initials);

foreach(var group in groups2)
{
  Console.WriteLine("{0}:", group.Key);   // Chicago, Evanston, ...
  foreach(var initials in group)          // {mbl,jl,...}, {ch,cm,...}, ...
    Console.WriteLine("  {0}", initials);
}
```

Additional versions allow you to provide a comparer of type *IEqualityComparer* for comparing keys.

Use of the *group by* clause in a query expression translates into application of the GroupBy operator. For example, the following statements are equivalent:

```
var groups = doctors.GroupBy(doc => doc.City);
var groups = from doc in doctors
             group doc by doc.City into g
             select g;

var groups2 = doctors.GroupBy(doc => doc.City, doc => doc.Initials);
var groups2 = from doc in doctors
              group doc.Initials by doc.City into g
              select g;
```

See also: OrderBy.

6.1.20. GroupJoin

The GroupJoin operator performs a join of two sequences S1 and S2, based on the keys selected from S1's and S2's elements. The keys are yielded by functions applied to each element; a third function determines the data projected by the join.

Unlike an inner join which yields essentially a table of joined records, the result of a GroupJoin is hierarchical. For each element in S1, there's a possibly empty sub-sequence of matching elements from S2. For example, the following query joins Doctors and Calls via the doctors' initials to determine which doctors are working on what dates:

```
DataSets.SchedulingDocs ds = FillDataSet();

var working = ds.Doctors.GroupJoin( ds.Calls,
                                    doc => doc.Initials,
                                    call => call.Initials,
                                    (doc,call) => new { doc.Initials, Calls=call }
                                  );

foreach(var record in working)
{
  Console.WriteLine("{0}:", record.Initials);
  foreach(var call in record.Calls)
    Console.WriteLine("  {0}", call.DateOfCall);
}
```

Here's the output:

```
ay:
  11/2/2006 12:00:00 AM
bb:
ch:
cm:
jl:
  10/2/2006 12:00:00 AM
  11/1/2006 12:00:00 AM
.
.
.
```

Use of the *join into* clause in a query expression translates into application of the GroupJoin operator. For example, the following statements are equivalent:

```
var working = ds.Doctors.GroupJoin( ds.Calls,
                                    doc => doc.Initials,
                                    call => call.Initials,
                                    (doc,call) => new { doc.Initials, Calls=call }
                                  );

var working = from doc in ds.Doctors
              join call in ds.Calls
              on doc.Initials equals call.Initials
              into j
              select new { doc.Initials, Calls = j };
```

To perform a traditional left outer join (and thus flatten the result into a table), use the DefaultIfEmpty operator as follows:

```
var working = from doc in ds.Doctors
              join call in ds.Calls
              on doc.Initials equals call.Initials
              into j
              from r in j.DefaultIfEmpty()
              select new { doc.Initials, Call = r };

foreach(var record in working)
  Console.WriteLine("{0}: {1}",
    record.Initials,
    (record.Call == null) ? "" : record.Call.DateOfCall.ToString());
```

See also: Join.

6.1.21. Intersect

Given two sequences of elements S1 and S2, the Intersect operator returns the distinct elements of S1 that also appear in S2. In other words, Intersect computes the set intersection of S1 and S2.

```
int[] intsS1 = { 1, 2, 2, 3, 2, 3, 4, 5, 6, 8 };
int[] intsS2 = { 6, 1, 3, 6, 7 };
var   query1 = from ...;
var   query2 = from ...;

var commonInts    = intsS1.Intersect(intsS2);
var commonResults = query1.Intersect(query2);

foreach(var x in commonInts)  // 1, 3, 6
  Console.WriteLine(x);
foreach(var r in commonResults)
  Console.WriteLine(r);
```

The sequences may contain elements of any type *T*. This element type T must be the same at compile-time (e.g. object), but may differ at run-time:

```
object[] objects1 = { "abc", "def" };
object[] objects2 = { 1, 2, 3 };

var result = objects1.Intersect(objects2);
```

See also: Distinct, Except, and Union.

6.1.22. Join

The Join operator performs an inner equijoin of two sequences S1 and S2, based on the keys selected from S1's and S2's elements. The keys are yielded by functions applied to each element; a third function determines the data projected by the join. For example, the following query joins Doctors and Calls via the doctors' initials to determine which doctors are working on what dates:

```
DataSets.SchedulingDocs ds = FillDataSet();

var working = ds.Doctors.Join( ds.Calls,
                               doc => doc.Initials,
                               call => call.Initials,
                               (doc,call) => new { doc.Initials, call.DateOfCall }
                              );

foreach(var record in working)
  Console.WriteLine(record);
```

The query yields pairs of the form (Initials, DateOfCall), created from the joined Doctor and Call elements. Here's the output:

```
{Initials=ay, DateOfCall=11/2/2006 12:00:00 AM}
{Initials=jl, DateOfCall=10/2/2006 12:00:00 AM}
{Initials=jl, DateOfCall=11/1/2006 12:00:00 AM}
.
.
.
```

Use of the *join* clause in a query expression translates into application of the Join operator. For example, the following statements are equivalent:

```
var working = ds.Doctors.Join( ds.Calls,
                               doc => doc.Initials,
                               call => call.Initials,
                               (doc,call) => new { doc.Initials, call.DateOfCall }
                              );

var working = from doc in ds.Doctors
              join call in ds.Calls
              on doc.Initials equals call.Initials
              select new { doc.Initials, call.DateOfCall };
```

See also: GroupJoin.

6.1.23. Last

The Last operator returns the last element of a sequence; the sequence must be non-empty.

```
int[]   ints    = { 1, 2, 3, 4, 5, 6 };
Doctors doctors = new Doctors();
var     query   = from ...;

int    last   = ints.Last();
Doctor doctor = doctors.Last(doc => doc.City == "Chicago");
var    result = query.Last();

Console.WriteLine(last);              // 6
Console.WriteLine(doctor.Initials);   // tm
Console.WriteLine(result);
```

The sequence can be of any type T.

See also: LastOrDefault.

6.1.24. LastOrDefault

The LastOrDefault operator returns the last element of a possibly empty sequence.

```
int[]   ints1   = { 1, 2, 3, 4, 5, 6 };
int[]   ints2   = { };
Doctors doctors = new Doctors();
var     query   = from ...;

int    x1     = ints1.LastOrDefault();
int    x2     = ints2.LastOrDefault();
Doctor doc1   = doctors.LastOrDefault(doc => doc.City == "Chicago");
Doctor doc2   = doctors.LastOrDefault(doc => doc.City == "Planet Mars");
var    result = query.LastOrDefault();

Console.WriteLine(x1);             // 6
Console.WriteLine(x2);             // 0
Console.WriteLine(doc1 == null);   // False
Console.WriteLine(doc2 == null);   // True
Console.WriteLine(result);
```

The sequence can be of any type T; if the sequence is empty, the default value for type T is returned.

See also: Last.

6.1.25. LongCount

The LongCount operator counts the number of elements in a sequence, yielding a long result. The elements are either the sequence itself, or selected from a sequence of objects.

```
int[]      ints   = { 1, 2, 3, 4, 5, 6 };
decimal?[] values = { 1, null, 2, null, 3 };
IEnumerable<Doctor> doctors = new Doctors();
var        query  = from ...;

long count1 = ints.LongCount();
long count2 = values.LongCount();
long count3 = doctors.LongCount();
long count4 = doctors.LongCount( doc => doc.City == "Chicago" );
long count  = query.LongCount();

Console.WriteLine(count1);   // 6
Console.WriteLine(count2);   // 5
Console.WriteLine(count3);   // 12
Console.WriteLine(count4);   // 5
Console.WriteLine(count);
```

The sequence can be of any type T.

See also: Aggregate, Average, Count, Max, Min, and Sum.

6.1.26. Max

The Max operator finds the maximum of a sequence of values. The values are either the sequence itself, or selected out of a sequence of objects, and must implement *IComparable* for comparison purposes.

```
int[]      ints    = { 1, 2, 3, 6, 5, 4 };
decimal?[] values  = { 1, null, 4, null, 3, 2 };
Doctors    doctors = new Doctors();
var        query   = from ...;

int      max1        = ints.Max();
decimal? max2        = values.Max();
string   maxInitials = doctors.Max( doc => doc.Initials );
double   maxYears    = doctors.Max( doc =>
                       DateTime.Now.Subtract(doc.StartDate).Days / 365.25 );
var      max         = query.Max();

Console.WriteLine(max1);          // 6
Console.WriteLine(max2);          // 4
Console.WriteLine(maxInitials);   // "vj"
Console.WriteLine(maxYears.ToString("0.00"));   // 11.43
Console.WriteLine(max);
```

The sequence can be of any type T; the resulting type is the same.

See also: Aggregate, Average, Count, LongCount, Min, and Sum.

6.1.27. Min

The Min operator finds the minimum of a sequence of values. The values are either the sequence itself, or selected out of a sequence of objects, and must implement *IComparable* for comparison purposes.

```
int[]      ints    = { 6, 2, 3, 1, 5, 4 };
decimal?[] values  = { 4, null, 1, null, 3, 2 };
Doctors    doctors = new Doctors();
var        query   = from ...;

int      min1       = ints.Min();
decimal? min2       = values.Min();
```

```
string   minInitials = doctors.Min( doc => doc.Initials );
double   minYears    = doctors.Min( doc =>
                         DateTime.Now.Subtract(doc.StartDate).Days / 365.25 );
var      min         = query.Min();

Console.WriteLine(min1);          // 1
Console.WriteLine(min2);          // 1
Console.WriteLine(minInitials);   // "ay"
Console.WriteLine(minYears.ToString("0.00"));   // 0.67
Console.WriteLine(min);
```

The sequence can be of any type T; the resulting type is the same.

See also: Aggregate, Average, Count, LongCount, Max, and Sum.

6.1.28. OfType

The OfType operator yields the elements of a sequence that match a given type T. In other words, OfType filters a sequence by type.

```
System.Collections.ArrayList  al = new System.Collections.ArrayList();
al.Add(1);
al.Add("abc");
al.Add(2);
al.Add("def");
al.Add(3);

var strings = al.OfType<string>();

foreach(string s in strings)   // "abc", "def"
      Console.WriteLine(s);
```

The OfType operator is commonly used to (a) wrap pre-2.0 collections (such as ArrayLists) for use with LINQ and (b) filter mixed collections. Here's a more interesting example of searching a user's Outlook contacts with LINQ:

```
Outlook.MAPIFolder folder = this.ActiveExplorer().Session.
  GetDefaultFolder( Outlook.OlDefaultFolders.olFolderContacts );

var contacts = from contact in folder.Items.OfType<Outlook.ContactItem>()
               where .
                     .  // search criteria, e.g. contact.Email1Address != null
                     .
               select contact;
```

See also: Cast.

6.1.29. OrderBy

The OrderBy operator orders a sequence of elements into ascending order; the keys used to order the sequence are yielded by a function applied to each element.

```
int[] ints      = {3, 1, 6, 4, 2, 5};
Doctors doctors = new Doctors();

var sorted = ints.OrderBy(x => x);
var docs   = doctors.OrderBy(doc => doc.Initials);

foreach(var x in sorted)   // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(var doc in docs)   // ay, bb, ..., vj
  Console.WriteLine(doc.Initials);
```

A second version allows you to provide a comparer of type *IComparer* for comparing keys and thus controlling the ordering, e.g., when you want to compare strings in a case-insensitive manner:

```
var docs = doctors.OrderBy(doc => doc.Initials,
                           StringComparer.CurrentCultureIgnoreCase);
```

Use of the *orderby* clause in a query expression translates into application of the OrderBy operator. For example, the following statements are equivalent:

```
var docs = doctors.OrderBy(doc => doc.Initials);

var docs = from doc in doctors
           orderby doc.Initials
           select doc;
```

See also: OrderByDescending, ThenBy, and ThenByDescending.

6.1.30. OrderByDescending

The OrderByDescending operator orders a sequence of elements into descending order; the keys used to order the sequence are yielded by a function applied to each element.

```
int[] ints      = {3, 1, 6, 4, 2, 5};
Doctors doctors = new Doctors();

var sorted = ints.OrderByDescending(x => x);
var docs   = doctors.OrderByDescending(doc => doc.Initials);

foreach(var x in sorted)   // 6, 5, 4, 3, 2, 1
  Console.WriteLine(x);
foreach(var doc in docs)   // vj, tm, ..., ay
  Console.WriteLine(doc.Initials);
```

A second version allows you to provide a comparer of type *IComparer* for comparing keys and thus controlling the ordering, e.g., when you want to compare strings in a case-insensitive manner:

```
var docs = doctors.OrderByDescending(doc => doc.Initials,
                                     StringComparer.CurrentCultureIgnoreCase);
```

Use of the *orderby* clause in a query expression with the *descending* keyword translates into application of the OrderByDescending operator. For example, the following statements are equivalent:

```
var docs = doctors.OrderByDescending(doc => doc.Initials);

var docs = from doc in doctors
           orderby doc.Initials descending
           select doc;
```

See also: OrderBy, ThenBy, and ThenByDescending.

6.1.31. Range

The Range operator yields a sequence of integers in the given range, inclusive.

```
var oneToTen = System.Query.Sequence.Range(1, 10);

foreach(var x in oneToTen)  // 1, 2, 3, ..., 10
  Console.WriteLine(x);
```

See also: Empty and Repeat.

6.1.32. Repeat

The Repeat operator yields a sequence of values by repeating a given value *n* times.

```
var zeros   = System.Query.Sequence.Repeat(0, 8);
var strings = System.Query.Sequence.Repeat("", n);

Console.WriteLine(zeros.Count());     // 8
Console.WriteLine(strings.Count());   // n

foreach(var zero in zeros)   // 0, 0, ..., 0
  Console.WriteLine(zero);
```

See also: Empty and Range.

6.1.33. Reverse

The Reverse operator reverses the elements of a sequence.

```
int[] ints      = {1, 2, 3, 4, 5, 6};
Doctors doctors = new Doctors();

var revInts = ints.Reverse();
var docs    = from doc in doctors
              orderby doc.Initials
              select doc;
var revDocs = docs.Reverse();

foreach(var x in revInts)     // 6, 5, 4, 3, 2, 1
  Console.WriteLine(x);
foreach(var doc in revDocs)   // vj, tm, ..., ay
  Console.WriteLine(doc.Initials);
```

See also: OrderBy and OrderByDescending.

6.1.34. Select

The Select operator applies a projection function over a sequence of elements, yielding a sequence of possibly new elements.

```
int[] ints      = {1, 2, 3, 4, 5, 6};
Doctors doctors = new Doctors();

var sameInts = ints.Select(x => x);
var chicago  = doctors.Where(d => d.City == "Chicago").Select(d => d.Initials);
var names    = doctors.Select(d => new {LN=d.FamilyLastName, FN=d.GivenFirstName});

foreach(var x in sameInts)   // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(var d in chicago)    // mbl, jl, kl, py, tm
  Console.WriteLine(d);
foreach(var n in names)      // {LN=Larsen, FN=Marybeth}, ...
  Console.WriteLine(n);
```

Use of the *select* clause in a query expression translates into application of the Select operator (except in trivial cases where the Select operator can be optimized away). For example, the following statements are equivalent:

```
var chicago = doctors.Where(d => d.City == "Chicago").Select(d => d.Initials);

var chicago = from d in doctors
              where d.City == "Chicago"
              select d.Initials;
```

A second version of the Select operator provides an element's 0-based index in the sequence along with the element itself, for example:

```
var sumLast3 = ints.Select( (x,index) => (index >= ints.Length-3) ? x : 0 ).Sum();

Console.WriteLine(sumLast3);   // 15
```

See also: SelectMany and Where.

6.1.35. SelectMany

The SelectMany operator applies a projection function over a sequence of sequences, yielding a "flattened" sequence of possibly new elements. For example, turning an array of arrays into an array:

```
List<int[]> list = new List<int[]>();
int[] ints123 = {1, 2, 3};
int[] ints456 = {4, 5, 6};

list.Add(ints123);
list.Add(ints456);

var flat = list.SelectMany(x => x);

foreach (var x in list)   // Int32[], Int32[]
  Console.WriteLine(x);
foreach (var x in flat)   // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
```

The SelectMany operator is commonly used to flatten hierarchical data. For example, the following query yields the list of doctors working in October 2006, with sub-lists of the calls each is working:

```
DataSets.SchedulingDocs ds = FillDataSet();

var oct2006 = from d in ds.Doctors
               join c in ds.Calls
               on d.Initials equals c.Initials
               where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                     c.DateOfCall <= new DateTime(2006, 10, 31)
               group c by d.Initials into g
               select g;

foreach (var group in oct2006)  // jl: 1, mbl: 2
{
  Console.WriteLine("{0}: {1}", group.Key, group.Count());
  foreach (var call in group)
    Console.WriteLine(call.DateOfCall);
}
```

To flatten the hierarchy into just the list of calls:

```
var calls = oct2006.SelectMany(call => call);

foreach (var c in calls)
  Console.WriteLine(c.DateOfCall);
```

Use of the *select* clause in a query expression on all but the initial *from* clause translates into application of the SelectMany operator. For example, here's another way to flatten the list of calls for doctors working in October 2006:

```
var calls = from d in ds.Doctors
              from c in ds.Calls
              where
                d.Initials == c.Initials &&
                c.DateOfCall >= new DateTime(2006, 10, 1) &&
                c.DateOfCall <= new DateTime(2006, 10, 31)
              select c;
```

This is translated into the following application of the SelectMany operator:

```
var calls = ds.Doctors.SelectMany(
              d => ds.Calls.Where(c => d.Initials == c.Initials &&
                                   c.DateOfCall >= new DateTime(2006, 10, 1) &&
                                   c.DateOfCall <= new DateTime(2006, 10, 31)));
```

A second version of the SelectMany operator provides an element's 0-based index in the outer sequence along with the element itself.

See also: Select and Where.

6.1.36. Single

The Single operator returns the lone element of a sequence; the sequence must contain exactly one element.

```
int[]   ints    = { 3 };
Doctors doctors = new Doctors();
var     query   = from ...;

int    lone   = ints.Single();
Doctor doctor = doctors.Single(doc => doc.Initials == "mbl");
var    result = query.Single();

Console.WriteLine(lone);                   // 3
Console.WriteLine(doctor.PagerNumber);   // 52248
Console.WriteLine(result);
```

The sequence can be of any type T.

See also: SingleOrDefault.

6.1.37. SingleOrDefault

The SingleOrDefault operator returns the lone element of a sequence; the sequence must be empty or contain exactly one element.

```
int[]   ints1   = { 3 };
int[]   ints2   = { };
Doctors doctors = new Doctors();
var     query   = from ...;

int    x1     = ints1.SingleOrDefault();
int    x2     = ints2.SingleOrDefault();
Doctor doc1   = doctors.SingleOrDefault(doc => doc.Initials == "mbl");
Doctor doc2   = doctors.SingleOrDefault(doc => doc.Initials == "AAA");
var    result = query.SingleOrDefault();

Console.WriteLine(x1);              // 3
Console.WriteLine(x2);              // 0
Console.WriteLine(doc1 == null);   // False
Console.WriteLine(doc2 == null);   // True
Console.WriteLine(result);
```

The sequence can be of any type T; if the sequence is empty, the default value for type T is returned.

See also: Single.

6.1.38. Skip

The Skip operator skips the first *n* elements of a sequence, yielding the remaining elements. If *n* <= 0 the result is the sequence itself; if *n* >= sequence's length the result is an empty sequence.

```
int[] ints  = { 1, 2, 3, 4, 5, 6 };
var   query = from ...;

var last3    = ints.Skip(3);
var bottom10 = query.Skip( query.Count() - 10 );

foreach(var x in last3)  // 4, 5, 6
  Console.WriteLine(x);
foreach(var r in bottom10)
  Console.WriteLine(r);
```

The sequence can be of any type T.

See also: Take.

6.1.39. SkipWhile

Given a function F and a sequence S, the SkipWhile operator skips the first *n* elements of S where F returns true, yielding the remaining elements. Two versions of F are supported: accepting an element, or accepting an element along with its 0-based index in the sequence.

```
int[] ints  = { 1, 2, 3, 4, 5, 6 };
var   query = from ...;

var all   = ints.SkipWhile(x => false);
var none  = ints.SkipWhile(x => true);
var last3 = ints.SkipWhile( (x,i) => i < 3 );
var lastN = query.SkipWhile(result => ...);

foreach(var x in all)     // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(var x in none)    // <none>
  Console.WriteLine(x);
foreach(var x in last3)   // 4, 5, 6
  Console.WriteLine(x);
foreach(var r in lastN)
  Console.WriteLine(r);
```

The sequence can be of any type T.

See also: TakeWhile and Where.

6.1.40. Sum

The Sum operator computes the sum of a sequence of numeric values. The values are either the sequence itself or selected out of a sequence of objects.

```
int[]      ints    = { 1, 2, 3, 4, 5, 6 };
decimal?[] values  = { 1, null, 2, null, 3, 4 };
Doctors    doctors = new Doctors();
var        query   = from ...;

int      sum1     = ints.Sum();
decimal? sum2     = values.Sum();
double   sumYears = doctors.Sum( doc =>
                      DateTime.Now.Subtract(doc.StartDate).Days / 365.25 );
var      sum      = query.Sum();

Console.WriteLine(sum1);   // 21
Console.WriteLine(sum2);   // 10
Console.WriteLine(sumYears.ToString("0.00"));   // 68.61
Console.WriteLine(sum);
```

The values can be of type int, int?, long, long?, decimal, decimal?, double, or double?. The resulting type is the same.

See also: Aggregate, Average, Count, LongCount, Max, and Min.

6.1.41. Take

The Take operator yields the first *n* elements of a sequence. If *n* <= 0 the result is an empty sequence; if *n* >= sequence's length the result is the sequence itself.

```
int[] ints  = { 1, 2, 3, 4, 5, 6 };
var   query = from ...;

var first3 = ints.Take(3);
var top10  = query.Take(10);

foreach(var x in first3)  // 1, 2, 3
  Console.WriteLine(x);
foreach(var r in top10)
  Console.WriteLine(r);
```

The sequence can be of any type T.

See also: Skip.

6.1.42. TakeWhile

Given a function F and a sequence S, the TakeWhile operator yields the first n elements of S where F returns true. Two versions of F are supported: accepting an element, or accepting an element along with its 0-based index in the sequence.

```
int[] ints  = { 1, 2, 3, 4, 5, 6 };
var   query = from ...;

var all    = ints.TakeWhile(x => true);
var none   = ints.TakeWhile(x => false);
var first3 = ints.TakeWhile( (x,i) => i < 3 );
var firstN = query.TakeWhile(result => ...);

foreach(var x in all)      // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(var x in none)     // <none>
  Console.WriteLine(x);
foreach(var x in first3)   // 1, 2, 3
  Console.WriteLine(x);
foreach(var r in firstN)
  Console.WriteLine(r);
```

The sequence can be of any type T.

See also: SkipWhile and Where.

6.1.43. ThenBy

The ThenBy operator takes an ordered sequence and yields a secondary, ascending ordering; the keys used for the secondary ordering are yielded by a function applied to each element. Ordered sequences are typically produced by OrderBy or OrderByDescending, but can also be produced by ThenBy and ThenByDescending to yield additional sub-orderings.

```
Doctors doctors = new Doctors();

var docs = doctors.OrderBy(doc => doc.City).ThenBy(doc => doc.Initials);

foreach(var doc in docs)
  Console.WriteLine("{0}: {1}", doc.City, doc.Initials);
```

A second version allows you to provide a comparer of type *IComparer* for comparing keys and thus controlling the ordering, e.g., when you want to compare strings in a case-insensitive manner:

```
var docs = doctors.OrderBy(doc => doc.City)
                  .ThenBy(doc => doc.Initials,
                          StringComparer.CurrentCultureIgnoreCase);
```

Use of an *orderby* clause in a query expression with subsequent keys translates into application of the ThenBy operator. For example, the following are equivalent:

```
var docs = doctors.OrderBy(doc => doc.City).ThenBy(doc => doc.Initials);

var docs = from doc in doctors
           orderby doc.City, doc.Initials
           select doc;
```

See also: OrderBy, OrderByDescending, and ThenByDescending.

6.1.44. ThenByDescending

The ThenByDescending operator takes an ordered sequence and yields a secondary, descending ordering; the keys used for the secondary ordering are yielded by a function applied to each element. Ordered sequences are typically produced by OrderBy or OrderByDescending, but can also be produced by ThenBy and ThenByDescending to yield additional sub-orderings.

```
Doctors doctors = new Doctors();

var docs = doctors.OrderBy(doc => doc.City).ThenByDescending(doc => doc.Initials);

foreach(var doc in docs)
  Console.WriteLine("{0}: {1}", doc.City, doc.Initials);
```

A second version allows you to provide a comparer of type *IComparer* for comparing keys and thus controlling the ordering, e.g., when you want to compare strings in a case-insensitive manner:

```
var docs = doctors.OrderBy(doc => doc.City)
                  .ThenByDescending(doc => doc.Initials,
                                    StringComparer.CurrentCultureIgnoreCase);
```

Use of an *orderby* clause in a query expression with subsequent keys and the *descending* keyword translates into application of the ThenBy operator. For example, the following are equivalent:

```
var docs = doctors.OrderBy(doc => doc.City).ThenByDescending(doc => doc.Initials);

var docs = from doc in doctors
           orderby doc.City, doc.Initials descending
           select doc;
```

See also: OrderBy, OrderByDescending, and ThenBy.

6.1.45. ToArray

The ToArray operator iterates across a sequence of values, yielding an array containing these values. For example, doctors living in Chicago:

```
Doctors doctors = new Doctors();

var query = from doc in doctors
            where doc.City == "Chicago"
            select doc;

Doctor[] chicago = query.ToArray();
```

Since queries are lazily evaluated, ToArray is commonly used to execute a query and capture the results in a simple data structure.

See also: ToDictionary, ToList, ToLookup, and ToSequence.

6.1.46. ToDictionary

The ToDictionary operator iterates across a sequence of values, yielding a *Dictionary<K, V>* of (key, value) pairs. Each key must be unique, resulting in a one-to-one mapping of key to value. For example, storing doctors by their initials:

```
Doctors doctors = new Doctors();

var query = from doc in doctors
            where doc.City == "Chicago"
            select doc;

Dictionary<string, Doctor> chicago = query.ToDictionary(doc => doc.Initials);

foreach(var pair in chicago)
  Console.WriteLine("{0}: {1}", pair.Key, pair.Value.PagerNumber);
```

Since queries are lazily evaluated, ToDictionary is commonly used to execute a query and capture the results in a data structure that supports efficient lookup by key. For example, finding a doctor via their initials:

```
Doctor mbl = chicago["mbl"];
```

Another version provides control over exactly what values are stored in the data structure, e.g., only the doctor's email address:

```
Dictionary<string, string> emails = doctors.ToDictionary(doc => doc.Initials,
                                                          doc => doc.EmailAddress);
```

Finally, additional versions of ToDictionary allow you to provide a comparer of type *IEqualityComparer* for comparing keys.

See also: ToArray, ToList, ToLookup, and ToSequence.

6.1.47. ToList

The ToList operator iterates across a sequence of values, yielding a *List<T>* containing these values. For example, doctors living in Chicago:

```
Doctors doctors = new Doctors();

var query = from doc in doctors
            where doc.City == "Chicago"
            select doc;

List<Doctor> chicago = query.ToList();
```

Since queries are lazily evaluated, ToList is commonly used to execute a query and capture the results in a flexible data structure.

See also: ToArray, ToDictionary, ToLookup, and ToSequence.

6.1.48. ToLookup

The ToLookup operator iterates across a sequence of values, yielding a *Lookup<K, V>* of (key, value) pairs. The keys do not need to be unique; values with the same key form a collection under that key, resulting in a one-to-many mapping of key to values. For example, grouping doctors by city:

```
Doctors doctors = new Doctors();

var query = from doc in doctors
            select doc;
```

```
Lookup<string, Doctor> cities = query.ToLookup(doc => doc.City);

foreach(var pair in cities)
{
  Console.WriteLine("{0}:", pair.Key);  // city
  foreach(var doc in pair)              // doctors in that city
    Console.WriteLine("  {0}", doc.Initials);
}
```

Since queries are lazily evaluated, `ToLookup` is commonly used to execute a query and capture the results in a data structure that supports efficient lookup by key. For example, finding the doctors living in Chicago:

```
IEnumerable<Doctor> docs = cities["Chicago"];
```

Another version provides control over exactly what values are stored in the data structure, e.g., only the doctor's initials:

```
Lookup<string, string> cities = doctors.ToLookup(doc => doc.City,
                                                  doc => doc.Initials);
```

Finally, additional versions of `ToLookup` allow you to provide a comparer of type *IEqualityComparer* for comparing keys.

See also: ToArray, ToDictionary, ToList, and ToSequence.

6.1.49. ToSequence

The `ToSequence` operator casts a sequence as a sequence, thereby hiding any public members of the original sequencein particular those that might conflict or compete with the standard query operators. Use this operator when you want to gain access to the standard LINQ query operators.

For example, the following use of the *Count* query operator fails to compile:

```
Doctors doctors = new Doctors();
int count = doctors.Count(doc => doc.City == "Chicago");  // ERROR!
```

It fails because the Doctors class provides a conflicting `Count` property (inherited from *List<T>*). The `ToSequence` operator provides a quick solution:

```
int count1 = doctors.Count;
int count2 = doctors.ToSequence().Count();
int count3 = doctors.ToSequence().Count( doc => doc.City == "Chicago" );

Console.WriteLine(count1);   // 12
Console.WriteLine(count2);   // 12
Console.WriteLine(count3);   // 5
```

See also: ToArray, ToDictionary, ToList, and ToLookup.

6.1.50. Union

Given two sequences of elements S1 and S2, the `Union` operator returns the distinct elements of S1, followed by the distinct elements of S2 not in S1. In other words, `Union` computes the set union of S1 and S2.

```
int[] intsS1 = { 1, 2, 2, 3, 2, 3, 4, 6 };
int[] intsS2 = { 6, 1, 3, 5 };
var   query1 = from ...;
var   query2 = from ...;

var allInts    = intsS1.Union(intsS2);
var allResults = query1.Union(query2);

foreach(var x in allInts)  // 1, 2, 3, 4, 6, 5
```

```
  Console.WriteLine(x);
foreach(var r in allResults)
  Console.WriteLine(r);
```

The sequences may contain elements of any type *T*. This element type T must be the same at compile-time (e.g. `object`), but may differ at run-time:

```
object[] objects1 = { "abc", "def" };
object[] objects2 = { 1, 2, 3 };

var result = objects1.Union(objects2);
```

See also: Distinct, Except, and Intersect.

6.1.51. Where

The `Where` operator applies a function to a sequence of elements, yielding a sub-sequence of these elements.

```
int[] ints = {1, 2, 3, 4, 5, 6};
Doctors doctors = new Doctors();

var even    = ints.Where( x => x % 2 == 0);
var chicago = doctors.Where( doc => doc.City == "Chicago" );

foreach(var x in even)      // 2, 4, 6
  Console.WriteLine(x);
foreach(var doc in chicago)   // mbl, jl, kl, py, tm
  Console.WriteLine(doc.Initials);
```

Use of the *where* clause in a query expression translates into application of the `Where` operator. For example, the following statements are equivalent:

```
var chicago = doctors.Where( doc => doc.City == "Chicago" );

var chicago = from doc in doctors
              where doc.City == "Chicago"
              select doc;
```

A second version of the `Where` operator provides an element's 0-based index in the sequence along with the element itself, for example:

```
var last3 = ints.Where( (x,index) => (index >= ints.Length-3) ? true : false );

foreach (var x in last3)   // 4, 5, 6
  Console.WriteLine(x);
```

See also: Select.

NOTE

The application of LINQ is limited only by your imagination (and time). Here are a couple of the more interesting LINQ extensions floating around on the Web:

1. **BLinq** is a tool that generates an ASP.NET web application from a database schema. The web site supports the display, creation, and manipulation of the data in the database. LINQ to SQL is used for all database access, making the generated code easier to understand and modify. For more information see http://www.asp.net/sandbox/app_blinq.aspx?tabid=62.
2. **LINQ to Amazon** extends LINQ to support the querying of Amazon.com. See http://weblogs.asp.net/fmarguerie/archive/2006/06/26/Introducing-Linq-to-Amazon.aspx (http://weblogs.asp.net/fmarguerie/archive/2006/06/26/Introducing-Linq-to-Amazon.aspx:).

Chapter 7. Extending LINQ

In closing, let's touch on the extensibility of LINQ, another of its elegant design characteristics. As you have seen, LINQ interoperates with a range of data sources, from object graphs to relational databases to XML documents. Let's look at a quick example of how we go about extending one of our own classes to natively support LINQ.

Our running demo has focused on doctors working at the fictitious University Hospital. You've seen many queries involving doctors, often using the Distinct operator to eliminate duplicates. For example, a query to yield the distinct doctors working in October 2006:

```
DataSets.SchedulingDocs ds = FillDataSet();

var oct2006 = (
                from doc in ds.Doctors
                join call in ds.Calls
                on doc.Initials equals call.Initials
                where call.DateOfCall >= new DateTime(2006, 10, 1) &&
                      call.DateOfCall <= new DateTime(2006, 10, 31)
                select doc
              )
              .Distinct();
```

Since we designed the data model, we know that a doctor's initials are unique. We can take advantage of this information to optimize the application of Distinct in the query above by providing our own implementation specific to our DataSet:

```
using DS = DataSets.SchedulingDocs;

public static class DataSetsExtension
{

  public static IEnumerable<DS.DoctorsRow> Distinct(this
                                                    IEnumerable<DS.DoctorsRow> src)
  {
    // optimize by using a hashtable with doctors' initials:
    System.Collections.Hashtable ht = new System.Collections.Hashtable();

    foreach(var doc in src)  // for each doctor in the original sequence:
    {
      if (ht[doc.Initials] == null)  // first time, so remember and return:
      {
        ht.Add(doc.Initials, doc.Initials);
        yield return doc;
      }
      else  // we've yielded this doc already, so skip:
        continue;
    }
  }

}//class
```

For sequences of type DoctorsRowi.e., doctors in our DataSetthis implementation will be used in place of the standard LINQ implementation of Distinct. Thus, for any query of the form,

```
var query = from doc in ds.Doctors
            .
            .
            select doc;
```

the application of Distinct will invoke our optimized implementation. That's it!

Chapter 8. For Further Information

For all things LINQ, start at http://msdn.microsoft.com/data/ref/linq/. This short video with Anders Hejlsberg and Luca Bolognese provides a great overview of LINQ: http://channel9.msdn.com/showpost.aspx?postid=114680. Finally, keep an eye out for LINQ-related webcasts by yours truly, which will be announced at the O'Reilly site for this Short Cut (http://www.oreilly.com/catalog/language1) and on my blog (http://www.pluralsight.com/blogs/drjoe/).

Chapter 9. Acknowledgements

I'd like to thank the LINQ team at Microsoft for their insightful review of this Short Cut. The result is a much improved document, which benefits us all.

Chapter 9. Acknowledgements

I'd like to thank the LINQ team at Microsoft for their insightful review of this Short Cut. The result is a much improved document, which benefits us all.

4.3. Anonymous Types and Object Initializers

Consider the previous query to find the names of all doctors living in Chicago:

```
var query = from d in doctors
            where d.City == "Chicago"
            select new { d.GivenFirstName, d.FamilyLastName };
```

The query interacts with complete Doctor objects from the doctors collection, but then projects only the doctor's first and last name. Look carefully at the syntax. We see the new keyword without a class name, and { } instead of ( ):

```
new { d.GivenFirstName, d.FamilyLastName }
```

The new operator is in fact instantiating an object, but the class is an *anonymous type*the C# 3.0 compiler is automatically creating an appropriate class definition with a unique name and substituting that name. The only constructor provided by the generated class is a default one (i.e. parameter-less), which favors serialization but complicates the issue of object initialization. This is the motivation for the { }, which denote C# 3.0's new *object initializer* syntax. Object initializers provide an inline mechanism for initializing objects. And in the case of anonymous types, object initializers define both the data members of the type and their initial values.

For example, our earlier object instantiation is translated to

```
new <Projection>f__12 { d.GivenFirstName, d.FamilyLastName }
```

where the compiler has generated

```
public sealed class <Projection>f__12
{
    private string _GivenFirstName;
    private string _FamilyLastName;

    public string GivenFirstName
    { get { return _GivenFirstName; } set { _GivenFirstName = value; } }
    public string FamilyLastName
    { get { return _FamilyLastName; } set { _FamilyLastName = value; } }

    public <Projection>f__12( )              // default constructor: empty
    { }
    public override string ToString()        // dumps contents of all fields
    { ... }
    public override bool Equals(object obj)  // all fields must be Equals
    { ... }
    public override int GetHashCode()
    { ... }
}
```

As you can see, the names and types of the properties are inferred from the initializers. If you prefer, you can assign your own names to the data members of the anonymous type. For example:

```
new { First = d.GivenFirstName, Last = d.FamilyLastName }
```

In this case the compiler generates:

```
public sealed class <Projection>f__13
```

```
{
      private string _First;
      private string _Last;

      public string First
      { get { return _First; } set { _First = value; } }
      public string Last
      { get { return _Last; } set { _Last = value; } }
      .
      .
      .
}
```

Anonymous types are being added to C# 3.0 primarily to support of LINQ, allowing developers to work with just the data they need in a familiar, object-oriented package. And courtesy of type inference, you can work with anonymous types in a safe, type-checked manner.

A current limitation of anonymous types is that they cannot be accessed outside the defining assemblywhat name would you use to refer to the class? In theory you could compile the code in a separate assembly and then use the generated name, but this is an obvious bad practice (and currently prevented by the fact that the generated name is invalid at the source code level). At issue is the design of N-tier applications, whose tiers communicate by passing data. If the data is packaged as an anonymous type, how do the other tiers refer to it? Since they cannot, anonymous types should be viewed as a technology for local use only.

PREV NEXT

Chapter 5. Applying LINQ

LINQ is really the centerpiece of a wide range of data access technologiesfrom datasets to databases, text files to XML documents, objects to object graphs. The extensibility of LINQ is one of its most elegant features, allowing LINQ to query just about any enumerable data source.

In this section we'll look at some of the more interesting ways you can use LINQ to query DataSets, databases, XML documents, text files, and more.

Chapter 2. A Quick Introduction to LINQ

Anders Hejlsberg, the chief architect of the C# programming language, summarizes LINQ quite nicely:

"*It's about turning query set operations and transforms into first-class concepts of the language*" (http://channel9.msdn.com/showpost.aspx?postid=114680)

Let's look at an example. Imagine we are writing a scheduling application for a hospital, where every night a doctor must be on call to handle emergencies. The main objects we are interacting with are of type `Doctor`, stored in a `Doctors` collection that inherits from `List<Doctor>`:

```
Doctors doctors = new Doctors();
```

Suppose we need a list of all the doctors living within the Chicago city limits. The obvious approach involves a simple iteration across the collection:

```
List<Doctor> inchicago = new List<Doctor>();
foreach (Doctor d in doctors)
  if (d.City == "Chicago")
    inchicago.Add(d);
```

In LINQ, this search reduces to the following query, with the IDE providing keyword highlighting, IntelliSense, and compile-time type checking:

```
var inchicago = from d in doctors where d.City == "Chicago" select d;
```

Need a list sorted by last name? How about:

```
var byname = from d in doctors
             where d.City == "Chicago"
             orderby d.FamilyLastName
             select d;
```

Also need a list sorted by pager number? In LINQ:

```
var bypager = from d in doctors
              where d.City == "Chicago"
              orderby d.PagerNumber
              select d;
```

Finally, suppose we need a count of how many doctors live in each city (not just Chicago, but every city for which at least one doctor lives). With LINQ, we can compute this result by grouping the doctors by city, much like we would in SQL:

```
var bycity = from d in doctors
             group d by d.City into g
             orderby g.Key
             select new { City = g.Key, Count = g.Count() };
```

These types of queries begin to reveal the real power of LINQ. The results are easily harvested by iterating across the query, for example:

```
foreach (var result in bycity)
  System.Console.WriteLine("{0}: {1}", result.City, result.Count);
```

This yields the following output for our set of doctors:

```
Chicago: 5
Elmhurst: 1
Evanston: 3
Oak Park: 2
Wilmette: 1
```

LINQ does not limit us to a read-only data access strategy. CRUD-like (create, read, update, and delete) operations are easily performed using the traditional object-oriented approach. For example, have the *Larsens* moved to the suburbs? If so, search for the appropriate Doctor objects and update the City:

```
var larsens = from d in doctors where d.FamilyLastName == "Larsen" select d;
foreach (var d in larsens)
  d.City = "Suburb";
```

Need to create a doctor? Add an instance of the Doctor class to the doctors collection. Need to delete a doctor? Query to find the corresponding Doctor object, and remove from the collection. In short, while the LINQ API does not provide direct support for CRUD, these operations are easily accomplished.

NOTE

Here are the definitions of the Doctors and Doctor classes we'll be using throughout this Short Cut.

```
public class Doctors : List<Doctor>
 {
   public Doctors()  // constructor (fills list with Doctors):
   {
     this.Add( new Doctor("mbl", "Marybeth", "Larsen", 52248,
                          "mlarsen@uhospital.edu", "1305 S. Michigan", "Chicago",
                          new DateTime(1998, 12, 1)) );
     this.Add( new Doctor("jl", "Joe", "Larsen", 52249,
                          "jlarsen@uhospital.edu", "1305 S. Michigan", "Chicago",
                          new DateTime(1999, 11, 1)) );
     this.Add( new Doctor("ch", "Carl", "Harding", 53113,
                          "charding@uhospital.edu", "2103 Oak St", "Evanston",
                          new DateTime(2000, 10, 1)) );
     .
     .
     .
   }
 }
 // continued
  public class Doctor : IComparable<Doctor>
// NOTE: IComparable NOT required by LINQ.
 {
        private string   m_Initials, m_GivenFirstName, m_FamilyLastName;
        private int      m_PagerNumber;
        private string   m_EmailAddress, m_StreetAddress, m_City;
        private DateTime m_StartDate;

        public string Initials
        { get { return m_Initials;       } set { m_Initials = value; }       }
        public string GivenFirstName
        { get { return m_GivenFirstName; } set { m_GivenFirstName = value; } }
        public string FamilyLastName
        { get { return m_FamilyLastName; } set { m_FamilyLastName = value; } }
        public int PagerNumber
        { get { return m_PagerNumber;    } set { m_PagerNumber = value; }    }
        public string EmailAddress
        { get { return m_EmailAddress;   } set { m_EmailAddress = value; }   }
        public string StreetAddress
        { get { return m_StreetAddress;  } set { m_StreetAddress = value; }  }
        public string City
        { get { return m_City;           } set { m_City = value; }           }
        public DateTime StartDate
        { get { return m_StartDate;      } set { m_StartDate = value; }      }

        public Doctor(string initials, string givenFirstName, string familyLastName,
              int pagerNumber, string emailAddress, string streetAddress,
string city,
              DateTime startDate)
        {
              this.m_Initials       = initials;
              this.m_GivenFirstName = givenFirstName;
```

```
            this.m_FamilyLastName = familyLastName;
            this.m_PagerNumber    = pagerNumber;
            this.m_EmailAddress   = emailAddress;
            this.m_StreetAddress  = streetAddress;
            this.m_City           = city;
            this.m_StartDate      = startDate;
        }

        public override bool Equals(object obj)  // NOTE: Equals NOT required by LINQ.
        {
            if (obj == null || this.GetType().Equals(obj.GetType()) == false)
              return false;
            Doctor other = (Doctor) obj;
            return this.m_Initials.Equals(other.m_Initials);
        }

        public override int GetHashCode()  // NOTE: GetHashCode NOT required by LINQ.
        { return this.m_Initials.GetHashCode(); }

        public int CompareTo(Doctor other)
        { return this.m_Initials.CompareTo(other.m_Initials); }
    }
```

PREV NEXT

Copyright

5.3. Create, Read, Update, and Delete with LINQ

Most of the discussion thus far has focused on data querying, and not data modification. Don't be mistaken, LINQ provides full support for read/write data access, commonly referred to as CRUD.

Interestingly, while data is read using an SQL-like query language, data modification is approached using more traditional, object-oriented mechanisms. For example, to schedule the doctor *mbl* for call on November 30, 2006 in our *SchedulingDocs* database, we do two things. First, we add a new row to the object representing the Calls table:

```
db.Calls.Add( new Databases.Calls{Initials="mbl",
                         DateOfCall=new DateTime(2006, 11, 30} );
```

Second, we flush the change back to the database:

```
db.SubmitChanges();
```

The first step makes a local, in-memory change only; the second step is what triggers the underlying SQL (or stored procedure) to update the database. LINQ to SQL will automatically generate the appropriate SQL for inserts, updates, and deletes, or interoperate with your custom stored procedures.

To delete a call, we find the corresponding object, remove it from the table, and update the database:

```
var del = from c in db.Calls
          where c.Initials == "mbl" && c.DateOfCall == new DateTime(2006, 11, 30)
          select c;

foreach (var c in del)
  db.Calls.Remove(c);

db.SubmitChanges();
```

Since there is at most one doctor on call for any given day, we know the above query will return exactly one record. In this case we can use the standard query operator `Single`, passing a lambda expression for the search criteria:

```
var call = db.Calls.Single( c => c.Initials == "mbl" &&
                        c.DateOfCall == new DateTime(2006, 11, 30) );
db.Calls.Remove(call);
db.SubmitChanges();
```

If it's possible that the search may fail, use the query operator `SingleOrDefault`, and check the query result for null.

Finally, to update existing data, the approach is (1) query to find the corresponding objects, (2) update those objects, and (3) flush the changes. For example, if the doctor *ay*'s pager number changes to 53301, we update the database as follows:

```
var ay = db.Doctors.Single( d => d.Initials == "ay" );
ay.PagerNumber = 53301;
db.SubmitChanges();
```

The same logic applies to other LINQ scenarios, such as XML documents and DataSets. For example, with the typed DataSet `SchedulingDocs` (see Figure 2), scheduling a doctor on call is simply a matter of adding a new row to the Calls table:

```
ds.Calls.AddCallsRow( "mbl", new DateTime(2006, 11, 30) );
```

Much like the database objects, DataSets are a local, in-memory collection of objects. To persist your changes, an updated DataSet must be written to some durable medium, such as the file system or a database:

```
dbConn.Open();
callsAdapter.Update( ds.Calls );
dbConn.Close();
```

Here we re-open the connection, update the database to match, and close the connectionthe equivalent of `db.SubmitChanges()`. The key difference is that in the case of LINQ to SQL, the SQLMetal tool generates the necessary code to update the underlying database. In the case of DataSets and XML documents (and other flavors of LINQ), it's typically our responsibility to load the data, and consequently to persist it back.

PREV NEXT

Chapter 7. Extending LINQ

In closing, let's touch on the extensibility of LINQ, another of its elegant design characteristics. As you have seen, LINQ interoperates with a range of data sources, from object graphs to relational databases to XML documents. Let's look at a quick example of how we go about extending one of our own classes to natively support LINQ.

Our running demo has focused on doctors working at the fictitious University Hospital. You've seen many queries involving doctors, often using the Distinct operator to eliminate duplicates. For example, a query to yield the distinct doctors working in October 2006:

```
DataSets.SchedulingDocs ds = FillDataSet();

var oct2006 = (
                from doc in ds.Doctors
                join call in ds.Calls
                on doc.Initials equals call.Initials
                where call.DateOfCall >= new DateTime(2006, 10, 1) &&
                      call.DateOfCall <= new DateTime(2006, 10, 31)
                select doc
              )
              .Distinct();
```

Since we designed the data model, we know that a doctor's initials are unique. We can take advantage of this information to optimize the application of Distinct in the query above by providing our own implementation specific to our DataSet:

```
using DS = DataSets.SchedulingDocs;

public static class DataSetsExtension
{

  public static IEnumerable<DS.DoctorsRow> Distinct(this
                                                    IEnumerable<DS.DoctorsRow> src)
  {
    // optimize by using a hashtable with doctors' initials:
    System.Collections.Hashtable ht = new System.Collections.Hashtable();

    foreach(var doc in src)  // for each doctor in the original sequence:
    {
      if (ht[doc.Initials] == null)  // first time, so remember and return:
      {
        ht.Add(doc.Initials, doc.Initials);
        yield return doc;
      }
      else  // we've yielded this doc already, so skip:
        continue;
    }
  }

}//class
```

For sequences of type DoctorsRowi.e., doctors in our DataSetthis implementation will be used in place of the standard LINQ implementation of Distinct. Thus, for any query of the form,

```
var query = from doc in ds.Doctors
            .
            .
            select doc;
```

the application of Distinct will invoke our optimized implementation. That's it!

4.5. Extension Methods

So how exactly are query expressions executed? Query expressions are translated into traditional object-oriented method calls by way of *extension methods*. Extension methods, new in C# 3.0, extend a class without actually being members of that class. For example, consider the following query expression:

```
using System.Query;

// initials of all doctors living in Chicago, in no particular order:
var chicago = from d in doctors
              where d.City == "Chicago"
              select d.Initials;
```

Using extension methods, this query can be rewritten as follows:

```
using System.Query;

var chicago = doctors.
              Where(d => d.City == "Chicago").
              Select(d => d.Initials);
```

This statement calls two methods, Where and Select, passing them lambda expressionsyet these methods are not members of the Doctors class. How and why does this compile?

In C# 3.0, extension methods are defined as static methods in a static class, annotated with the System.Runtime.CompilerServices.Extension attribute. By importing the namespace that includes this static class, the compiler treats the extension methods as if they were instance methods. For example, the LINQ standard query operators are defined as extension methods in the class System.Query.Sequence. By importing the namespace System.Query, we gain access to the query operators as if they were members of the Doctors class.

Let's look at an extension method in more detail. Here's the signature for System.Query.Sequence.Where:

```
namespace System.Query
{
    public delegate ReturnT Func<ArgT, ReturnT>(ArgT arg);

    public static class Sequence
    {
        public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
                                              Func<T, bool> predicate)
        { ... }
        .
        .
        .
    }
}
```

The extension method Where returns an IEnumerable object by iterating across an existing IEnumerable object (*source*), applying a delegate-based Boolean function (*predicate*) to determine membership in the result set. Notice that both the first parameter and the return value are defined in terms of IEnumerable<T>, establishing the link between query expressions and the extension methods that drive them.

Given a query expression or a query written using extension methods, the C# 3.0 compiler translates these into calls to the underlying static methods. For example, either form of our query above is conceptually translated into the following:

```
var temp    = System.Query.Sequence.Where(doctors, d => d.City == "Chicago");
var chicago = System.Query.Sequence.Select(temp, d => d.Initials);
```

In reality, this form is skipped and the translation proceeds directly to C# 2.0-compatible code:

```
IEnumerable<Doctor> temp;     // doctors living in chicago
IEnumerable<string> chicago;  // initials of doctors living in chicago

temp    = System.Query.Sequence.Where<Doctor>( doctors,
          new Func<Doctor, bool>(b__0) );
chicago = System.Query.Sequence.Select<Doctor, string>( temp,
          new Func<Doctor, string>(b__3) );
```

with the lambda expressions translated into delegate-invoked methods:

```
private static bool b__0(Doctor d)    // lambda: d => d.City == "Chicago"
{ return d.City == "Chicago"; }

private static string b__3(Doctor d)  // lambda: d => d.Initials
{ return d.Initials; }
```

Notice the important role that type inference plays during this translation. Extension methods and their supporting types (IEnumerable, Func, etc.) are elegantly defined once using generics. Type inference is relied on to determine the type T involved in each aspect of the query, and to then qualify the generic appropriately. In this case we see the inference of both Doctor and string.

In C# 3.0, what differentiates an extension method from an ordinary static method? Observe that a query based on extension methods is converted to a static version by passing the object instance as the first argument:

```
doctors.Where(...) ==> System.Query.Sequence.Where(doctors, ...)
```

This is identical to the standard mechanism for calling instance methods, where the first parameter is a reference to the object itselfproviding a value for this. This logic explains C#'s choice of the this keyword in the signature of extension methods:

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source, ... )
```

In C# 3.0 it is the presence of the this keyword on the first parameter that identifies a static method as an extension method.

What if the class (such as Doctors) or one of its base classes contain methods that conflict with imported extension methods? To avoid unexpected behavior, all other methods take priorityextension methods have the lowest precedence when the compiler performs name resolution, and thus become candidates only after all other possibilities have been exhausted. If two or more imported namespaces yield candidate extension methods, the compiler reports the conflict as a compilation error.

PREV NEXT

Chapter 8. For Further Information

For all things LINQ, start at http://msdn.microsoft.com/data/ref/linq/. This short video with Anders Hejlsberg and Luca Bolognese provides a great overview of LINQ: http://channel9.msdn.com/showpost.aspx?postid=114680. Finally, keep an eye out for LINQ-related webcasts by yours truly, which will be announced at the O'Reilly site for this Short Cut (http://www.oreilly.com/catalog/language1) and on my blog (http://www.pluralsight.com/blogs/drjoe/).

PREV NEXT

6.1.

6.1.1. Aggregate

The Aggregate operator applies a function over a sequence, with or without an initial seed value. The result can be post-processed by another function is desired.

```
int[] ints  = { 1, 2, 3, 4, 5, 6 };
var   query = from ...;

int sum     = ints.Aggregate( (a, b) => a + b );
int product = ints.Aggregate( 1, (a, b) => a * b );
int sump1   = ints.Aggregate( 0, (a, b) => a + b, r => r+1 );
var result  = query.Aggregate(...);

Console.WriteLine(sum);      // 21
Console.WriteLine(product);  // 720
Console.WriteLine(sump1);    // 22
Console.WriteLine(result);
```

The sequence can be of any type *T*.

See also: Average, Count, LongCount, Max, Min, and Sum.

Table 1. List of Standard LINQ Query Operators

Operator	Lazy?	Description
Aggregate	No	Applies a function to a sequence, yielding a single value.
All	No	Applies a function to a sequence to see if all elements satisfy the function.
Any	No	Applies a function to a sequence to see if any element satisfies the function.
Average	No	Computes the average of a numeric sequence.
Cast	Yes	Yields the elements of a sequence type-casted to a given type.
Concat	Yes	Yields the concatenation of two sequences S1 and S2.
Contains	No	Searches a sequence to see if it contains a given element.
Count	No	Counts the number of elements in a sequence, yielding an integer result.
DefaultIfEmpty	Yes	Given a sequence S, yields S or a sequence with the default value if S is empty.
Distinct	Yes	Returns a sequence with duplicates eliminated.
ElementAt	No	Returns the i^th element of a sequence.
ElementAtOrDefault	No	Returns the i^th element of a sequence, or the default value if sequence is empty.
Empty	Yes	Yields an empty sequence.
EqualAll	No	Compares two sequences for equality.
Except	Yes	Given two sequences S1 and S2, returns the set difference S1 S2.
First	No	Returns the first element of a sequence.
FirstOrDefault	No	Returns the first element of a sequence, or the default value if sequence is empty.
Fold	No	Obsolete, see Aggregate.
GroupBy	Yes	Groups the elements of a sequence by key.
GroupJoin	Yes	Performs a join of two sequences S1 and S2, yielding a hierarchical result.
Intersect	Yes	Given two sequences S1 and S2, returns the set intersection of S1 and S2.
Join	Yes	Performs a traditional inner equijoin of two sequences S1 and S2.
Last	No	Returns the last element of a sequence.
LastOrDefault	No	Returns the last element of a sequence, or the default value if sequence is empty.

LongCount	No	Counts the number of elements in a sequence, yielding a long result.
Max	No	Returns the maximum of a sequence.
Min	No	Returns the minimum of a sequence.
OfType	Yes	Yields the elements of a sequence that match a given type.
OrderBy	Yes	Orders a sequence of elements by key into ascending order.
OrderByDescending	Yes	Orders a sequence of elements by key into descending order.
Range	Yes	Yields a sequence of integers in a given range.
Repeat	Yes	Yields a sequence of values by repeating a given value n times.
Reverse	Yes	Reverses the elements of a sequence.
Select	Yes	Applies a projection function to a sequence, yielding a new sequence.
SelectMany	Yes	Applies a projection function to flatten a sequence of sequences.
Single	No	Returns the lone element of a singleton sequence.
SingleOrDefault	No	Returns the lone element of a singleton sequence, or default if sequence is empty.
Skip	Yes	Skips the first n elements of a sequence, yielding the remaining elements.
SkipWhile	Yes	Given function F and sequence S, skips the initial elements of S where F is true.
Sum	No	Computes the sum of a numeric sequence.
Take	Yes	Yields the first n elements of a sequence.
TakeWhile	Yes	Given function F and sequence S, yields the initial elements of S where F is true.
ThenBy	Yes	Takes an ordered sequence and yields a secondary, ascending ordering.
TheyByDescending	Yes	Takes an ordered sequence and yields a secondary, descending ordering.
ToArray	No	Iterates across a sequence, capturing results in an array.
ToDictionary	No	Iterates across a sequence, capturing results in a *Dictionary<K, V>*.
ToList	No	Iterates across a sequence, capturing results in a *List<T>*.
ToLookup	No	Iterates across a sequence, capturing results in a *Lookup<K, IEnumerable<V>>*.
ToSequence	Yes	Casts a sequence as an *IEnumerable* sequence for use with standard query ops.
Union	Yes	Given two sequences S1 and S2, returns the set union of S1 and S2.
Where	Yes	Applies a Boolean function to a sequence, yielding a sub-sequence.

6.1.2. All

The All operator applies a function over a sequence, checking to see if all of the elements satisfy the function, i.e., cause the function to return true. For example, do all the doctors have pager numbers? How about all the doctors retrieved by a particular query?

```
Doctors doctors = new Doctors();

var query = from doc in doctors ...;

bool allHavePagers   = doctors.All(doc => doc.PagerNumber > 0);
bool theseHavePagers = query.All(doc => doc.PagerNumber > 0);
```

See also: Any, Contains, and EqualAll.

6.1.3. Any

The Any operator applies a function over a sequence, checking to see if any of the elements satisfy the function, i.e., cause the function to return true. For example, are there any doctors living in Lake Forest?

```
Doctors doctors = new Doctors();

bool inLakeForest = doctors.Any(doc => doc.City == "Lake Forest");
```

The function is optional; if omitted, the `Any` operator returns true if the sequence contains at least one element.

```
var query = from doc in doctors
            where doc.City == "Lake Forest"
            select doc;

bool inLakeForest = query.Any();
```

See also: All, Contains, and EqualAll.

6.1.4. Average

The `Average` operator computes the average of a sequence of numeric values. The values are either the sequence itself, or selected out of a sequence of objects.

```
int[]      ints    = { 1, 2, 3, 4, 5, 6 };
decimal?[] values  = { 1, null, 2, null, 3, 4 };
Doctors    doctors = new Doctors();
var        query   = from ...;

double   avg1     = ints.Average();
decimal? avg2     = values.Average();
double   avgYears = doctors.Average( doc =>
                      DateTime.Now.Subtract(doc.StartDate).Days / 365.25 );
var      avg      = query.Average();

Console.WriteLine(avg1);   // 3.5
Console.WriteLine(avg2);   // 2.5
Console.WriteLine(avgYears.ToString("0.00"));   // 5.72
Console.WriteLine(avg);
```

The values can be of type int, int?, long, long?, decimal, decimal?, double, or double?. The resulting type is double, double?, double, double?, decimal, decimal?, double, or double?, respectively.

See also: Aggregate, Count, LongCount, Max, Min, and Sum.

6.1.5. Cast

The `Cast` operator yields the elements of a sequence type-casted to a given type *T*.

```
System.Collections.ArrayList al = new System.Collections.ArrayList();
al.Add("abc");
al.Add("def");
al.Add("ghi");

var strings = al.Cast<string>();

foreach(string s in strings)   // "abc", "def", "ghi"
  Console.WriteLine(s);
```

The `Cast` operator is commonly used to wrap pre-2.0 collections (such as ArrayLists) for use with LINQ. Here's a more interesting example of searching the Windows event log for all events logged by an application:

```
var entries = from entry in applog.Entries.Cast<System.Diagnostics.EventLogEntry>()
              where entry.Source == "ApplicationName"
              orderby entry.TimeWritten descending
              select entry;
```

See also: OfType.

6.1.6. Concat

The Concat operator concatenates two sequences S1 and S2, yielding the elements of S1 followed by the elements of S2.

```
int[] ints1  = { 1, 2, 3 };
int[] ints2  = { 4, 5, 6 };
var   query1 = from ...;
var   query2 = from ...;

var all     = ints1.Concat(ints2);
var results = query1.Concat(query2);

foreach(var x in all)     // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(var r in results)
  Console.WriteLine(r);
```

The sequences may contain elements of any type *T*. This element type T must be the same at compile-time (e.g. object), but may differ at run-time:

```
object[] objects1 = { "abc", "def" };
object[] objects2 = { 1, 2, 3 };

var result = objects1.Concat(objects2);
```

See also: Union.

6.1.7. Contains

The Contains operator searches a sequence to see if it contains a given element. For example, is there a doctor with initials "*gg*" still working at University Hospital? One approach is to create a Doctor object with the initials we are looking for, and see if the collection contains this object:

```
Doctors doctors = new Doctors();

bool docExists = doctors.Contains( new Doctor("gg", ...) );
```

This assumes the Doctor class defines Equals based on a doctor's initials. Another approach (without this assumption) is to select all the initials and then see if the result contains the string "gg":

```
var query = from doc in doctors
            select doc.Initials

bool docExists = query.Contains("gg");
```

See also: All, Any, EqualAll, and Where.

6.1.8. Count

The Count operator counts the number of elements in a sequence, yielding an integer result. The elements are either the sequence itself, or selected from a sequence of objects.

```
int[]       ints   = { 1, 2, 3, 4, 5, 6 };
decimal?[] values = { 1, null, 2, null, 3 };
IEnumerable<Doctor> doctors = new Doctors();
var         query  = from ...;

int count1 = ints.Count();
int count2 = values.Count();
int count3 = doctors.Count();
int count4 = doctors.Count( doc => doc.City == "Chicago" );
int count  = query.Count();
```

```
Console.WriteLine(count1);   // 6
Console.WriteLine(count2);   // 5
Console.WriteLine(count3);   // 12
Console.WriteLine(count4);   // 5
Console.WriteLine(count);
```

The sequence can be of any type T.

See also: Aggregate, Average, LongCount, Max, Min, and Sum.

6.1.9. DefaultIfEmpty

Given a non-empty sequence, the DefaultIfEmpty operator yields this same sequence. If the sequence is empty, DefaultIfEmpty yields a sequence containing a single default value.

```
int[]   ints1   = { 1, 2, 3, 4, 5, 6 };
int[]   ints2   = { };
var     query   = from ...;

var  ints   = ints1.DefaultIfEmpty();
var  zero   = ints2.DefaultIfEmpty();
var  minus1 = ints2.DefaultIfEmpty(-1);
var  result = query.DefaultIfEmpty();

foreach(int x in ints)     // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(int x in zero)     // 0
  Console.WriteLine(x);
foreach(int x in minus1)   // -1
  Console.WriteLine(x);
foreach(var r in result)
  Console.WriteLine(r);
```

The sequence can be of any type T; if the sequence is empty and a default value is not provided, the default value for type T is used.

See also: FirstOrDefault, GroupJoin, LastOrDefault, SingleOrDefault, and ElementAtOrDefault.

6.1.10. Distinct

Given a sequence of elements, the Distinct operator returns the same sequence without duplicates.

```
int[] ints  = { 1, 2, 2, 3, 2, 3, 4 };
var   query = from ...;

var distinctInts    = ints.Distinct();
var distinctResults = query.Distinct();

foreach(var x in distinctInts)  // 1, 2, 3, 4
  Console.WriteLine(x);
foreach(var r in distinctResults)
  Console.WriteLine(r);
```

The sequence can be of any type T.

See also: Except, Intersect, and Union.

6.1.11. ElementAt

The ElementAt operator returns the *i*th element of a sequence; the sequence must be non-empty, and *i* is 0-based.

```
int[]   ints    = { 1, 2, 3, 4, 5, 6 };
Doctors doctors = new Doctors();
var     query   = from ...;

int    third  = ints.ElementAt(2);
Doctor doctor = doctors.ElementAt(2);
var    result = query.ElementAt(i);

Console.WriteLine(third);            // 3
Console.WriteLine(doctor.Initials);  // 3rd doc in Chicago: ch
Console.WriteLine(result);
```

The sequence can be of any type *T*.

See also: ElementAtOrDefault.

6.1.12. ElementAtOrDefault

The ElementAtOrDefault operator returns the *i*th element of a possibly empty sequence; *i* is 0-based.

```
int[]   ints1   = { 1, 2, 3, 4, 5, 6 };
int[]   ints2   = { };
Doctors doctors = new Doctors();
var     query   = from ...;

int    x1     = ints1.ElementAtOrDefault(2);
int    x2     = ints1.ElementAtOrDefault(6);
int    x3     = ints2.ElementAtOrDefault(0);
Doctor doc1   = doctors.ElementAtOrDefault(2);
Doctor doc2   = doctors.ElementAtOrDefault(-1);
var    result = query.ElementAtOrDefault(i);

Console.WriteLine(x1);           // 3
Console.WriteLine(x2);           // 0
Console.WriteLine(x3);           // 0
Console.WriteLine(doc1 == null);  // False
Console.WriteLine(doc2 == null);  // True
Console.WriteLine(result);
```

The sequence can be of any type T; if the sequence is empty or *i* is invalid, the default value for type T is returned.

See also: ElementAt.

6.1.13. Empty

The Empty operator yields an empty sequence of the given type.

```
var emptyDocs = System.Query.Sequence.Empty<Doctor>();

foreach(var doc in emptyDocs)   // <none>
  Console.WriteLine(doc);
```

This operator is helpful when you need an empty sequence for another operator or a method argument.

See also: Range and Repeat.

6.1.14. EqualAll

The EqualAll operator compares two sequences for equality. Two sequences are equal if they are of the same length, and contain the same sequence of elements.

```
var query1 = from ...;
var query2 = from ...;

bool equal = query1.EqualAll(query2);
```

The sequences may contain elements of any type *T*. This element type T must be the same at compile-time (e.g. object), but may differ at run-time:

```
object[] objects1 = { "abc", "def" };
object[] objects2 = { 1, 2, 3 };

bool isFalse = objects1.EqualAll(objects2);  // false
```

The result in such cases is false.

See also: All, Any, and Contains.

6.1.15. Except

Given two sequences of elements S1 and S2, the Except operator returns the distinct elements of S1 not in S2. In other words, Except computes the set difference S1 S2.

```
int[] intsS1 = { 1, 2, 2, 3, 2, 3, 4, 5, 6 };
int[] intsS2 = { 1, 3, 6, 7 };
var   query1 = from ...;
var   query2 = from ...;

var diffInts    = intsS1.Except(intsS2);
var diffResults = query1.Except(query2);

foreach(var x in diffInts)  // 2, 4, 5
  Console.WriteLine(x);
foreach(var r in diffResults)
  Console.WriteLine(r);
```

The sequences may contain elements of any type *T*. This element type T must be the same at compile-time (e.g. object), but may differ at run-time:

```
object[] objects1 = { "abc", "def" };
object[] objects2 = { 1, 2, 3 };

var result = objects1.Except(objects2);
```

See also: Distinct, Intersect, and Union.

6.1.16. First

The First operator returns the first element of a sequence; the sequence must be non-empty.

```
int[]   ints    = { 1, 2, 3, 4, 5, 6 };
Doctors doctors = new Doctors();
var     query   = from ...;

int    first  = ints.First();
Doctor doctor = doctors.First(doc => doc.City == "Chicago");
var    result = query.First();

Console.WriteLine(first);            // 1
Console.WriteLine(doctor.Initials);  // mbl
Console.WriteLine(result);
```

The sequence can be of any type T.

See also: FirstOrDefault.

6.1.17. FirstOrDefault

The FirstOrDefault operator returns the first element of a possibly empty sequence.

```
int[]   ints1   = { 1, 2, 3, 4, 5, 6 };
int[]   ints2   = { };
Doctors doctors = new Doctors();
var     query   = from ...;

int    x1      = ints1.FirstOrDefault();
int    x2      = ints2.FirstOrDefault();
Doctor doc1    = doctors.FirstOrDefault(doc => doc.City == "Chicago");
Doctor doc2    = doctors.FirstOrDefault(doc => doc.City == "Planet Mars");
var    result = query.FirstOrDefault();

Console.WriteLine(x1);             // 1
Console.WriteLine(x2);             // 0
Console.WriteLine(doc1 == null);   // False
Console.WriteLine(doc2 == null);   // True
Console.WriteLine(result);
```

The sequence can be of any type T; if the sequence is empty, the default value for type T is returned.

See also: First.

6.1.18. Fold

The Fold operator is considered obsolete, see Aggregate.

6.1.19. GroupBy

The GroupBy operator groups the elements of a sequence by key; the keys are yielded by a function applied to each element. Each resulting group is an *IEnumerable* sequence of elements S with a key K.

```
Doctors doctors = new Doctors();

var groups = doctors.GroupBy(doc => doc.City);

foreach(var group in groups)
{
  Console.WriteLine("{0}:", group.Key);   // Chicago, Evanston, ...
  foreach(var doc in group)               // {mbl,jl,...}, {ch,cm,...}, ...
    Console.WriteLine("  {0}", doc.Initials);
}
```

A second version allows you to project exactly what data to store in the group, such as only the doctor's initials:

```
Doctors doctors = new Doctors();

var groups2 = doctors.GroupBy(doc => doc.City, doc => doc.Initials);

foreach(var group in groups2)
{
  Console.WriteLine("{0}:", group.Key);   // Chicago, Evanston, ...
  foreach(var initials in group)          // {mbl,jl,...}, {ch,cm,...}, ...
    Console.WriteLine("  {0}", initials);
}
```

Additional versions allow you to provide a comparer of type *IEqualityComparer* for comparing keys.

Use of the *group by* clause in a query expression translates into application of the GroupBy operator. For example, the following statements are equivalent:

```
var groups = doctors.GroupBy(doc => doc.City);
var groups = from doc in doctors
             group doc by doc.City into g
             select g;

var groups2 = doctors.GroupBy(doc => doc.City, doc => doc.Initials);
var groups2 = from doc in doctors
              group doc.Initials by doc.City into g
              select g;
```

See also: OrderBy.

6.1.20. GroupJoin

The GroupJoin operator performs a join of two sequences S1 and S2, based on the keys selected from S1's and S2's elements. The keys are yielded by functions applied to each element; a third function determines the data projected by the join.

Unlike an inner join which yields essentially a table of joined records, the result of a GroupJoin is hierarchical. For each element in S1, there's a possibly empty sub-sequence of matching elements from S2. For example, the following query joins Doctors and Calls via the doctors' initials to determine which doctors are working on what dates:

```
DataSets.SchedulingDocs ds = FillDataSet();

var working = ds.Doctors.GroupJoin( ds.Calls,
                                    doc => doc.Initials,
                                    call => call.Initials,
                                    (doc,call) => new { doc.Initials, Calls=call }
                                  );

foreach(var record in working)
{
  Console.WriteLine("{0}:", record.Initials);
  foreach(var call in record.Calls)
    Console.WriteLine("  {0}", call.DateOfCall);
}
```

Here's the output:

```
ay:
  11/2/2006 12:00:00 AM
bb:
ch:
cm:
jl:
  10/2/2006 12:00:00 AM
  11/1/2006 12:00:00 AM
.
.
.
```

Use of the *join into* clause in a query expression translates into application of the GroupJoin operator. For example, the following statements are equivalent:

```
var working = ds.Doctors.GroupJoin( ds.Calls,
                                    doc => doc.Initials,
                                    call => call.Initials,
                                    (doc,call) => new { doc.Initials, Calls=call }
                                  );

var working = from doc in ds.Doctors
              join call in ds.Calls
              on doc.Initials equals call.Initials
              into j
              select new { doc.Initials, Calls = j };
```

To perform a traditional left outer join (and thus flatten the result into a table), use the DefaultIfEmpty operator as follows:

```
var working = from doc in ds.Doctors
              join call in ds.Calls
              on doc.Initials equals call.Initials
              into j
              from r in j.DefaultIfEmpty()
              select new { doc.Initials, Call = r };

foreach(var record in working)
  Console.WriteLine("{0}: {1}",
    record.Initials,
    (record.Call == null) ? "" : record.Call.DateOfCall.ToString());
```

See also: Join.

6.1.21. Intersect

Given two sequences of elements S1 and S2, the Intersect operator returns the distinct elements of S1 that also appear in S2. In other words, Intersect computes the set intersection of S1 and S2.

```
int[] intsS1 = { 1, 2, 2, 3, 2, 3, 4, 5, 6, 8 };
int[] intsS2 = { 6, 1, 3, 6, 7 };
var   query1 = from ...;
var   query2 = from ...;

var commonInts    = intsS1.Intersect(intsS2);
var commonResults = query1.Intersect(query2);

foreach(var x in commonInts)  // 1, 3, 6
  Console.WriteLine(x);
foreach(var r in commonResults)
  Console.WriteLine(r);
```

The sequences may contain elements of any type *T*. This element type T must be the same at compile-time (e.g. object), but may differ at run-time:

```
object[] objects1 = { "abc", "def" };
object[] objects2 = { 1, 2, 3 };

var result = objects1.Intersect(objects2);
```

See also: Distinct, Except, and Union.

6.1.22. Join

The Join operator performs an inner equijoin of two sequences S1 and S2, based on the keys selected from S1's and S2's elements. The keys are yielded by functions applied to each element; a third function determines the data projected by the join. For example, the following query joins Doctors and Calls via the doctors' initials to determine which doctors are working on what dates:

```
DataSets.SchedulingDocs ds = FillDataSet();

var working = ds.Doctors.Join( ds.Calls,
                               doc => doc.Initials,
                               call => call.Initials,
                               (doc,call) => new { doc.Initials, call.DateOfCall }
                              );

foreach(var record in working)
  Console.WriteLine(record);
```

The query yields pairs of the form (Initials, DateOfCall), created from the joined Doctor and Call elements. Here's the output:

```
{Initials=ay, DateOfCall=11/2/2006 12:00:00 AM}
{Initials=jl, DateOfCall=10/2/2006 12:00:00 AM}
{Initials=jl, DateOfCall=11/1/2006 12:00:00 AM}
.
.
.
```

Use of the *join* clause in a query expression translates into application of the Join operator. For example, the following statements are equivalent:

```
var working = ds.Doctors.Join( ds.Calls,
                               doc => doc.Initials,
                               call => call.Initials,
                               (doc,call) => new { doc.Initials, call.DateOfCall }
                              );

var working = from doc in ds.Doctors
              join call in ds.Calls
              on doc.Initials equals call.Initials
              select new { doc.Initials, call.DateOfCall };
```

See also: GroupJoin.

6.1.23. Last

The Last operator returns the last element of a sequence; the sequence must be non-empty.

```
int[]   ints    = { 1, 2, 3, 4, 5, 6 };
Doctors doctors = new Doctors();
var     query   = from ...;

int    last   = ints.Last();
Doctor doctor = doctors.Last(doc => doc.City == "Chicago");
var    result = query.Last();

Console.WriteLine(last);              // 6
Console.WriteLine(doctor.Initials);   // tm
Console.WriteLine(result);
```

The sequence can be of any type T.

See also: LastOrDefault.

6.1.24. LastOrDefault

The LastOrDefault operator returns the last element of a possibly empty sequence.

```
int[]   ints1   = { 1, 2, 3, 4, 5, 6 };
int[]   ints2   = { };
Doctors doctors = new Doctors();
var     query   = from ...;

int    x1     = ints1.LastOrDefault();
int    x2     = ints2.LastOrDefault();
Doctor doc1   = doctors.LastOrDefault(doc => doc.City == "Chicago");
Doctor doc2   = doctors.LastOrDefault(doc => doc.City == "Planet Mars");
var    result = query.LastOrDefault();

Console.WriteLine(x1);             // 6
Console.WriteLine(x2);             // 0
Console.WriteLine(doc1 == null);   // False
Console.WriteLine(doc2 == null);   // True
Console.WriteLine(result);
```

The sequence can be of any type T; if the sequence is empty, the default value for type T is returned.

See also: Last.

6.1.25. LongCount

The LongCount operator counts the number of elements in a sequence, yielding a long result. The elements are either the sequence itself, or selected from a sequence of objects.

```
int[]      ints   = { 1, 2, 3, 4, 5, 6 };
decimal?[] values = { 1, null, 2, null, 3 };
IEnumerable<Doctor> doctors = new Doctors();
var        query  = from ...;

long count1 = ints.LongCount();
long count2 = values.LongCount();
long count3 = doctors.LongCount();
long count4 = doctors.LongCount( doc => doc.City == "Chicago" );
long count  = query.LongCount();

Console.WriteLine(count1);   // 6
Console.WriteLine(count2);   // 5
Console.WriteLine(count3);   // 12
Console.WriteLine(count4);   // 5
Console.WriteLine(count);
```

The sequence can be of any type T.

See also: Aggregate, Average, Count, Max, Min, and Sum.

6.1.26. Max

The Max operator finds the maximum of a sequence of values. The values are either the sequence itself, or selected out of a sequence of objects, and must implement *IComparable* for comparison purposes.

```
int[]      ints    = { 1, 2, 3, 6, 5, 4 };
decimal?[] values  = { 1, null, 4, null, 3, 2 };
Doctors    doctors = new Doctors();
var        query   = from ...;

int      max1        = ints.Max();
decimal? max2        = values.Max();
string   maxInitials = doctors.Max( doc => doc.Initials );
double   maxYears    = doctors.Max( doc =>
                         DateTime.Now.Subtract(doc.StartDate).Days / 365.25 );
var      max         = query.Max();

Console.WriteLine(max1);          // 6
Console.WriteLine(max2);          // 4
Console.WriteLine(maxInitials);   // "vj"
Console.WriteLine(maxYears.ToString("0.00"));   // 11.43
Console.WriteLine(max);
```

The sequence can be of any type T; the resulting type is the same.

See also: Aggregate, Average, Count, LongCount, Min, and Sum.

6.1.27. Min

The Min operator finds the minimum of a sequence of values. The values are either the sequence itself, or selected out of a sequence of objects, and must implement *IComparable* for comparison purposes.

```
int[]      ints    = { 6, 2, 3, 1, 5, 4 };
decimal?[] values  = { 4, null, 1, null, 3, 2 };
Doctors    doctors = new Doctors();
var        query   = from ...;

int      min1        = ints.Min();
decimal? min2        = values.Min();
```

```
string   minInitials = doctors.Min( doc => doc.Initials );
double   minYears    = doctors.Min( doc =>
                         DateTime.Now.Subtract(doc.StartDate).Days / 365.25 );
var      min         = query.Min();

Console.WriteLine(min1);          // 1
Console.WriteLine(min2);          // 1
Console.WriteLine(minInitials);   // "ay"
Console.WriteLine(minYears.ToString("0.00"));   // 0.67
Console.WriteLine(min);
```

The sequence can be of any type T; the resulting type is the same.

See also: Aggregate, Average, Count, LongCount, Max, and Sum.

6.1.28. OfType

The OfType operator yields the elements of a sequence that match a given type T. In other words, OfType filters a sequence by type.

```
System.Collections.ArrayList  al = new System.Collections.ArrayList();
al.Add(1);
al.Add("abc");
al.Add(2);
al.Add("def");
al.Add(3);

var strings = al.OfType<string>();

foreach(string s in strings)   // "abc", "def"
    Console.WriteLine(s);
```

The OfType operator is commonly used to (a) wrap pre-2.0 collections (such as ArrayLists) for use with LINQ and (b) filter mixed collections. Here's a more interesting example of searching a user's Outlook contacts with LINQ:

```
Outlook.MAPIFolder folder = this.ActiveExplorer().Session.
  GetDefaultFolder( Outlook.OlDefaultFolders.olFolderContacts );

var contacts = from contact in folder.Items.OfType<Outlook.ContactItem>()
               where .
                     .  // search criteria, e.g. contact.Email1Address != null
                     .
               select contact;
```

See also: Cast.

6.1.29. OrderBy

The OrderBy operator orders a sequence of elements into ascending order; the keys used to order the sequence are yielded by a function applied to each element.

```
int[] ints      = {3, 1, 6, 4, 2, 5};
Doctors doctors = new Doctors();

var sorted = ints.OrderBy(x => x);
var docs   = doctors.OrderBy(doc => doc.Initials);

foreach(var x in sorted)   // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(var doc in docs)   // ay, bb, ..., vj
  Console.WriteLine(doc.Initials);
```

A second version allows you to provide a comparer of type *IComparer* for comparing keys and thus controlling the ordering, e.g., when you want to compare strings in a case-insensitive manner:

```
var docs = doctors.OrderBy(doc => doc.Initials,
                           StringComparer.CurrentCultureIgnoreCase);
```

Use of the *orderby* clause in a query expression translates into application of the OrderBy operator. For example, the following statements are equivalent:

```
var docs = doctors.OrderBy(doc => doc.Initials);

var docs = from doc in doctors
           orderby doc.Initials
           select doc;
```

See also: OrderByDescending, ThenBy, and ThenByDescending.

6.1.30. OrderByDescending

The OrderByDescending operator orders a sequence of elements into descending order; the keys used to order the sequence are yielded by a function applied to each element.

```
int[] ints       = {3, 1, 6, 4, 2, 5};
Doctors doctors = new Doctors();

var sorted = ints.OrderByDescending(x => x);
var docs   = doctors.OrderByDescending(doc => doc.Initials);

foreach(var x in sorted)   // 6, 5, 4, 3, 2, 1
  Console.WriteLine(x);
foreach(var doc in docs)   // vj, tm, ..., ay
  Console.WriteLine(doc.Initials);
```

A second version allows you to provide a comparer of type *IComparer* for comparing keys and thus controlling the ordering, e.g., when you want to compare strings in a case-insensitive manner:

```
var docs = doctors.OrderByDescending(doc => doc.Initials,
                                     StringComparer.CurrentCultureIgnoreCase);
```

Use of the *orderby* clause in a query expression with the *descending* keyword translates into application of the OrderByDescending operator. For example, the following statements are equivalent:

```
var docs = doctors.OrderByDescending(doc => doc.Initials);

var docs = from doc in doctors
           orderby doc.Initials descending
           select doc;
```

See also: OrderBy, ThenBy, and ThenByDescending.

6.1.31. Range

The Range operator yields a sequence of integers in the given range, inclusive.

```
var oneToTen = System.Query.Sequence.Range(1, 10);

foreach(var x in oneToTen)  // 1, 2, 3, ..., 10
  Console.WriteLine(x);
```

See also: Empty and Repeat.

6.1.32. Repeat

The Repeat operator yields a sequence of values by repeating a given value *n* times.

```
var zeros   = System.Query.Sequence.Repeat(0, 8);
var strings = System.Query.Sequence.Repeat("", n);

Console.WriteLine(zeros.Count());     // 8
Console.WriteLine(strings.Count());   // n

foreach(var zero in zeros)   // 0, 0, ..., 0
  Console.WriteLine(zero);
```

See also: Empty and Range.

6.1.33. Reverse

The Reverse operator reverses the elements of a sequence.

```
int[] ints      = {1, 2, 3, 4, 5, 6};
Doctors doctors = new Doctors();

var revInts = ints.Reverse();
var docs    = from doc in doctors
              orderby doc.Initials
              select doc;
var revDocs = docs.Reverse();

foreach(var x in revInts)     // 6, 5, 4, 3, 2, 1
  Console.WriteLine(x);
foreach(var doc in revDocs)   // vj, tm, ..., ay
  Console.WriteLine(doc.Initials);
```

See also: OrderBy and OrderByDescending.

6.1.34. Select

The Select operator applies a projection function over a sequence of elements, yielding a sequence of possibly new elements.

```
int[] ints      = {1, 2, 3, 4, 5, 6};
Doctors doctors = new Doctors();

var sameInts = ints.Select(x => x);
var chicago  = doctors.Where(d => d.City == "Chicago").Select(d => d.Initials);
var names    = doctors.Select(d => new {LN=d.FamilyLastName, FN=d.GivenFirstName});

foreach(var x in sameInts)   // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(var d in chicago)    // mbl, jl, kl, py, tm
  Console.WriteLine(d);
foreach(var n in names)      // {LN=Larsen, FN=Marybeth}, ...
  Console.WriteLine(n);
```

Use of the *select* clause in a query expression translates into application of the Select operator (except in trivial cases where the Select operator can be optimized away). For example, the following statements are equivalent:

```
var chicago = doctors.Where(d => d.City == "Chicago").Select(d => d.Initials);

var chicago = from d in doctors
              where d.City == "Chicago"
              select d.Initials;
```

A second version of the Select operator provides an element's 0-based index in the sequence along with the element itself, for example:

```
var sumLast3 = ints.Select( (x,index) => (index >= ints.Length-3) ? x : 0 ).Sum();

Console.WriteLine(sumLast3);   // 15
```

See also: SelectMany and Where.

6.1.35. SelectMany

The SelectMany operator applies a projection function over a sequence of sequences, yielding a "flattened" sequence of possibly new elements. For example, turning an array of arrays into an array:

```
List<int[]> list = new List<int[]>();
int[] ints123 = {1, 2, 3};
int[] ints456 = {4, 5, 6};

list.Add(ints123);
list.Add(ints456);

var flat = list.SelectMany(x => x);

foreach (var x in list)   // Int32[], Int32[]
  Console.WriteLine(x);
foreach (var x in flat)   // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
```

The SelectMany operator is commonly used to flatten hierarchical data. For example, the following query yields the list of doctors working in October 2006, with sub-lists of the calls each is working:

```
DataSets.SchedulingDocs ds = FillDataSet();

var oct2006 = from d in ds.Doctors
              join c in ds.Calls
              on d.Initials equals c.Initials
              where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                    c.DateOfCall <= new DateTime(2006, 10, 31)
              group c by d.Initials into g
              select g;

foreach (var group in oct2006)  // jl: 1, mbl: 2
{
  Console.WriteLine("{0}: {1}", group.Key, group.Count());
  foreach (var call in group)
    Console.WriteLine(call.DateOfCall);
}
```

To flatten the hierarchy into just the list of calls:

```
var calls = oct2006.SelectMany(call => call);

foreach (var c in calls)
  Console.WriteLine(c.DateOfCall);
```

Use of the *select* clause in a query expression on all but the initial *from* clause translates into application of the SelectMany operator. For example, here's another way to flatten the list of calls for doctors working in October 2006:

```
var calls = from d in ds.Doctors
                from c in ds.Calls
                where
                  d.Initials == c.Initials &&
                  c.DateOfCall >= new DateTime(2006, 10, 1) &&
                  c.DateOfCall <= new DateTime(2006, 10, 31)
                select c;
```

This is translated into the following application of the SelectMany operator:

```
var calls = ds.Doctors.SelectMany(
              d => ds.Calls.Where(c => d.Initials == c.Initials &&
                                 c.DateOfCall >= new DateTime(2006, 10, 1) &&
                                 c.DateOfCall <= new DateTime(2006, 10, 31)));
```

A second version of the SelectMany operator provides an element's 0-based index in the outer sequence along with the element itself.

See also: Select and Where.

6.1.36. Single

The Single operator returns the lone element of a sequence; the sequence must contain exactly one element.

```
int[]   ints    = { 3 };
Doctors doctors = new Doctors();
var     query   = from ...;

int    lone   = ints.Single();
Doctor doctor = doctors.Single(doc => doc.Initials == "mbl");
var    result = query.Single();

Console.WriteLine(lone);                // 3
Console.WriteLine(doctor.PagerNumber);  // 52248
Console.WriteLine(result);
```

The sequence can be of any type T.

See also: SingleOrDefault.

6.1.37. SingleOrDefault

The SingleOrDefault operator returns the lone element of a sequence; the sequence must be empty or contain exactly one element.

```
int[]   ints1   = { 3 };
int[]   ints2   = { };
Doctors doctors = new Doctors();
var     query   = from ...;

int    x1     = ints1.SingleOrDefault();
int    x2     = ints2.SingleOrDefault();
Doctor doc1   = doctors.SingleOrDefault(doc => doc.Initials == "mbl");
Doctor doc2   = doctors.SingleOrDefault(doc => doc.Initials == "AAA");
var    result = query.SingleOrDefault();

Console.WriteLine(x1);            // 3
Console.WriteLine(x2);            // 0
Console.WriteLine(doc1 == null);  // False
Console.WriteLine(doc2 == null);  // True
Console.WriteLine(result);
```

The sequence can be of any type T; if the sequence is empty, the default value for type T is returned.

See also: Single.

6.1.38. Skip

The Skip operator skips the first *n* elements of a sequence, yielding the remaining elements. If *n* <= 0 the result is the sequence itself; if *n* >= sequence's length the result is an empty sequence.

```
int[] ints  = { 1, 2, 3, 4, 5, 6 };
var   query = from ...;

var last3    = ints.Skip(3);
var bottom10 = query.Skip( query.Count() - 10 );

foreach(var x in last3)  // 4, 5, 6
  Console.WriteLine(x);
foreach(var r in bottom10)
  Console.WriteLine(r);
```

The sequence can be of any type T.

See also: Take.

6.1.39. SkipWhile

Given a function F and a sequence S, the SkipWhile operator skips the first *n* elements of S where F returns true, yielding the remaining elements. Two versions of F are supported: accepting an element, or accepting an element along with its 0-based index in the sequence.

```
int[] ints  = { 1, 2, 3, 4, 5, 6 };
var   query = from ...;

var all   = ints.SkipWhile(x => false);
var none  = ints.SkipWhile(x => true);
var last3 = ints.SkipWhile( (x,i) => i < 3 );
var lastN = query.SkipWhile(result => ...);

foreach(var x in all)     // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(var x in none)    // <none>
  Console.WriteLine(x);
foreach(var x in last3)   // 4, 5, 6
  Console.WriteLine(x);
foreach(var r in lastN)
  Console.WriteLine(r);
```

The sequence can be of any type T.

See also: TakeWhile and Where.

6.1.40. Sum

The Sum operator computes the sum of a sequence of numeric values. The values are either the sequence itself or selected out of a sequence of objects.

```
int[]      ints    = { 1, 2, 3, 4, 5, 6 };
decimal?[] values  = { 1, null, 2, null, 3, 4 };
Doctors    doctors = new Doctors();
var        query   = from ...;

int      sum1     = ints.Sum();
decimal? sum2     = values.Sum();
double   sumYears = doctors.Sum( doc =>
                    DateTime.Now.Subtract(doc.StartDate).Days / 365.25 );
var      sum      = query.Sum();

Console.WriteLine(sum1);   // 21
Console.WriteLine(sum2);   // 10
Console.WriteLine(sumYears.ToString("0.00"));   // 68.61
Console.WriteLine(sum);
```

The values can be of type int, int?, long, long?, decimal, decimal?, double, or double?. The resulting type is the same.

See also: Aggregate, Average, Count, LongCount, Max, and Min.

6.1.41. Take

The Take operator yields the first *n* elements of a sequence. If *n* <= 0 the result is an empty sequence; if *n* >= sequence's length the result is the sequence itself.

```
int[] ints  = { 1, 2, 3, 4, 5, 6 };
var   query = from ...;

var first3 = ints.Take(3);
var top10  = query.Take(10);

foreach(var x in first3)  // 1, 2, 3
  Console.WriteLine(x);
foreach(var r in top10)
  Console.WriteLine(r);
```

The sequence can be of any type T.

See also: Skip.

6.1.42. TakeWhile

Given a function F and a sequence S, the TakeWhile operator yields the first n elements of S where F returns true. Two versions of F are supported: accepting an element, or accepting an element along with its 0-based index in the sequence.

```
int[] ints  = { 1, 2, 3, 4, 5, 6 };
var   query = from ...;

var all    = ints.TakeWhile(x => true);
var none   = ints.TakeWhile(x => false);
var first3 = ints.TakeWhile( (x,i) => i < 3 );
var firstN = query.TakeWhile(result => ...);

foreach(var x in all)      // 1, 2, 3, 4, 5, 6
  Console.WriteLine(x);
foreach(var x in none)     // <none>
  Console.WriteLine(x);
foreach(var x in first3)   // 1, 2, 3
  Console.WriteLine(x);
foreach(var r in firstN)
  Console.WriteLine(r);
```

The sequence can be of any type T.

See also: SkipWhile and Where.

6.1.43. ThenBy

The ThenBy operator takes an ordered sequence and yields a secondary, ascending ordering; the keys used for the secondary ordering are yielded by a function applied to each element. Ordered sequences are typically produced by OrderBy or OrderByDescending, but can also be produced by ThenBy and ThenByDescending to yield additional sub-orderings.

```
Doctors doctors = new Doctors();

var docs = doctors.OrderBy(doc => doc.City).ThenBy(doc => doc.Initials);

foreach(var doc in docs)
  Console.WriteLine("{0}: {1}", doc.City, doc.Initials);
```

A second version allows you to provide a comparer of type *IComparer* for comparing keys and thus controlling the ordering, e.g., when you want to compare strings in a case-insensitive manner:

```
var docs = doctors.OrderBy(doc => doc.City)
                  .ThenBy(doc => doc.Initials,
                          StringComparer.CurrentCultureIgnoreCase);
```

Use of an *orderby* clause in a query expression with subsequent keys translates into application of the ThenBy operator. For example, the following are equivalent:

```
var docs = doctors.OrderBy(doc => doc.City).ThenBy(doc => doc.Initials);

var docs = from doc in doctors
           orderby doc.City, doc.Initials
           select doc;
```

See also: OrderBy, OrderByDescending, and ThenByDescending.

6.1.44. ThenByDescending

The ThenByDescending operator takes an ordered sequence and yields a secondary, descending ordering; the keys used for the secondary ordering are yielded by a function applied to each element. Ordered sequences are typically produced by OrderBy or OrderByDescending, but can also be produced by ThenBy and ThenByDescending to yield additional sub-orderings.

```
Doctors doctors = new Doctors();

var docs = doctors.OrderBy(doc => doc.City).ThenByDescending(doc => doc.Initials);

foreach(var doc in docs)
  Console.WriteLine("{0}: {1}", doc.City, doc.Initials);
```

A second version allows you to provide a comparer of type *IComparer* for comparing keys and thus controlling the ordering, e.g., when you want to compare strings in a case-insensitive manner:

```
var docs = doctors.OrderBy(doc => doc.City)
                  .ThenByDescending(doc => doc.Initials,
                                    StringComparer.CurrentCultureIgnoreCase);
```

Use of an *orderby* clause in a query expression with subsequent keys and the *descending* keyword translates into application of the ThenBy operator. For example, the following are equivalent:

```
var docs = doctors.OrderBy(doc => doc.City).ThenByDescending(doc => doc.Initials);

var docs = from doc in doctors
           orderby doc.City, doc.Initials descending
           select doc;
```

See also: OrderBy, OrderByDescending, and ThenBy.

6.1.45. ToArray

The ToArray operator iterates across a sequence of values, yielding an array containing these values. For example, doctors living in Chicago:

```
Doctors doctors = new Doctors();

var query = from doc in doctors
            where doc.City == "Chicago"
            select doc;

Doctor[] chicago = query.ToArray();
```

Since queries are lazily evaluated, ToArray is commonly used to execute a query and capture the results in a simple data structure.

See also: ToDictionary, ToList, ToLookup, and ToSequence.

6.1.46. ToDictionary

The ToDictionary operator iterates across a sequence of values, yielding a *Dictionary<K, V>* of (key, value) pairs. Each key must be unique, resulting in a one-to-one mapping of key to value. For example, storing doctors by their initials:

```
Doctors doctors = new Doctors();

var query = from doc in doctors
            where doc.City == "Chicago"
            select doc;

Dictionary<string, Doctor> chicago = query.ToDictionary(doc => doc.Initials);

foreach(var pair in chicago)
  Console.WriteLine("{0}: {1}", pair.Key, pair.Value.PagerNumber);
```

Since queries are lazily evaluated, ToDictionary is commonly used to execute a query and capture the results in a data structure that supports efficient lookup by key. For example, finding a doctor via their initials:

```
Doctor mbl = chicago["mbl"];
```

Another version provides control over exactly what values are stored in the data structure, e.g., only the doctor's email address:

```
Dictionary<string, string> emails = doctors.ToDictionary(doc => doc.Initials,
                                                         doc => doc.EmailAddress);
```

Finally, additional versions of ToDictionary allow you to provide a comparer of type *IEqualityComparer* for comparing keys.

See also: ToArray, ToList, ToLookup, and ToSequence.

6.1.47. ToList

The ToList operator iterates across a sequence of values, yielding a *List<T>* containing these values. For example, doctors living in Chicago:

```
Doctors doctors = new Doctors();

var query = from doc in doctors
            where doc.City == "Chicago"
            select doc;

List<Doctor> chicago = query.ToList();
```

Since queries are lazily evaluated, ToList is commonly used to execute a query and capture the results in a flexible data structure.

See also: ToArray, ToDictionary, ToLookup, and ToSequence.

6.1.48. ToLookup

The ToLookup operator iterates across a sequence of values, yielding a *Lookup<K, V>* of (key, value) pairs. The keys do not need to be unique; values with the same key form a collection under that key, resulting in a one-to-many mapping of key to values. For example, grouping doctors by city:

```
Doctors doctors = new Doctors();

var query = from doc in doctors
            select doc;
```

```
Lookup<string, Doctor> cities = query.ToLookup(doc => doc.City);

foreach(var pair in cities)
{
  Console.WriteLine("{0}:", pair.Key);  // city
  foreach(var doc in pair)              // doctors in that city
    Console.WriteLine("  {0}", doc.Initials);
}
```

Since queries are lazily evaluated, ToLookup is commonly used to execute a query and capture the results in a data structure that supports efficient lookup by key. For example, finding the doctors living in Chicago:

```
IEnumerable<Doctor> docs = cities["Chicago"];
```

Another version provides control over exactly what values are stored in the data structure, e.g., only the doctor's initials:

```
Lookup<string, string> cities = doctors.ToLookup(doc => doc.City,
                                                  doc => doc.Initials);
```

Finally, additional versions of ToLookup allow you to provide a comparer of type *IEqualityComparer* for comparing keys.

See also: ToArray, ToDictionary, ToList, and ToSequence.

6.1.49. ToSequence

The ToSequence operator casts a sequence as a sequence, thereby hiding any public members of the original sequencein particular those that might conflict or compete with the standard query operators. Use this operator when you want to gain access to the standard LINQ query operators.

For example, the following use of the *Count* query operator fails to compile:

```
Doctors doctors = new Doctors();
int count = doctors.Count(doc => doc.City == "Chicago");  // ERROR!
```

It fails because the Doctors class provides a conflicting Count property (inherited from *List<T>*). The ToSequence operator provides a quick solution:

```
int count1 = doctors.Count;
int count2 = doctors.ToSequence().Count();
int count3 = doctors.ToSequence().Count( doc => doc.City == "Chicago" );

Console.WriteLine(count1);   // 12
Console.WriteLine(count2);   // 12
Console.WriteLine(count3);   // 5
```

See also: ToArray, ToDictionary, ToList, and ToLookup.

6.1.50. Union

Given two sequences of elements S1 and S2, the Union operator returns the distinct elements of S1, followed by the distinct elements of S2 not in S1. In other words, Union computes the set union of S1 and S2.

```
int[] intsS1 = { 1, 2, 2, 3, 2, 3, 4, 6 };
int[] intsS2 = { 6, 1, 3, 5 };
var   query1 = from ...;
var   query2 = from ...;

var allInts    = intsS1.Union(intsS2);
var allResults = query1.Union(query2);

foreach(var x in allInts)  // 1, 2, 3, 4, 6, 5
```

```
  Console.WriteLine(x);
foreach(var r in allResults)
  Console.WriteLine(r);
```

The sequences may contain elements of any type *T*. This element type T must be the same at compile-time (e.g. `object`), but may differ at run-time:

```
object[] objects1 = { "abc", "def" };
object[] objects2 = { 1, 2, 3 };

var result = objects1.Union(objects2);
```

See also: Distinct, Except, and Intersect.

6.1.51. Where

The `Where` operator applies a function to a sequence of elements, yielding a sub-sequence of these elements.

```
int[] ints = {1, 2, 3, 4, 5, 6};
Doctors doctors = new Doctors();

var even    = ints.Where( x => x % 2 == 0);
var chicago = doctors.Where( doc => doc.City == "Chicago" );

foreach(var x in even)      // 2, 4, 6
  Console.WriteLine(x);
foreach(var doc in chicago)   // mbl, jl, kl, py, tm
  Console.WriteLine(doc.Initials);
```

Use of the *where* clause in a query expression translates into application of the `Where` operator. For example, the following statements are equivalent:

```
var chicago = doctors.Where( doc => doc.City == "Chicago" );

var chicago = from doc in doctors
              where doc.City == "Chicago"
              select doc;
```

A second version of the `Where` operator provides an element's 0-based index in the sequence along with the element itself, for example:

```
var last3 = ints.Where( (x,index) => (index >= ints.Length-3) ? true : false );

foreach (var x in last3)   // 4, 5, 6
  Console.WriteLine(x);
```

See also: Select.

NOTE

The application of LINQ is limited only by your imagination (and time). Here are a couple of the more interesting LINQ extensions floating around on the Web:

1. **BLinq** is a tool that generates an ASP.NET web application from a database schema. The web site supports the display, creation, and manipulation of the data in the database. LINQ to SQL is used for all database access, making the generated code easier to understand and modify. For more information see http://www.asp.net/sandbox/app_blinq.aspx?tabid=62.
2. **LINQ to Amazon** extends LINQ to support the querying of Amazon.com. See http://weblogs.asp.net/fmarguerie/archive/2006/06/26/Introducing-Linq-to-Amazon.aspx (http://weblogs.asp.net/fmarguerie/archive/2006/06/26/Introducing-Linq-to-Amazon.aspx:).

4.1. Lambda Expressions

From the developer's perspective, *lambda expressions* in C# 3.0 are a straightforward simplification of anonymous methods. Whereas an anonymous method is an unnamed block of code, a lambda expression is an unnamed expression that evaluates to a single value. Given a value x and an expression f(x) to evaluate, the corresponding lambda expression is written

```
x => f(x)
```

For example, in C# 3.0 the previous call to FindAll with an anonymous method is significantly shortened by using a lambda expression:

```
List<Doctor> inchicago = doctors.FindAll(x => x.City == "Chicago");
```

In this case, x is a doctor, and f(x) is the expression x.City == "Chicago", which evaluates to true or false. For readability, let's substitute d for x:

```
List<Doctor> inchicago = doctors.FindAll(d => d.City == "Chicago");
```

Notice that this lambda expression matches the parameter and body of the anonymous method we saw earlier, minus the syntactic extras like {}.

Lambda expressions are equivalent to anonymous methods that return a value when invoked, and are thus interchangeable. In fact, the compiler translates lambda expressions into delegate-based code, exactly as it does for anonymous methods.

PREV NEXT

4.6. Lazy Evaluation

Interestingly, you may be a bit surprised by the behavior of queries in LINQ. Let's look at an example. First, consider the following Boolean function that as a side-effect outputs the doctor's initials:

```
public static bool dump(Doctor d)
{
  System.Console.WriteLine(d.Initials);
  return true;
}
```

Now let's use dump in a simple query that will end up selecting all the doctors because the function always returns true:

```
var query = from d in doctors
            where dump(d)
            select d;
```

Here's the million dollar question: what does this query output? [*Nothing!*]

It turns out that the declaration of a query expression does just thatdeclares a query, but does *not* evaluate it. Evaluation in most cases is delayed until the results of the query are actually requested, an approach known as *lazy evaluation*. For example, the following statement outputs the initials of the first doctor:

```
query.GetEnumerator().MoveNext();  // outputs ==> "mbl"
```

By "moving" to the first element in the result set, we trigger a call to dump based on the first doctor, which outputs the doctor's initials. Since dump returns true, this doctor satisfies the where condition, the doctor is considered part of the result set, and MoveNext returns because it has successfully moved to the first element in the result set. Hence the initials of the first doctor, and only the first doctor, are output. [*What do you think happens if dump returns false for the first doctor?*]

To output the initials of all the doctors, we iterate across the entire result:

```
foreach (var result in query)  // "mbl", "jl", "ch", ...
  ;
```

Again, it's sufficient to simply request each result in order to trigger evaluation (and in this case the output of the doctor's initials), we do not have to access the resulting value.

You are probably wondering how lazy evaluation is supported in C# 3.0. In fact, the support was introduced in C# 2.0 via the yield construct. For example, here's the complete implementation of the standard LINQ query operator Where that we discussed earlier:

```
public static class Sequence
{
       public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
                                             Func<T, bool> predicate)
       {
         foreach (T element in source)
           if (predicate(element))
             yield return element;
       }
}
```

The *yield return* pattern produces the next value of the iteration and then returns. In response to yield, the C# compiler generates the necessary code (essentially a nested iterator class) to implement IEnumerable and enable iteration to continue where it left off.

NOTE

To learn more about how things work and what the C# compiler is doing, I highly recommend Lutz Roeder's *Reflector* tool as a way to reverse-engineer your compiled code and see what's going on: http://www.aisto.com/roeder/dotnet.

One of the implications of LINQ's lazy evaluation is that query expressions are re-evaluated each time they are processed. In other words, the results of a query are not stored or cached in a collection, but lazily evaluated and returned on each request. The advantage is that changes in the data are automatically reflected in the next processing cycle:

```
foreach (var result in query)    // outputs initials for all doctors
  ;

doctors.Add( new Doctor("xyz", ...) );

foreach (var result in query)    // output now includes new doctor "xyz"
  ;
```

Another advantage is that complex queries can be constructed from simpler ones, and execution is delayed until the final query is ready (and possibly optimized!).

If you want to force immediate query evaluation and cache the results, the simplest approach is to call the ToArray or ToList methods on the query. For example, the following code fragment outputs the initials of all doctors *three* consecutive times:

```
var cache1 = query.ToArray();  // evaluate query & cache results as an array

doctors.Add( new Doctor("xyz", ...) );

System.Console.WriteLine("###");
var cache2 = query.ToList();   // evaluate query again & cache results as a list

System.Console.WriteLine("###");
System.Console.WriteLine( cache1.GetType() );
System.Console.WriteLine( cache2.GetType() );

foreach (var result in cache1)  // output results of initial query:
  System.Console.WriteLine(result.Initials);

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

The second cache contains the new doctor, but the first does not. Here's the output:

```
mbl
jl
.
.
ks
###
mbl
jl
.
.
ks
xyz
###
Doctor[]
System.Collections.Generic.List`1[Doctor]
mbl
jl
.
.
ks
###
```

Most (but not all) of the standard LINQ query operators are lazily evaluated. The exception are operators that return a single (*scalar*) value, such as Min and Max, which might as well produce the value instead of a collection containing that value:

```
int minPager = doctors.Min(d => d.PagerNumber);
int maxPager = doctors.Max(d => d.PagerNumber);

System.Console.WriteLine("Pager range: {0}..{1}", minPager, maxPager);
```

The section on standard LINQ query operators will make note of which operators are lazily evaluated, and which are not. But first, let's take a deeper look at what LINQ offers.

PREV NEXT

Chapter 1. Learn LINQ and the C# 3.0 Features That Support It

Imagine writing SQL-like queries entirely in C#, with IntelliSense and compile-time type checking. Imagine queries as flexible as dynamically-generated SQL, yet as secure and efficient as calling stored procedures. Now take these same queries and execute them against an XML document or against a collection of data objects. While Microsoft isn't quite there yet, this is the goal of the LINQ project.

Language Integrated Query (LINQ) is a C# 3.0 API centered on data access. Developers focused on data access will be able to leverage the LINQ API to interoperate with a variety of data sources and vendors in a consistent, object-oriented fashion. The LINQ API is also extensible, as demonstrated by Microsoft's most important components of LINQ, LINQ to SQL and LINQ to XML. The former provides LINQ-like manipulation of relational databases and the latter of XML documents. You can expect future componentsboth from Microsoft and developers like yourselfto extend LINQ in new and interesting ways.

The vast majority of LINQ is made possible by language extensions in C# 3.0 and VB 9.0, which will appear in the upcoming 3.0 release of .NET. However, LINQ will also require a new version of the .NET Framework, which will follow the upcoming 3.0 release. We shall tentatively refer to this version as .NET 3.5. The implication is that developers will need to redeploy on .NET 3.5 to use LINQ.

NOTE

LINQ is still in development, and will be released in an upcoming version of .NET (3.5?). In the meantime, you can experiment with LINQ by downloading the latest CTP for Visual Studio 2005 here: http://msdn.microsoft.com/data/ref/linq.

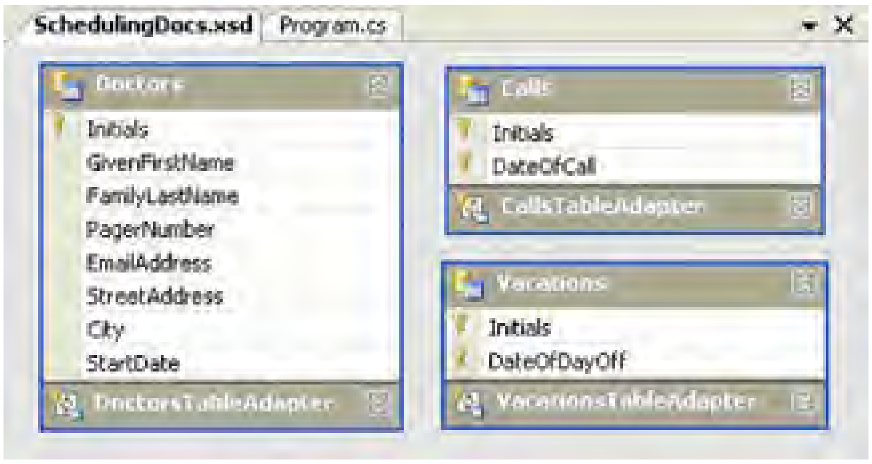
The material for this Short Cut is based on the May 2006 CTP of LINQ. Since LINQ is beta software, I encourage you to install Visual Studio 2005 and LINQ in a virtual environment, such as one created with Virtual PC.

The source code accompanying this Short Cut is available at either of the following URLs: http://www.oreilly.com/catalog/language1 or http://pluralsight.com/drjoe/pdfs/pdfs.aspx. To get started yourself, install the latest LINQ CTP, startup Visual Studio 2005, expand the project types for C#, and select "LINQ Preview." To see lots of examples of LINQ in action, try this site: http://msdn.microsoft.com/vcsharp/future/linqsamples.

5.1. LINQ to DataSets

LINQ supports the querying of both typed and untyped DataSets. Expanding on our theme of a hospital scheduling application, suppose we have a namespace DataSets with a typed dataset SchedulingDocs containing three tables: *Doctors*, *Calls*, and *Vacations*. The Doctors table contains one record for each doctor, the Calls table keeps track of which doctor is on call each day, and the Vacations table makes note of vacation requests. The DataSet is summarized in Figure 2.

Figure 5-1. Tables in the *SchedulingDocs* typed DataSet



Let's assume an instance of SchedulingDocs has been created and filled:

```
DataSets.SchedulingDocs ds = new DataSets.SchedulingDocs();  // create dataset
.
.  // open connection to a database and fill each table?
.
```

To find all the doctors living within Chicago, the query is exactly as we've seen before:

```
var chicago = from d in ds.Doctors
              where d.City == "Chicago"
              select d;
```

In this case ds.Doctors denotes a DataTable object, and d represents a DataRow object. Since the DataSet is typed, these objects are strongly-typed as DoctorsTable and DoctorsRow, respectively.

NOTE

For completeness, here's the code to create and fill an instance of SchedulingDocs, declared within the DataSets namespace:

```
using System.Data.Common;
using DataSets;

SchedulingDocsTableAdapters.DoctorsTableAdapter    doctorsAdapter;
SchedulingDocsTableAdapters.CallsTableAdapter      callsAdapter;
SchedulingDocsTableAdapters.VacationsTableAdapter  vacationsAdapter;
SchedulingDocs       ds;
DbProviderFactory  dbFactory;

dbFactory        = DbProviderFactories.GetFactory(providerInfo);
ds               = new SchedulingDocs();
doctorsAdapter   = new SchedulingDocsTableAdapters.DoctorsTableAdapter();
```

```
callsAdapter     = new SchedulingDocsTableAdapters.CallsTableAdapter();
vacationsAdapter = new SchedulingDocsTableAdapters.VacationsTableAdapter();

using (DbConnection dbConn = dbFactory.CreateConnection())
{
  dbConn.ConnectionString = connectionInfo;
  dbConn.Open();

  doctorsAdapter.Fill(ds.Doctors);
  callsAdapter.Fill(ds.Calls);
  vacationsAdapter.Fill(ds.Vacations);
} //dbConn.Close();
```

LINQ supports the notion of joins, including inner and outer joins. For example, let's join the Doctors and Calls tables to see which doctors are scheduled in the month of October 2006:

```
var oct2006 = (
                from d in ds.Doctors
                join c in ds.Calls
                on d.Initials equals c.Initials
                where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                      c.DateOfCall <= new DateTime(2006, 10, 31)
                orderby d.Initials
                select d.Initials
               )
              .Distinct();
```

This query expression uses a number of standard LINQ query operators, including `Join`, `OrderBy`, and `Distinct`; `Join` implements an *inner equijoin*.

LINQ is not limited to yielding tabular data, but will produce hierarchical results as appropriate. For example, suppose we want to know not only which doctors are on call in October 2006, but also the dates. In this case, we join the Doctors and Calls tables, now grouping the results:

```
var oct2006 = from d in ds.Doctors
               join c in ds.Calls
               on d.Initials equals c.Initials
               where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                     c.DateOfCall <= new DateTime(2006, 10, 31)
               group c by d.Initials into g
               select g;
```

Notice we group the calls ("c") on a per doctor basis ("d"). For each scheduled doctor, this yields an enumerable collection of calls. Here's how we process the query:

```
            foreach (var group in oct2006)
{
  System.Console.WriteLine("{0}: ", group.Key);
  foreach (var call in group)
     System.Console.WriteLine("  {0}", call.DateOfCall.ToString("dd MMM yyyy"));
  System.Console.WriteLine("  calls = {0}", group.Count());
}
```

The hierarchical output appears as follows:

```
jl:
  02 Oct 2006
  calls = 1
mbl:
  01 Oct 2006
  31 Oct 2006
  calls = 2
.
.
.
```

How about we re-order the results by those working the most, and select just the data we need in the result set:

```
var oct2006 = from d in ds.Doctors
              join c in ds.Calls
              on d.Initials equals c.Initials
              where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                    c.DateOfCall <= new DateTime(2006, 10, 31)
              group c by d.Initials into g
              orderby g.Count() descending
              select new { Initials = g.Key,
                           Count = g.Count(),
                           Dates = from c in g
                                   select c.DateOfCall };

foreach (var result in oct2006)
{
  System.Console.WriteLine("{0}:", result.Initials);
  foreach (var date in result.Dates)
     System.Console.WriteLine("  {0}", date.ToString("dd MMM yyyy"));
  System.Console.WriteLine("  calls = {0}", result.Count);
}
```

By projecting just the needed data, we can do things like data-bind the result for ease of display. The trade-off is that this *potentially* requires another set of custom objects to be instantiated. However, keep in mind that the design of LINQ enables the query operators to optimize away unnecessary object creation, much like compilers routinely eliminate unneeded code. This is especially true when applying LINQ in other situations, such as against a database (i.e., "*LINQ to SQL*").

PREV NEXT

5.5. LINQ to IEnumerable

One of the elegant design aspects of LINQ is that queries can be executed against any enumerable data source. If an object implements IEnumerable, then LINQ can access the data behind that object. For example, suppose we need to search the current user's *My Documents* folder (and sub-folders) for all non-system files modified in the last hour. Using LINQ we do this as follows:

```
using SIO = System.IO;

string[] files;
string   mydocs;

mydocs = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
files  = SIO.Directory.GetFiles(mydocs, "*.*", SIO.SearchOption.AllDirectories);

var query = from file in files
            let lasthour = DateTime.Now.Subtract(new TimeSpan(0, 1, 0, 0))
            where SIO.File.GetLastAccessTime(file) >= lasthour &&
                (SIO.File.GetAttributes(file) & SIO.FileAttributes.System) == 0
            select file;
```

Notice the presence of the let statement, which allows for the definition of values local to the query; let is used to improve readability and efficiency by factoring out common operations.

This example should not be very surprising, since what we are really doing is iterating across an array of filenames ("*files*"), not the file system itself. But this is a design artifact of the .NET Framework, not a limitation of LINQ. A similar example is the searching of a file, which is easily done in LINQ by iterating across the lines of the file:

```
string filename = ...;  // file to search

var lines = from line in SIO.File.ReadAllLines(filename)
            where line.Contains("class")
            select line;
```

In this example, we are reading all the lines into an array and then searching the array to select those lines containing the character sequence "class."

Need to search the Windows event log? An event log is a collection of EventLogEntry objects, and is accessed in .NET by creating an instance of the EventLog class. For example, here's how we gain access to the *Application* event log on the current machine:

```
using SD = System.Diagnostics;

SD.EventLog applog = new SD.EventLog("Application", ".");
```

Suppose we need to find all events logged by our application for a particular user. This is easily expressed as a LINQ query:

```
string appname  = ...;   // name of our application, e.g. "SchedulingApp"
string username = ...;   // login name for user, e.g. "DOMAIN\\hummel"

var entries = from entry in applog.Entries
              where entry.Source == appname &&
                    entry.UserName == username
              orderby entry.TimeWritten descending
              select entry;
```

Interestingly, while this query makes perfect sense, it does *not* compile. The issue is that EnTRies is a pre-2.0 collection, which means it implements IEnumerable and not IEnumerable<T>. Since IEnumerable is defined in terms of object and not a specific type T, the C# type inference engine cannot infer the type of objects the query expression is working with. The designers of LINQ provide an easy solution in the form of the standard query operator Cast. The Cast operator wraps a generic enumerable object with a type-specific one suitable for LINQ:

```
var entries = from entry in applog.Entries.Cast<SD.EventLogEntry>()
```

```
          where entry.Source == appname &&
                entry.UserName == username
          orderby entry.TimeWritten descending
          select entry;
```

The `Cast` operator allows LINQ to support .NET 1.*x* collections.

As a final example, let's apply LINQ to the world of Visual Studio Tools for Office (VSTO). To search a user's Outlook contacts, the basic query is as follows:

```
Outlook.MAPIFolder folder = this.ActiveExplorer().Session.
  GetDefaultFolder( Outlook.OlDefaultFolders.olFolderContacts );

var contacts = from contact in folder.Items.OfType<Outlook.ContactItem>()
               where ...  // search criteria, e.g. contact.Email1Address != null
               select contact;
```

We take advantage of LINQ's `OfType` query operator, which (a) wraps pre-2.0 collections and (b) filters the collection to return only those objects of the desired type. Here's a query to collect all distinct email addresses from a user's Outlook contacts:

```
var emails = (
              from contact in folder.Items.OfType<Outlook.ContactItem>()
              where contact.Email1Address != null
              select contact.Email1Address
             )
             .Distinct();
```

Finally, given collections of email addresses from different folders or users, we can use LINQ to perform various set operations over these collections:

```
var union        = emails.Union(emails2);
var intersection = emails.Intersect(emails2);
var difference   = emails.Except(emails2);
```

PREV NEXT

5.2. LINQ to SQL

Instead of executing our queries against a DataSet, suppose we want to execute against the database directly? No problem. With LINQ to SQL, we change only the target of our query:

```
        Databases.SchedulingDocs db = new Databases.SchedulingDocs(connectionInfo);

var oct2006 = ( // find all doctors scheduled for October 2006:
                from d in db.Doctors
                join c in db.Calls
                on d.Initials equals c.Initials
                where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                      c.DateOfCall <= new DateTime(2006, 10, 31)
                orderby d.Initials
                select d.Initials
               )
               .Distinct();
```

In this case, SchedulingDocs is a class denoting a SQL Server 2005 database named *SchedulingDocs*. This class, and its associated entity classesD o ctors, Calls, and Vacationswere automatically generated by LINQ's *SQLMetal* tool to represent the database and its tables. As you would expect, the query expression is lazily evaluated, waiting for the query to be consumed:

```
        foreach (var initials in oct2006)  // execute the query:
  System.Console.WriteLine("{0}", initials);
```

The query is now translated into parameterized SQL, sent to the database for execution, and the result set produced. How efficient is the generated SQL? In this case (and the May CTP of LINQ), a single select statement is executed against the database:

```
SELECT DISTINCT [t0].[Initials]
  FROM [Doctors] AS [t0], [Calls] AS [t1]
  WHERE ([t1].[DateOfCall] >= @p0) AND
        ([t1].[DateOfCall] <= @p1) AND
        ([t0].[Initials] = [t1].[Initials])
  ORDER BY [t0].[Initials]
```

NOTE

In the May CTP of LINQ, SQLMetal is provided as a command-line tool. To run, first open a command window and cd to the install directory for LINQ (most likely *C:\Program Files\LINQ Preview\Bin*). Now ask SQLMetal to read the metadata from your database and generate the necessary class files. For the SQL Server 2005 database named SchedulingDocs, the command is:

```
  C:\...\Bin>sqlmetal /server:. /database:SchedulingDocs
/language:csharp /code:SchedulingDocs.cs /namespace:Databases
```

This will generate a source code file *SchedulingDocs.cs* in the current directory.

Let's look at a more complex query that computes the number of calls for every doctor in the month of October 2006. Recall that an inner join produces results for only those doctors that are working:

```
var oct2006 = from d in db.Doctors
               join c in db.Calls
               on d.Initials equals c.Initials
               where c.DateOfCall >= new DateTime(2006, 10, 1) &&
                     c.DateOfCall <= new DateTime(2006, 10, 31)
               group c by d.Initials into g
               select new { Initials = g.Key, Count = g.Count() };
```

An *outer join* is needed to capture the results for all doctors, whether scheduled or not. Outer joins are based on LINQ's *join ... into* syntax:

```
var allOct2006 = from d1 in db.Doctors      // join all doctors
                 join d2 in oct2006         // with those working in Oct 2006
                 on d1.Initials equals d2.Initials
                 into j
                 from r in j.DefaultIfEmpty()
                 select new { Initials = d1.Initials,
                              Count = (r == null ? 0 : r.Count) };
```

This left outer join produces a result set "*into*" j, which is then enumerated across using the sub-expression "*from r in ...*". If a given doctor is working, then r is the joined result; if the doctor is not working then the result is empty, in which case j.DefaultIfEmpty() returns null. For each doctor, we then project their initials and the number of calls they are workingeither 0 or the count from the inner join. Iterating across the query:

```
foreach (var result in allOct2006)
    System.Console.WriteLine("{0}: {1}", result.Initials, result.Count);
```

Yields:

```
ay: 7
bb: 0
ch: 3
.
.
.
```

When the query is executed, the following SQL is sent to the database (this is a test intended for the SQL wizards in the audience):

```
SELECT [t7].[Initials], [t7].[value] AS [Count]
FROM (
    SELECT
        (CASE
            WHEN [t4].[test] IS NULL THEN 0
              ELSE (
                SELECT COUNT(*)
                FROM [Doctors] AS [t5], [Calls] AS [t6]
                WHERE ([t4].[Initials] = [t5].[Initials]) AND
                      ([t6].[DateOfCall] >= @p0) AND
                      ([t6].[DateOfCall] <= @p1) AND
                      ([t5].[Initials] = [t6].[Initials])
              )
        END) AS [value], [t0].[Initials]
    FROM [Doctors] AS [t0]
    LEFT OUTER JOIN (
        SELECT 1 AS [test], [t3].[Initials]
        FROM (
            SELECT [t1].[Initials]
            FROM [Doctors] AS [t1], [Calls] AS [t2]
            WHERE ([t2].[DateOfCall] >= @p0) AND
                  ([t2].[DateOfCall] <= @p1) AND
                  ([t1].[Initials] = [t2].[Initials])
            GROUP BY [t1].[Initials]
        ) AS [t3]
    ) AS [t4] ON [t0].[Initials] = [t4].[Initials]
) AS [t7]
```

NOTE

In all fairness, it should be noted that the focus of this Short Cut is LINQ, and not particularly LINQ to SQL. For this reason, the picture painted of LINQ to SQL is quite superficial. For example, in LINQ to SQL, queries are translated to SQL and executed with SQL semantics. In comparison, most other LINQ queries are directly executed by .NET Framework objects with CLR semantics. Likewise, LINQ to SQL handles changes to the data quite differently than other flavors of LINQ. For more details, we encourage you to read the forthcoming Part 2 of this Short Cut series on LINQ (expected fall 2006), which focuses exclusively on LINQ to SQL. Watch for an announcement at http://oreilly.com

5.4. LINQ to XML

From its beginnings, LINQ was designed to manipulate XML data as easily as it manipulates relational data. LINQ to XML represents a new API for XML-based development, equivalent in power to *XPath* and *XQuery* yet far simpler for most developers to use.

For example, let's assume the data source for our hospital scheduling application is an XML document stored in the file *SchedulingDocs.xml*. Here's the basic structure of the document:

```
<?xml version="1.0" standalone="yes"?>
<SchedulingDocs>
  <Calls>
    <Call>
      <Initials>mbl</Initials>
      <DateOfCall>2006-10-01T00:00:00-05:00</DateOfCall>
    </Call>
    .
    .
    .
  </Calls>
  <Doctors>
    <Doctor>
      <Initials>ay</Initials>
      <GivenFirstName>Amy</GivenFirstName>
      <FamilyLastName>Yang</FamilyLastName>
      <PagerNumber>53300</PagerNumber>
      <EmailAddress>ayang@uhospital.edu</EmailAddress>
      <StreetAddress>1400 Ridge Ave.</StreetAddress>
      <City>Evanston</City>
    </Doctor>
    .
    .
    .
  </Doctors>
  <Vacations>
    <Vacation>
      <Initials>jl</Initials>
      <DateOfDayOff>2006-10-03T00:00:00-05:00</DateOfDayOff>
    </Vacation>
    .
    .
    .
  </Vacations>
</SchedulingDocs>
```

Using LINQ, we load this document as follows:

```
import System.Xml.XLinq;  // LINQ to XML

XElement root, calls, doctors, vacations;

root = XElement.Load("SchedulingDocs.xml");

calls     = root.Element("Calls");
doctors   = root.Element("Doctors");
vacations = root.Element("Vacations");
```

We now have access to the three main elements of the XML document: *calls*, *doctors*, and *vacations*. To select all the doctors, it's a simple query expression:

```
var docs = from doc in doctors.Elements()
           select doc;
```

And to find just those doctors living in Chicago:

```
var chicago = from doc in doctors.Elements()
```

```
              where doc.Element("City").Value == "Chicago"
              orderby doc.Element("Initials").Value
              select doc;
```

As you can see, querying XML documents with LINQ is conceptually the same as that of relational databases, DataSets, and other objects. The difference is that the structure of the XML document must be taken into account, e.g., in this case the document's hierarchical design and its use of elements over attributes.

An important aspect of LINQ is the ability to easily transform data into other formats. In the world of XML, transformation is commonplace given the need to create XML documents as well as translate from one schema to another. For example, suppose we need to produce a new XML document containing just the names of the doctors, with their initials as an attribute:

```
<?xml version="1.0" standalone="yes"?>
<Doctors>
  <Doctor Initials="bb">Boswell, Bryan</Doctor>
  <Doctor Initials="lg">Goldstein, Luther</Doctor>
  .
  .
  .
</Doctors>
```

This document is easily produced by the following query, which simply projects new *XElements*:

```
var docs = from doc in doctors.Elements()
           orderby doc.Element("FamilyLastName").Value,
                   doc.Element("GivenFirstName").Value
           select new XElement("Doctor",
                    new XAttribute("Initials", doc.Element("Initials").Value),
                    doc.Element("FamilyLastName").Value +
                    ", " +
                    doc.Element("GivenFirstName").Value);

XElement newroot = new XElement("Doctors", docs);
```

The last statement creates the root element `<Doctors>`, using the query to generate the `<Doctor>` sub-elements.

Finally, here's a real-world example of translating a text-based IIS logfile into an XML document. This example comes from a series of posts to the MSDN LINQ Project General Forum, "Transforming a TXT file into XML with LINQ to XML," http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=574140&SiteID=1. The logfile contains 0 or more lines of the form:

```
Time  IP-address  Method  URI  Status
```

For example:

```
#Software: Microsoft Internet Information Services 5.1
#Version: 1.0
#Date: 2006-06-23 12:37:18
#Fields: time c-ip cs-method cs-uri-stem sc-status
12:37:18 127.0.0.1 GET /cobrabca 404
12:37:25 127.0.0.1 GET /cobranca 401
.
.
.
```

Here's the LINQ query to produce an XML document from such a log:

```
var logIIS = new XElement("LogIIS",
               from line in File.ReadAllLines("file.log")
               where !line.StartsWith("#")
               let items = line.Split(' ')
               select new XElement("Entry",
                 new XElement("Time", items[0]),
                 new XElement("IP", items[1]),
                 new XElement("Url", items[3]),
                 new XElement("Status", items[4])
               )
```

```
        );
```

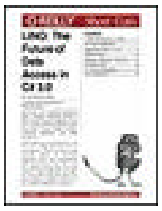
LINQ over a text file? The next section will discuss this, and other examples, in more detail.

NOTE

The general focus of this Short Cut precludes an in-depth treatment of LINQ to XML. For more details, we encourage you to read the forthcoming Part 3 of this Short Cut series (expected Q4 2006), which focuses exclusively on LINQ to XML. Watch for an announcement at http://oreilly.com

PREV NEXT

NEXT



LINQ: The Future of Data Access in C# 3.0

By Joe Hummel

....
Publisher: **O'Reilly**
Pub Date: **October 01, 2006**
Print ISBN-10: **0-596-52841-8**
Print ISBN-13: **978-0-59-652841-6**
Pages: **64**

Table of Contents

Overview

Language Integrated Query (LINQ) is Microsoft's new technology for powerful, general purpose data access. This technology provides a fully-integrated query language, available in both C# 3.0 and VB 9.0, for high-level data access against objects, relational databases, and XML documents. In this Short Cut you'll learn about LINQ and the proposed C# 3.0 extensions that support it. You'll also see how you can use LINQ and C# to accomplish a variety of tasks, from querying objects to accessing relational data and XML. Best of all, you'll be able to test the examples and run your own code using the latest LINQ CTP, available free from Microsoft. This Short Cut includes a complete reference to the standard LINQ query operators.

NEXT

4.4. Query Expressions

A LINQ query is called a *query expression*. Query expressions start with the keyword from, and are written using SQL-like *query operators* such as Select, Where, and OrderBy:

```
using System.Query;  // import standard LINQ query operators

// all doctors living in Chicago, sorted by last name, first name:
var chicago = from d in doctors
              where d.City == "Chicago"
              orderby d.FamilyLastName, d.GivenFirstName
              select d;

foreach (var r in chicago)
  System.Console.WriteLine("{0}, {1}", r.FamilyLastName, r.GivenFirstName);
```

By default, you must import the namespace System.Query to gain access to the standard LINQ query operators.

As noted earlier, type inference is used to make query expressions easier to write and consume:

```
      var chicago = from d in doctors ... select d;

foreach (var r in chicago) ... ;
```

But what exactly is a query expression? What type is inferred for the variable chicago above? Consider SQL. In SQL, a *select* query is a declarative statement that operates on one or more tables, producing a table. In LINQ, a query expression is a declarative expression operating on one or more IEnumerable objects, returning an IEnumerable object. Thus, a query expression is an expression of iteration across one or more objects, producing an object over which you iterate to collect the result. For example, let's be type-specific in our previous declaration:

```
      IEnumerable<Doctor> chicago = from d in doctors ... select d;
```

LINQ defines query expressions in terms of IEnumerable<T> to hide implementation details (preserving flexibility!) while conveying in a strongly-typed way the key concept that a query expression can be iterated across:

```
foreach (Doctor d in chicago) ... ;
```

Of course, type inference conveniently hides this level of detail without any loss of safety or performance.

Chapter 6. Standard LINQ Query Operators

LINQ provides a wide-range of query operators, many of which have been demonstrated in the previous sections. The purpose of this section is to succinctly summarize the complete set of LINQ query operators listed in Table 1. Recall that you must import the System.Query namespace to use these operators.

PREV NEXT

Chapter 4. Supporting LINQ in C# 3.0

Generics, delegates, and anonymous methods in C# 2.0 provide insight into how LINQ is supported in C# 3.0. Let's consider once again the problem of finding all doctors living within the Chicago city limits. As shown earlier, here's the obvious approach using foreach:

```
List<Doctor> inchicago = new List<Doctor>();
foreach (Doctor d in doctors)
  if (d.City == "Chicago")
    inchicago.Add(d);
```

A more elegant approach currently available in C# 2.0 is to take advantage of the collection's FindAll method, which can be passed a *delegate* to a function that determines whether the doctor resides in Chicago:

```
public delegate bool Predicate<T>(T obj);  // pre-defined in .NET 2.0

public bool IsInChicago(Doctor obj)  // notice how this matches delegate signature
{
  return obj.City == "Chicago";
}
.
.
.

List<Doctor> inchicago = doctors.FindAll(new Predicate<Doctor>(this.IsInChicago));
```

FindAll iterates through the collection, building a new collection containing those objects for which the delegate-invoked function returns true.

A more succinct version passes an *anonymous method* to FindAll:

```
List<Doctor> inchicago = doctors.FindAll(delegate(Doctor d)
                          {
                            return d.City == "Chicago";
                          } );
```

The {} denote the body of the anonymous method; notice these fall within the scope of the () in the call to FindAll. The signature of the anonymous methodin this case a Boolean function with a single Doctor argumentis type-checked by the compiler to ensure that it matches the definition of the argument to FindAll. The compiler then translates this version into an explicit delegate-based version, assigning a unique name to the underlying method:

```
private static bool b__0(Doctor d)
{
  return d.City == "Chicago";
}

List<Doctor> inchicago = doctors.FindAll( new Predicate<Doctor>(b__0) );
```

While this has nothing to do with LINQ per se, this approach of translating from one abstraction to another exemplifies how LINQ integrates into C# 3.0.

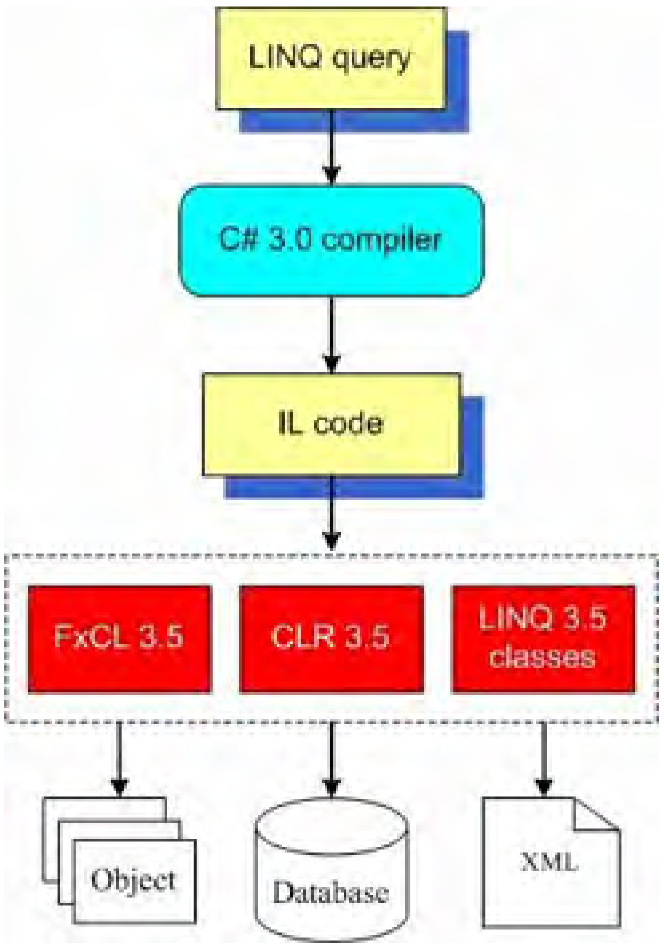
PREV NEXT

PREV NEXT

Chapter 3. The LINQ Architecture

In scanning the examples in the previous section, you probably noticed use of the var keyword, as well as the use of the new keyword without an accompanying class name. What you see are two new features of C# 3.0, namely type inference and anonymous types, added to the language in support of LINQ. Additional language features include query expressions, lambda expressions, expression trees, extension methods, and object initializers, all of which will be explained in the following sections.

Figure 3-1. LINQ_Architecture



LINQ's architecture is designed around a high-level, collection-like abstraction. This means that LINQ queries are written in C# 3.0 against the notion of an object collection, and then translated by the compiler into IL for execution against a particular data source (see Figure 1). This provides many benefits, including:

- Ability to target different data sources, e.g. databases and XML documents.
- Ability to use LINQ with existing .NET 1.* and 2.0 objects.
- Ability to extend LINQ to support new classes and technologies.

As an analogy, consider the foreach loop. The foreach is merely syntactic sugar for a while loop written against the IEnumerable and IEnumerator interfaces. However, foreach represents a convenient and powerful abstraction, allowing intuitive iteration across different types of collections, today and in the future. It's now hard to imagine C# or VB without foreach.

LINQ: The Future of Data Access in C# 3.0

By Joe Hummel

Publisher: **O'Reilly**
Pub Date: **October 01, 2006**
Print ISBN-10: **0-596-52841-8**
Print ISBN-13: **978-0-59-652841-6**
Pages: **64**

Table of Contents

4.2. Type Inference

Interestingly, there is one very subtle difference between lambda expressions and anonymous methods: the latter require type information, while the former do not. Returning to our example, notice that the lambda expression

```
d => d.City == "Chicago"
```

does not specify a type for d. Without a type, the compiler cannot translate the lambda expression into the equivalent anonymous method:

```
delegate(?????? d)  // what type is the argument d?
{
  return d.City == "Chicago";
}
```

To make this work, C# 3.0 is actually *inferring* the type of d in the lambda expression, based on contextual information. For example, since doctors is of type List<Doctor>, the compiler can prove that in the context of calling FindAll

```
doctors.FindAll(d => d.City == "Chicago")
```

d must be of type Doctor.

Type inference is used throughout C# 3.0 to make LINQ more convenient, without any loss of safety or performance. The idea of *type inference* is that the compiler infers the types of your variables based on context, not programmer-supplied declarations. The compiler does this while continuing to enforce strict, compile-time type checking.

As we saw earlier when introducing LINQ, developers can take advantage of type inference by using the var keyword when declaring local variables. For example, the declarations

```
var sum = 0;
var avg = 0.0;
var obj = new Doctor(...);
```

trigger the inference of int for sum, double for avg, and Doctor for obj. The initializer in the declaration is used to drive the inference engine, and is required. The following are thus illegal:

```
var obj2;          // ERROR: must have an initializer
var obj3 = null;   // ERROR: must have a specific type
```

Assuming the inference is successful, the inferred type becomes permanent and compilation proceeds as usual. Apparent misuses of the type are detected and reported by the compiler:

```
var obj4 = "hi!";
.
.
.
obj4.Close();  // Oops, wrong object! (ERROR: 'string' does not contain 'Close').
```

Do not confuse var with the concept of a VB variant (it's not), nor with the concept of var in dynamic languages like JavaScript (where var really means *object*). In these languages the variable's type can change, and so type checking is performed at runtimeincreased flexibility at the cost of safety. In C# 3.0 the type cannot change, and all type checking is done at compile-time. For example, if the inferred type is object (as for obj6 below), in C# 3.0 you end up with an object reference of very little functionality:

```
object obj5 = "hi";  // obj5 references the string "hi!", but type is object
var    obj6 = obj5;  // obj6 also references "hi!", with inferred type object
.
.
.
string s1 = obj6.ToUpper();  // ERROR: 'object' does not contain 'ToUpper'
```

While developers will find type inference useful in isolation, the real motivation is LINQ. Type inference is critical to the success of LINQ since queries can yield complex results. LINQ would be far less attractive if developers had to explicitly type all aspects of their queries. In fact, specifying a type is sometimes impossible, e.g., when projections select new patterns of data:

```
      var query = from d in doctors
      where d.City == "Chicago"
      select new { d.GivenFirstName, d.FamilyLastName };  // type?

foreach (var r in query)
  System.Console.WriteLine("{0}, {1}", r.FamilyLastName, r.GivenFirstName);
```

In these cases, typing is better left to the compiler.

PREV NEXT