CHAPTER 1

Hello LINQ

Listing 1-1. *Hello LINQ*

```
using System;
using System.Linq;

string[] greetings = {"hello world", "hello LINQ", "hello Apress"};

var items =
  from s in greetings
  where s.EndsWith("LINQ")
  select s;

foreach (var item in items)
  Console.WriteLine(item);
```

Note The code in Listing 1-1 was added to a project created with the console application template in Visual Studio 2008. If one is not already present, you should add a `using` directive for the `System.Linq` namespace.

Running the previous code by pressing Ctrl+F5 outputs the following data to the console window:

```
hello LINQ
```

A Paradigm Shift

Did you just feel *your* world shift? As a .NET developer, you should have. With the trivial programming example in Listing 1-1, you just ran what somewhat appears to be a Structured Query Language (SQL) query on an array of strings.[1] Check out that `where` clause. If it looks like I used the `EndsWith` method of a `string` object, it's because I did. You may be wondering what is with that variable type `var`? Is C# still performing static type checking? The answer is yes, it still statically checks types at compile time. What feature or features of C# are allowing all of this? The answer is Microsoft's Language Integrated Query, otherwise known as *LINQ*.

1. Most noticeably, the order is inverted from typical SQL. Additionally, there is the added "s in" portion of the query that provides a reference to the set of elements contained in the source, which in this case is the array of strings "hello world," "hello LINQ," and "hello Apress".

Query XML

While the example in Listing 1-1 is trivial, the example in Listing 1-2 may begin to indicate the potential power that LINQ puts into the hands of the .NET developer. It displays the ease with which one can interact with and query Extensible Markup Language (XML) data utilizing the LINQ to XML API. You should pay particular attention to how I construct the XML data into an object named `books` that I can programmatically interact with.

Listing 1-2. *A Simple XML Query Using LINQ to XML*

```
using System;
using System.Linq;
using System.Xml.Linq;

XElement books = XElement.Parse(
  @"<books>
      <book>
        <title>Pro LINQ: Language Integrated Query in C# 2008</title>
        <author>Joe Rattz</author>
      </book>
      <book>
        <title>Pro WF: Windows Workflow in .NET 3.0</title>
        <author>Bruce Bukovics</author>
      </book>
      <book>
        <title>Pro C# 2005 and the .NET 2.0 Platform, Third Edition</title>
        <author>Andrew Troelsen</author>
      </book>
    </books>");

var titles =
  from book in books.Elements("book")
  where (string) book.Element("author") == "Joe Rattz"
  select book.Element("title");

foreach(var title in titles)
  Console.WriteLine(title.Value);
```

■ **Note** The code in Listing 1-2 requires adding the `System.Xml.Linq.dll` assembly to the project references if it is not already added. Also notice that I added a `using` directive for the `System.Xml.Linq` namespace.

Running the previous code by pressing Ctrl+F5 outputs the following data to the console window:

```
Pro LINQ: Language Integrated Query in C# 2008
```

Did you notice how I parsed the XML data into an object of type `XElement`? Nowhere did I create an `XmlDocument`. Among the benefits of LINQ to XML are the extensions made to the XML API. Now instead of being `XmlDocument`-centric as the W3C Document Object Model (DOM) XML API requires, LINQ to XML allows the developer to interact at the element level using the `XElement` class.

■ **Note** In addition to query features, LINQ to XML provides a more powerful and easier to use interface for working with XML data.

Again notice that I utilized the same SQL-like syntax to query the XML data, as though it were a database.

Query a SQL Server Database

My next example shows how to use LINQ to SQL to query database[2] tables. In Listing 1-3, I query the standard Microsoft Northwind sample database.

Listing 1-3. *A Simple Database Query Using LINQ to SQL*

```
using System;
using System.Linq;
using System.Data.Linq;

using nwind;

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

var custs =
  from c in db.Customers
  where c.City == "Rio de Janeiro"
  select c;

foreach (var cust in custs)
  Console.WriteLine("{0}", cust.CompanyName);
```

■ **Note** The code in Listing 1-3 requires adding the `System.Data.Linq.dll` assembly to the project references if it is not already added. Also notice that I added a `using` directive for the `System.Data.Linq` namespace.

You can see that I added a `using` directive for the `nwind` namespace. For this example to work, you must use the SQLMetal command-line utility, or the Object Relational Designer, to generate entity classes for the targeted database, which in this example is the Microsoft Northwind sample database. See Chapter 12 to read how this is done with SQLMetal. The generated entity classes are created in the `nwind` namespace, which I specify when generating them. I then add the SQLMetal generated source module to my project and add the `using` directive for the `nwind` namespace.

■ **Note** You may need to change the connection string that is passed to the `Northwind` constructor in Listing 1-3 for the connection to be properly made. Read the section on `DataContext()` and `[Your]DataContext()` in Chapter 16 to see different ways to connect to the database.

2. At the present time, LINQ to SQL only supports SQL Server.

Running the previous code by pressing Ctrl+F5 outputs the following data to the console window:

```
Hanari Carnes
Que Delícia
Ricardo Adocicados
```

This simple example demonstrates querying the Customers table of the Northwind database for customers in Rio de Janeiro. While it may appear that there is nothing new or special going on here that I wouldn't already have with existing means, there are some significant differences. Most noticeably, this query is integrated into the language, and this means I get language-level support, which includes syntax checking and IntelliSense. Gone will be the days of writing a SQL query into a string and not detecting a syntax error until runtime. Want to make your `where` clause dependent on a field in the Customers table but you cannot remember the name of the field? IntelliSense will show the table's fields to you. Once you type in **c.** in the previous example, IntelliSense will display all the fields of the Customers table to you.

All of the previous queries use the *query expression* syntax. You will learn in Chapter 2 that there are two syntaxes available for LINQ queries, of which the query expression syntax is one. Of course, you may always use the *standard dot notation* syntax that you are accustomed to seeing in C# instead. This syntax is the normal `object.method()` invocation pattern you have always been using.

Introduction

As the Microsoft .NET platform, and its supporting languages C# and VB, have matured, it has become apparent that one of the more troublesome areas still remaining for developers is that of accessing data from different data sources. In particular, database access and XML manipulation is often cumbersome at best, and problematic in the worst cases.

The database problems are numerous. First, there is the issue that we cannot programmatically interact with a database at the native language level. This means syntax errors often go undetected until runtime. Incorrectly referenced database fields are not detected either. This can be disastrous, especially if this occurs during the execution of error-handling code. Nothing is more frustrating than having an entire error-handling mechanism fail because of syntactically invalid code that has never been tested. Sometimes this is unavoidable due to unanticipated error behavior. Having database code that is not validated at compile time can certainly lead to this problem.

A second problem is the nuisance caused by the differing data types utilized by a particular data domain, such as database or XML data types versus the native language in which the program is written. In particular, dates and times can be quite a hassle.

XML parsing, iterating, and manipulation can be quite tedious. Often an XML fragment is all that is desired, but due to the W3C DOM XML API, an `XmlDocument` must be created just to perform various operations on the XML fragment.

Rather than just add more classes and methods to address these deficiencies in a piecemeal fashion, the development team at Microsoft decided to go one step further by abstracting the fundamentals of data query from these particular data domains. The result was LINQ. LINQ is Microsoft's technology to provide a language-level support mechanism for querying data of all types. These types include in-memory arrays and collections, databases, XML documents, and more.

LINQ Is About Data Queries

For the most part, LINQ is all about queries, whether they are queries returning a set of matching objects, a single object, or a subset of fields from an object or set of objects. In LINQ, this returned set of objects is called a *sequence*. Most LINQ sequences are of type `IEnumerable<T>`, where `T` is the

data type of the objects stored in the sequence. For example, if you have a sequence of integers, they would be stored in a variable of type `IEnumerable<int>`. You will see that `IEnumerable<T>` runs rampant in LINQ. Many of the LINQ methods return an `IEnumerable<T>`.

In the previous examples, all of the queries actually return an `IEnumerable<T>` or a type that inherits from `IEnumerable<T>`. However, I use the `var` keyword for the sake of simplicity at this point, which is a new shorthand technique that I cover in Chapter 2. You will see that the examples will begin demonstrating that sequences are truly stored in variables implementing the `IEnumerable<T>` interface.

Components

Because LINQ is so powerful, you should expect to see a lot of systems and products become LINQ compatible. Virtually any data store would make a good candidate for supporting LINQ queries. This includes databases, Microsoft's Active Directory, the registry, the file system, an Excel file, and so on.

Microsoft has already identified the components in this section as necessary for LINQ. There is no doubt that there will be more to come.

LINQ to Objects

LINQ to Objects is the name given to the `IEnumerable<T>` API for the Standard Query Operators. It is LINQ to Objects that allows you to perform queries against arrays and in-memory data collections. Standard Query Operators are the static methods of the static `System.Linq.Enumerable` class that you use to create LINQ to Objects queries.

LINQ to XML

LINQ to XML is the name given to the LINQ API dedicated to working with XML. This interface was previously known as *XLinq* in older prereleases of LINQ. Not only has Microsoft added the necessary XML libraries to work with LINQ, it has addressed other deficiencies in the standard XML DOM, thereby making it easier than ever to work with XML. Gone are the days of having to create an `XmlDocument` just to work with a small piece of XML. To take advantage of LINQ to XML, you must have a reference to the `System.Xml.Linq.dll` assembly in your project and have a `using` directive such as the following:

```
using System.Xml.Linq;
```

LINQ to DataSet

LINQ to DataSet is the name given to the LINQ API for DataSets. Many developers have a lot of existing code relying on DataSets. Those who do will not be left behind, nor will they need to rewrite their code to take advantage of the power of LINQ.

LINQ to SQL

LINQ to SQL is the name given to the `IQueryable<T>` API that allows LINQ queries to work with Microsoft's SQL Server database. This interface was previously known as *DLinq* in older prereleases of LINQ. To take advantage of LINQ to SQL, you must have a reference to the `System.Data.Linq.dll` assembly in your project and have a `using` directive such as the following:

```
using System.Data.Linq;
```

LINQ to Entities

LINQ to Entities is an alternative LINQ API that is used to interface with a database. It decouples the entity object model from the physical database by injecting a logical mapping between the two. With

this decoupling comes increased power and flexibility, as well as complexity. Because LINQ to Entities appears to be outside the core LINQ framework, it is not covered in this book. However, if you find that you need more flexibility than LINQ to SQL permits, it would be worth considering as an alternative. Specifically, if you need looser coupling between your entity object model and database, entity objects comprised of data coming from multiple tables, or more flexibility in modeling your entity objects, LINQ to Entities may be your answer.

How to Obtain LINQ

Technically, there is no LINQ product to obtain. LINQ is just the project code name for the query feature being added to C# 3.0 and the .NET Framework 3.5, which will make their debut in the next version of Visual Studio, Visual Studio 2008.

To get up-to-date information on LINQ and Visual Studio 2008, visit `http://www.linqdev.com` and `http://apress.com/book/bookDisplay.html?bID=10241`.

LINQ Is Not Just for Queries

Maybe by definition you could say LINQ is just for queries because it stands for *Language Integrated Query*. But please don't think of it only in that context. Its power transcends mere data queries. I prefer to think of LINQ as a data iteration engine, but perhaps Microsoft didn't want a technology named DIE.

Have you ever called a method and it returned data back in some data structure that you then needed to convert to yet another data structure before you could pass it to another method? Let's say for example you call method A, and method A returns an array of type `string` that contains numeric values stored as strings. You then need to call method B, but method B requires an array of integers. You normally end up writing a loop to iterate through the array of strings and populate a newly constructed array of integers. What a nuisance. Allow me to point out the power of Microsoft's LINQ.

Let's pretend I have an array of strings that I received from some method A, as shown in Listing 1-4.

Listing 1-4. *Converting an Array of Strings to Integers*

```
string[] numbers = { "0042", "010", "9", "27" };
```

For this example, I'll just statically declare an array of strings. Now before I call method B, I need to convert the array of strings to an array of integers:

```
int[] nums = numbers.Select(s => Int32.Parse(s)).ToArray();
```

That's it. How much easier could it get? Even just saying "abracadabra" only saves you 48 characters. Here is some code to display the resulting array of integers:

```
foreach(int num in nums)
  Console.WriteLine(num);
```

Here is the output showing the integers:

```
42
10
9
27
```

I know what you are thinking: maybe I just trimmed off the leading zeros. If I sort it, will you then be convinced? If they were still strings, 9 would be at the end, and 10 would be first. Listing 1-5 contains some code to do the conversion and sort the output.

Listing 1-5. *Converting an Array of Strings to Integers and Sorting It*

```
string[] numbers = { "0042", "010", "9", "27" };

int[] nums = numbers.Select(s => Int32.Parse(s)).OrderBy(s => s).ToArray();

foreach(int num in nums)
  Console.WriteLine(num);
```

Here are the results:

```
9
10
27
42
```

How slick is that? OK, you say, that is nice, but it sure is a simple example. Now I'll give you a more complex example.

Let's say you have some common code that contains an `Employee` class. In that `Employee` class is a method to return all the employees. Also assume you have another code base of common code that contains a `Contact` class, and in that class is a method to publish contacts. Let's assume you have the assignment to publish all employees as contacts.

The task seems simple enough, but there is a catch. The common `Employee` method that retrieves the employees returns the employees in an `ArrayList` of `Employee` objects, and the `Contact` method that publishes contacts requires an array of type `Contact`. Here is that common code:

```
namespace LINQDev.HR
{
  public class Employee
  {
    public int id;
    public string firstName;
    public string lastName;

    public static ArrayList GetEmployees()
    {
      //  Of course the real code would probably be making a database query
      //  right about here.
      ArrayList al = new ArrayList();

      //  Man, do the new C# object initialization features make this a snap.
      al.Add(new Employee { id = 1, firstName = "Joe", lastName = "Rattz"} );
      al.Add(new Employee { id = 2, firstName = "William", lastName = "Gates"} );
      al.Add(new Employee { id = 3, firstName = "Anders", lastName = "Hejlsberg"} );
      return(al);
    }
  }
}
```

```
namespace LINQDev.Common
{
  public class Contact
  {
    public int Id;
    public string Name;

    public static void PublishContacts(Contact[] contacts)
    {
      //  This publish method just writes them to the console window.
      foreach(Contact c in contacts)
        Console.WriteLine("Contact Id:  {0}  Contact:  {1}", c.Id, c.Name);
    }
  }
}
```

As you can see, the `Employee` class and `GetEmployees` method are in one namespace, `LINQDev.HR`, and the `GetEmployees` method returns an `ArrayList`. The `PublishContacts` method is in another namespace, `LINQDev.Common`, and requires an array of `Contact` objects to be passed.

Previously, this always meant iterating through the `ArrayList` returned by the `GetEmployees` method and creating a new array of type `Contact` to be passed to the `PublishContacts` method. LINQ makes it easy, as shown in Listing 1-6.

Listing 1-6. *Calling the Common Code*

```
ArrayList alEmployees = LINQDev.HR.Employee.GetEmployees();

LINQDev.Common.Contact[] contacts = alEmployees
  .Cast<LINQDev.HR.Employee>()
  .Select(e => new LINQDev.Common.Contact {
                 Id = e.id,
                 Name = string.Format("{0} {1}", e.firstName, e.lastName)
               })
  .ToArray<LINQDev.Common.Contact>();

LINQDev.Common.Contact.PublishContacts(contacts);
```

To convert the `ArrayList` of `Employee` objects to an array of `Contact` objects, I first cast the `ArrayList` of `Employee` objects to an `IEnumerable<Employee>` sequence using the `Cast` Standard Query Operator. This is necessary because the legacy `ArrayList` collection class was used. Syntactically speaking, objects of the `System.Object` class type are stored in an `ArrayList`, not objects of the `Employee` class type. So I must cast them to `Employee` objects. Had the `GetEmployees` method returned a generic `List` collection, this would not have been necessary. However, that collection type was not available when this legacy code was written.

Next, I call the `Select` operator on the returned sequence of `Employee` objects and in the *lambda expression*, the code passed inside the call to the `Select` method, I instantiate and initialize a `Contact` object using the new C# 3.0 object initialization features to assign the values from the input `Employee` element into a newly constructed output `Contact` element. A lambda expression is a new C# 3.0 feature that allows a new shorthand for specifying anonymous methods that I explain in Chapter 2. Lastly, I convert the sequence of newly constructed `Contact` objects to an array of `Contact` objects using the `ToArray` operator because that is what the `PublishContacts` method requires. Isn't that slick? Here are the results:

```
Contact Id:  1  Contact:  Joe Rattz
Contact Id:  2  Contact:  William Gates
Contact Id:  3  Contact:  Anders Hejlsberg
```

As you can see, LINQ can do a lot besides just querying data. As you read through the chapters of this book, try to think of additional uses for the features LINQ provides.

Tips to Get You Started

While working with LINQ to write this book, I often found myself confused, befuddled, and stuck. While there are many very useful resources available to the developer wanting to learn to use LINQ to its fullest potential, I want to offer a few tips to get you started. In some ways, these tips feel like they should come at the end of the book. After all, I haven't even explained what some of these concepts are at this point. But it would seem a bit sadistic to make you read the full text of the book first, only to offer the tips at the end. So with that said, this section contains some tips I think you might find useful, even if you do not fully understand them or the context.

Use the var Keyword When Confused

While it is necessary to use the `var` keyword when capturing a sequence of anonymous classes to a variable, sometimes it is a convenient way to get code to compile if you are confused. While I am very much in favor of developers knowing exactly what type of data is contained in a sequence—meaning that for `IEnumerable<T>` you should know what data type `T` is—sometimes, especially when just starting with LINQ, it can get confusing. If you find yourself stuck, where code will not compile because of some sort of data type mismatch, consider changing explicitly stated types so that they use the `var` keyword instead.

For example, let's say you have the following code:

```
//  This code will not compile.
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IEnumerable<?> orders = db.Customers
  .Where(c => c.Country == "USA" && c.Region == "WA")
  .SelectMany(c => c.Orders);
```

It may be a little unclear what data type you have an `IEnumerable` sequence of. You know it is an `IEnumerable` of some type `T`, but what is `T`? A handy trick would be to assign the query results to a variable whose type is specified with the `var` keyword, then to get the type of that variable so you know what type `T` is. Listing 1-7 shows what the code would look like.

Listing 1-7. *Code Sample Using the var Keyword*

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

var orders = db.Customers
  .Where(c => c.Country == "USA" && c.Region == "WA")
  .SelectMany(c => c.Orders);

Console.WriteLine(orders.GetType());
```

In this example, please notice that the orders variable type is now specified using the var keyword. Running this code produces the following:

```
System.Data.Linq.DataQuery`1[nwind.Order]
```

There is a lot of compiler gobbledygook there, but the important part is the nwind.Order portion. You now know that the data type you are getting a sequence of is nwind.Order.

If the gobbledygook is throwing you, running the example in the debugger and examining the orders variable in the Locals window reveals that the data type of orders is this:

```
System.Linq.IQueryable<nwind.Order> {System.Data.Linq.DataQuery<nwind.Order>}
```

This makes it clearer that you have a sequence of nwind.Order. Technically, you have an IQueryable<nwind.Order> here, but that can be assigned to an IEnumerable<nwind.Order> if you like, since IQueryable<T> inherits from IEnumerable<T>.

So you could rewrite the previous code, plus enumerate through the results, as shown in Listing 1-8.

Listing 1-8. *Sample Code from Listing 1-7 Except with Explicit Types*

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IEnumerable<Order> orders = db.Customers
  .Where(c => c.Country == "USA" && c.Region == "WA")
  .SelectMany(c => c.Orders);

foreach(Order item in orders)
  Console.WriteLine("{0} - {1} - {2}", item.OrderDate, item.OrderID, item.ShipName);
```

■ **Note** For the previous code to work, you will need to have a using directive for the System.Collections.Generic namespace, in addition to the System.Linq namespace you should always expect to have when LINQ code is present.

This code would produce the following abbreviated results:

```
3/21/1997 12:00:00 AM - 10482 - Lazy K Kountry Store
5/22/1997 12:00:00 AM - 10545 - Lazy K Kountry Store
…
4/17/1998 12:00:00 AM - 11032 - White Clover Markets
5/1/1998 12:00:00 AM - 11066 - White Clover Markets
```

Use the Cast or OfType Operators for Legacy Collections

You will find that the majority of LINQ's Standard Query Operators can only be called on collections implementing the IEnumerable<T> interface. None of the legacy C# collections—those in the System.Collection namespace—implement IEnumerable<T>. So the question becomes, how do you use LINQ with legacy collections in your existing code base?

There are two Standard Query Operators specifically for this purpose, `Cast` and `OfType`. Both of these operators can be used to convert legacy collections to `IEnumerable<T>` sequences. Listing 1-9 shows an example.

Listing 1-9. *Converting a Legacy Collection to an IEnumerable<T> Using the Cast Operator*

```
//  I'll build a legacy collection.
ArrayList arrayList = new ArrayList();
//  Sure wish I could use collection initialization here, but that
//  doesn't work with legacy collections.
arrayList.Add("Adams");
arrayList.Add("Arthur");
arrayList.Add("Buchanan");

IEnumerable<string> names = arrayList.Cast<string>().Where(n => n.Length < 7);
foreach(string name in names)
  Console.WriteLine(name);
```

Listing 1-10 shows the same example using the `OfType` operator.

Listing 1-10. *Using the OfType Operator*

```
//  I'll build a legacy collection.
ArrayList arrayList = new ArrayList();
//  Sure wish I could use collection initialization here, but that
//  doesn't work with legacy collections.
arrayList.Add("Adams");
arrayList.Add("Arthur");
arrayList.Add("Buchanan");

IEnumerable<string> names = arrayList.OfType<string>().Where(n => n.Length < 7);
foreach(string name in names)
  Console.WriteLine(name);
```

Both examples provide the exact same results. Here they are:

```
Adams
Arthur
```

The difference between the two operators is that the `Cast` operator will attempt to cast every element in the collection to the specified type to be put into the output sequence. If there is a type in the collection that cannot be cast to the specified type, an exception will be thrown. The `OfType` operator will only attempt to put those elements that can be cast to the type specified into the output sequence.

Prefer the OfType Operator to the Cast Operator

One of the most important reasons why generics were added to C# was to give the language the ability to have data collections with static type checking. Prior to generics—barring creating your own specific collection type for every type of data for which you wanted a collection—there was no way to ensure that every element in a legacy collection, such as an `ArrayList`, `Hashtable`, and so on, was of the same and correct type. Nothing in the language prevented code from adding a `Textbox` object to an `ArrayList` meant to contain only `Label` objects.

With the introduction of generics in C# 2.0, C# developers now have a way to explicitly state that a collection can only contain elements of a specified type. While either the `OfType` or `Cast` operator may work for a legacy collection, `Cast` requires that every object in the collection be of the correct type, which is the fundamental original flaw in the legacy collections for which generics were created. When using the `Cast` operator, if any object is unable to be cast to the specified data type, an exception is thrown. Instead, use the `OfType` operator. With it, only objects of the specified type will be stored in the output `IEnumerable<T>` sequence, and no exception will be thrown. The best-case scenario is that every object will be of the correct type and be in the output sequence. The worst case is that some elements will get skipped, but they would have thrown an exception had the `Cast` operator been used instead.

Don't Assume a Query Is Bug-Free

In Chapter 3, I discuss that LINQ queries are often deferred and not actually executed when it appears you are calling them. For example, consider this code fragment from Listing 1-1:

```
var items =
  from s in greetings
  where s.EndsWith("LINQ")
  select s;

foreach (var item in items)
  Console.WriteLine(item);
```

While it *appears* the query is occurring when the `items` variable is being initialized, that is not the case. Because the `Where` and `Select` operators are deferred, the query is not actually being performed at that point. The query is merely being called, declared, or defined, but not performed. The query will actually take place the first time a result from it is needed. This is typically when the query results variable is enumerated. In this example, a result from the query is not needed until the `foreach` statement is executed. That is the point in time that the query will be performed. In this way, we say that the query is *deferred*.

It is often easy to forget that many of the query operators are deferred and will not execute until a sequence is enumerated. This means you could have an improperly written query that will throw an exception when the resulting sequence is eventually enumerated. That enumeration could take place far enough downstream that it is easily forgotten that a query may be the culprit.

Let's examine the code in Listing 1-11.

Listing 1-11. *Query with Intentional Exception Deferred Until Enumeration*

```
string[] strings = { "one", "two", null, "three" };

Console.WriteLine("Before Where() is called.");
IEnumerable<string> ieStrings = strings.Where(s => s.Length == 3);
Console.WriteLine("After Where() is called.");

foreach(string s in ieStrings)
{
  Console.WriteLine("Processing " + s);
}
```

I know that the third element in the array of strings is a `null`, and I cannot call `null.Length` without throwing an exception. The execution steps over the line of code calling the query just fine. It is not until I enumerate the sequence `ieStrings`, and specifically the third element, that the exception occurs. Here are the results of this code:

```
Before Where() is called.
After Where() is called.
Processing one
Processing two

Unhandled Exception: System.NullReferenceException: Object reference not set to an
instance of an object.
…
```

As you can see, I called the `Where` operator without exception. It's not until I try to enumerate the third element of the sequence that an exception is thrown. Now imagine if that sequence, `ieStrings`, is passed to a function that downstream enumerates the sequence, perhaps to populate a drop-down list or some other control. It would be easy to think the exception is caused by a fault in that function, not the LINQ query itself.

Take Advantage of Deferred Queries

In Chapter 3, I go into deferred queries in more depth. However, I want to point out that if a query is a deferred query that ultimately returns an `IEnumerable<T>`, that `IEnumerable<T>` object can be enumerated over, time and time again, obtaining the latest data from the data source. You don't need to actually call or, as I earlier pointed out, declare the query again.

In most of the code samples in this book, you will see a query called and an `IEnumerable<T>` for some type `T` being returned and stored in a variable. Then I typically call `foreach` on the `IEnumerable<T>` sequence. This is for demonstration purposes. If that code is executed multiple times, calling the actual query each time is needless work. It might make more sense to have a query initialization method that gets called once for the lifetime of the scope and to construct all the queries there. Then you could enumerate over a particular sequence to get the latest version of the query results at will.

Use the DataContext Log

When working with LINQ to SQL, don't forget that the database class that is generated by SQLMetal inherits from `System.Data.Linq.DataContext`. This means that your generated `DataContext` class has some useful built-in functionality, such as a `TextWriter` object named `Log`.

One of the niceties of the `Log` object is that it will output the equivalent SQL statement of an `IQueryable<T>` query prior to the parameter substitution. Have you ever had code break in production that you think might be data related? Wouldn't it be nice if there was a way to get the query executed against the database, so that you could enter it in SQL Server Enterprise Manager or Query Analyzer and see the exact data coming back? The `DataContext`'s `Log` object will output the SQL query for you. An example is shown in Listing 1-12.

Listing 1-12. *An Example Using the DataContext.Log Object*

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

db.Log = Console.Out;

IQueryable<Order> orders = from c in db.Customers
                           from o in c.Orders
                           where c.Country == "USA" && c.Region == "WA"
                           select o;

foreach(Order item in orders)
  Console.WriteLine("{0} - {1} - {2}", item.OrderDate, item.OrderID, item.ShipName);
```

This code produces the following output:

```
SELECT [t1].[OrderID], [t1].[CustomerID], [t1].[EmployeeID], [t1].[OrderDate],
[t1].[RequiredDate], [t1].[ShippedDate], [t1].[ShipVia], [t1].[Freight],
[t1].[ShipName], [t1].[ShipAddress], [t1].[ShipCity], [t1].[ShipRegion],
[t1].[ShipPostalCode], [t1].[ShipCountry]
FROM [dbo].[Customers] AS [t0], [dbo].[Orders] AS [t1]
WHERE ([t0].[Country] = @p0) AND ([t0].[Region] = @p1) AND ([t1].[CustomerID] =
[t0].[CustomerID])
-- @p0: Input String (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p1: Input String (Size = 2; Prec = 0; Scale = 0) [WA]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

3/21/1997 12:00:00 AM - 10482 - Lazy K Kountry Store
5/22/1997 12:00:00 AM - 10545 - Lazy K Kountry Store
6/19/1997 12:00:00 AM - 10574 - Trail's Head Gourmet Provisioners
6/23/1997 12:00:00 AM - 10577 - Trail's Head Gourmet Provisioners
1/8/1998 12:00:00 AM - 10822 - Trail's Head Gourmet Provisioners
7/31/1996 12:00:00 AM - 10269 - White Clover Markets
11/1/1996 12:00:00 AM - 10344 - White Clover Markets
3/10/1997 12:00:00 AM - 10469 - White Clover Markets
3/24/1997 12:00:00 AM - 10483 - White Clover Markets
4/11/1997 12:00:00 AM - 10504 - White Clover Markets
7/11/1997 12:00:00 AM - 10596 - White Clover Markets
10/6/1997 12:00:00 AM - 10693 - White Clover Markets
10/8/1997 12:00:00 AM - 10696 - White Clover Markets
10/30/1997 12:00:00 AM - 10723 - White Clover Markets
11/13/1997 12:00:00 AM - 10740 - White Clover Markets
1/30/1998 12:00:00 AM - 10861 - White Clover Markets
2/24/1998 12:00:00 AM - 10904 - White Clover Markets
4/17/1998 12:00:00 AM - 11032 - White Clover Markets
5/1/1998 12:00:00 AM - 11066 - White Clover Markets
```

Use the LINQ Forum

Despite providing the best tips I can think of, there will more than likely be times when you get stuck. Don't forget that there is a forum dedicated to LINQ at MSDN.com. You can find a link to it here: `http://www.linqdev.com`. This forum is monitored by Microsoft developers, and you will find a wealth of knowledgeable resources there.

Summary

I sense that by now you are chomping at the bit to move on to the next chapter, but before you do I want to remind you of a few things.

First, LINQ *is* going to change the way .NET developers query data. Vendors will more than likely be lining up to add a "LINQ Compatible" sticker to their products, just like they currently do with XML.

Bear in mind that LINQ is not just a new library to be added to your project. It is a total approach to querying data that comprises several components depending on the data store being queried. At the present time, you can use LINQ to query the following data sources; in-memory data collections using LINQ to Objects, XML using LINQ to XML, DataSets using LINQ to DataSet, and SQL Server databases using LINQ to SQL.

Also, please remember what I said about LINQ not being just for queries. In a sample project I have been working on using LINQ, I have found LINQ very useful for not only querying data, but for getting data into the necessary format for presentation in a WinForm control.

Last but not least, I hope you didn't skip over the tips I provide in this chapter. If you don't understand some of them, that is no problem. They will make more sense as you progress through the book. Just. keep them in mind if you find yourself stalled.

No doubt that after seeing some of the LINQ examples and tips in this chapter, you may find yourself puzzled by some of the syntax that makes this all possible. If so, don't worry because in the next chapter, I cover the enhancements Microsoft has made to C# in version 3.0 that make all of this possible.