

Paul Kimmel

Foreword by Darryl Hogan  
Architect Evangelist, Microsoft Corporation

# LINQ

UNLEASHED

for C#

**SAMS**

## **LINQ Unleashed for C#**

Copyright © 2009 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-32983-8

ISBN-10: 0-672-32983-2

*Library of Congress Cataloging-in-Publication Data*

Kimmel, Paul.

LINQ unleashed for C# / Paul Kimmel. — 1st ed.

p. cm.

ISBN 978-0-672-32983-8

1. C# (Computer program language) 2. Microsoft LINQ. I. Title.

QA76.73.C154K5635 2009

005.13'3—dc22

2008030703

Printed in the United States of America

First Printing August 2008

## **Trademarks**

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

## **Warning and Disclaimer**

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

## **Bulk Sales**

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

**U.S. Corporate and Government Sales**

**1-800-382-3419**

**corpsales@pearsontechgroup.com**

For sales outside of the U.S., please contact

**International Sales**

**international@pearson.com**

## **Editor-in-Chief**

Karen Gettman

## **Executive Editor**

Neil Rowe

## **Development Editor**

Mark Renfrow

## **Managing Editor**

Kristy Hart

## **Project Editor**

Betsy Harris

## **Copy Editor**

Karen Annett

## **Indexers**

Lisa Stumpf

Publishing Works

## **Proofreader**

Linda Seifer

## **Technical Editor**

Joe Kunk

## **Publishing**

**Coordinator**

Cindy Teeters

## **Cover Designer**

Gary Adair

## **Compositor**

Jake McFarland

# Foreword

Data affects just about every aspect of our lives. Everything we do is analyzed, scrutinized, and delivered back to us in the form of coupons and other marketing materials. When you write an application, you can be sure that data in one form or another will be part of the solution. As software developers, the ease with which we can store, retrieve, and analyze data is crucial to our ability to develop compelling applications. Add to that the fact that data can come in a number of different shapes and formats, and it quickly comes to light that there is tremendous value in a consistent framework for accessing many types of data.

Several different data access approaches have been developed for Windows developers over the years. ADO and OLEDB and subsequently ADO.NET gave us universal access to relational databases. MSXML and ADO.NET made it possible to inspect and manipulate XML documents. Each of these technologies had their benefits and drawbacks, but one common thread ran through each of them: They failed to deliver data access capabilities in a way that felt natural to developers.

LINQ now makes data access a first-class programming concept in .NET, making it possible for developers to express queries in a way that makes sense to developers. What makes LINQ unique is that it enables programmers to create type-safe data access code complete with Intellisense support and compile time syntax checking.

Paul Kimmel has done an excellent job of presenting LINQ in a concise and complete manner. Not only has he made LINQ approachable, but he has also masterfully explained concepts such as Anonymous Types and Lambda Expressions that help make LINQ a reality. The sample code throughout the book demonstrates the application of the technology in a clear and meaningful way. This is a great “Saturday morning with a pot of coffee” kind of book. I hope you’ll dive in and get as much out of this book as I did.

**Darryl Hogan**  
Architect Evangelist, Microsoft

# Introduction

By the time you are holding this book in your hands, I will have 30 years in since the first time I wrote some code. That code was ROM-BASIC on a TRS-80 in Washington grammar school in Owosso, Michigan, and I was in the fifth grade. Making the “tank” slide back and forth shooting blips across the screen was neat. Changing the code to change blip speeds and numbers of targets was exhilarating. Three decades later and I get more excited each passing year. There are great technologies on the horizon like Microsoft Surface, Popfly, and LINQ. This book is about LINQ, or Language INtegrated Query.

LINQ is a SQL-like language for C#. When I first saw it, I didn't like it. My first impression was that someone had glommed on a bastardization of C# and it was ugly like SQL can get. I didn't like it because I didn't understand it. However, I gave LINQ a second chance (as I want you to do) and discovered that LINQ is thoroughly integrated, tremendously powerful, and almost as much fun as a Tesla Roadster or doing hammerheads in an Extra 300L.

The query capabilities of LINQ are extended to objects, SQL, DataSets, XML, XSD, entities, and can be extended to other providers like Active Directory or SharePoint. This means that you can write queries—that are similar in syntax—against objects, data, XML, XSD, entities, or Active Directory (with a little work) much like you would a SQL query in a database. And, LINQ is actually engineered artfully and brilliantly on top of generics as well as some new features in .NET 3.5, such as extension methods, anonymous types, and Lambda Expressions. Another very important characteristic of LINQ is that it clearly demonstrates Microsoft's willingness to innovate and take the best of existing technologies

like Lambda Calculus—invented in the 1930s—and if it's good or great, incorporate these elements into the tools and languages we love.

LINQ and its underpinnings are powerful *and* challenging, and in this book you will get what you need to know to completely understand all that makes LINQ work and begin using it immediately. You will learn about anonymous methods, extension methods, Lambda Expressions, state machines, how generics and the CodeDOM play a big role in powerful tools like LINQ, and writing LINQ queries and why you will want to do it in the bigger, grander scheme of things. You will also learn how to save a ton of time and effort by not hard-coding those elements that you will no longer need or want to hard-code, and you will have a better grasp of how LINQ fits into n-tier architectures without breaking guidelines that have helped you succeed to date.

Brought to you by a four-time Microsoft MVP and columnist for over a decade, *LINQ Unleashed for C#* will teach you everything you need to know about LINQ and .NET 3.5 features and how to be more productive and have more fun than ever before.

## Conventions Used in This Book

The following typographic conventions are used in this book:

Code lines, commands, statements, variables, and text you see onscreen appear in a monospace typeface.

Occasionally in listings bold is used to draw attention to the snippet of code being discussed.

Placeholders in syntax descriptions appear in an *italic monospace* typeface. You replace the placeholder with the actual filename, parameter, or whatever element it represents.

*Italics* highlight technical terms when they're being defined.

A code-continuation icon is used before a line of code that is really a continuation of the preceding line. Sometimes a line of code is too long to fit as a single line on the page. If you see ➤ before a line of code, remember that it's part of the line immediately above it.

The book also contains Notes, Tips, and Cautions to help you spot important or useful information more quickly.

## CHAPTER 1

# Programming with Anonymous Types

*“Begin at the beginning and go on till you come to the end: then stop.”*

—Lewis Carroll, from *Alice’s Adventures in Wonderland*

Finding a beginning is always a little subjective in computer books. This is because so many things depend on so many other things. Often, the best we can do is put a stake in the ground and start from that point. Anonymous types are our stake.

Anonymous types use the keyword `var`. `Var` is an interesting choice because it is still used in Pascal and Delphi today, but `var` in Delphi is like `ByRef` in Visual Basic (VB) or `ref` in C#. The `var` introduced with .NET 3.5 indicates an anonymous type. Now, our VB friends are going to think, *“Well, we have had variants for years; big deal.”* But `var` is not a dumbing down and clogging up of C#. Anonymous types are something new and necessary.

Before looking at anonymous types, let’s put a target on our end goal. Our end goal is to master LINQ (integrated queries) in C# for objects, Extensible Markup Language (XML), and data. We want to do this because it’s cool, it’s fun, and, more important, it is very powerful and expressive. To get there, we have to start somewhere and anonymous types are our starting point.

Anonymous types quite simply mean that you don’t specify the type. You write `var` and C# figures out what type is defined by the right side, and C# emits (writes the code), indicating the type. From that point on, the type is strongly defined, checked by the compiler (not at runtime), and exists as a complete type in your code. Remember, you

## IN THIS CHAPTER

- ▶ Understanding Anonymous Types
- ▶ Programming with Anonymous Types
- ▶ Databinding Anonymous Types
- ▶ Testing Anonymous Type Equality
- ▶ Using Anonymous Types with LINQ Queries
- ▶ Introducing Generic Anonymous Methods

didn't write the type definition; C# did. This is important because in a query language, you are asking for and getting *ad hoc* types that are defined by the context, the query result. In short, your query's result might return a previously undefined type.

An important concept here is that you don't write code to define the *ad hoc* types—C# does—so, you save time by not writing code. You save design time, coding time, and debug time. Microsoft pays that cost. Anonymous types are the vessel that permit you to use these *ad hoc* types. By the time you are done with this chapter, you will have mastered the left side of the operator and a critical part of LINQ.

In addition, to balance the book, the chapters are laced with useful or related concepts that are generally helpful. This chapter includes a discussion on generic anonymous methods.

## Understanding Anonymous Types

Anonymous types defined with `var` are not VB variants. The `var` keyword signals the compiler to emit a strong type based on the value of the operator on the right side. Anonymous types can be used to initialize simple types like integers and strings but detract modestly from clarity and add little value. Where `var` adds punch is by initializing composite types on the fly, such as those returned from LINQ queries. When such an anonymous type is defined, the compiler emits an immutable—read-only properties—class referred to as a projection.

Anonymous types support IntelliSense, but the class should not be referred to in code, just the members.

The following list includes some basic rules for using anonymous types:

- ▶ Anonymous types must always have an initial assignment and it can't be null because the type is inferred and fixed to the initializer.
- ▶ Anonymous types can be used with simple or complex types but add little value to simple type definitions.
- ▶ Composite anonymous types require member declarators; for example, `var joe = new {Name="Joe" [, declaratory=value, ...]}`. (In the example, `Name` is the member declaratory.)
- ▶ Anonymous types support IntelliSense.
- ▶ Anonymous types cannot be used for a class field.
- ▶ Anonymous types can be used as initializers in `for` loops.
- ▶ The `new` keyword can be used and has to be used for array initializers.
- ▶ Anonymous types can be used with arrays.
- ▶ Anonymous types are all derived from the `Object` type.
- ▶ Anonymous types can be returned from methods but must be cast to `object`, which defeats the purpose of strong typing.

- Anonymous types can be initialized to include methods, but these might only be of interest to linguists.

The single greatest value and the necessity of anonymous types is they support creating single-use elements and composite types returned by LINQ queries without the need for the programmer to fully define these types in static code. That is, the designers can focus significantly on primary domain types, and the programmers can still create single-use anonymous types ad hoc, letting the compiler write the class definition.

Finally, because anonymous types are immutable—think no property setters—two separately defined anonymous types with the same field values are considered equal.

## Programming with Anonymous Types

This chapter continues by exploring all of the ways you can use anonymous types, paving the way up to anonymous types returned by LINQ queries, stopping at the full explanation of the LINQ query here. You can simply think of the query as a first look at queries with the focus being on the anonymous type itself and what you can do with those types.

### Defining Simple Anonymous Types

A simple anonymous type begins with the `var` keyword, the assignment operator (`=`), and a non-null initial value. The anonymous type is assigned to the name on the left side of the assignment operator, and the type emitted by the compiler to Microsoft Intermediate Language (MSIL) is determined by the right side of the operator. For instance:

```
var title = "LINQ Unleashed for C#";
```

uses the anonymous type syntax and assigns the string value to “LINQ Unleashed for C#”. This code is identical in the MSIL to the following:

```
string title = "LINQ Unleashed for C#";
```

This emitted code equality can be seen by looking at the Intermediate Language (IL) with the Intermediate Language Disassembler (ILDASM) utility (see Figure 1.1).

The support for declaring simple anonymous types exists more for completeness and symmetry than utility. In departmental language wars, purists are likely to rail against such use as it adds ambiguity to code. The truth is the type of the data is obvious in such simple use examples and it hardly matters.

### Using Array Initializer Syntax

You can use anonymous type syntax for initializing arrays, too. The requirements are that the `new` keyword must be used. For example, the code in Listing 1.1 shows a simple console application that initializes an anonymous array of Fibonacci numbers. (The anonymous type and array initialization statement are highlighted in bold font.)



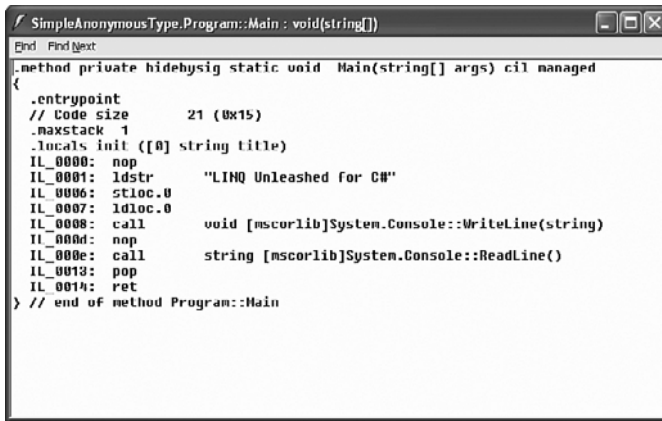


FIGURE 1.1 Looking at the `.locals init` statement and the `Console::Write(string)` statement in the MSIL, it is clear that `title` is emitted as a string.

#### LISTING 1.1 An Anonymous Type Initialized with an Array of Integers

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ArrayInitializer
{
    class Program
    {
        static void Main(string[] args)
        {
            // array initializer
            var fibonacci = new int[] { 1, 1, 2, 3, 5, 8, 13, 21 };
            Console.WriteLine(fibonacci[0]);
            Console.ReadLine();
        }
    }
}

```

The first eight numbers in the Fibonacci system are defined on the line that begins `var fibonacci`. Fibonacci numbers start with the number 1 and the sequence is resolved by adding the prior two numbers. (For more information on Fibonacci numbers, check out Wikipedia; Wikipedia is wicked cool at providing detailed facts about such esoterica.)

Even in the example shown in Listing 1.1, you are less likely to get involved in language ambiguity wars if you use the actual type `int[]` instead of the anonymous type syntax for arrays.

## Creating Composite Anonymous Types

Anonymous types really start to shine when they are used to define composite types, that is, classes without the “typed” class definition. Think of this use of anonymous types as defining an inline class without all of the typing. Listing 1.2 shows an anonymous type representing a lightweight person class.

LISTING 1.2 An Anonymous Type Containing Two Fields and Two Properties Without All of the Class Plumbing Typed By the Programmer

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ImmutableAnonymousTypes
{
    class Program
    {
        static void Main(string[] args)
        {
            var dena = new {First="Dena", Last="Swanson"};
            //dena.First = "Christine"; // error - immutable
            Console.WriteLine(dena);
            Console.ReadLine();
        }
    }
}
```

The anonymous type defined on the line starting with `var dena` emits a class, referred to as a projection, in the MSIL (see Figure 1.2). Although the projection’s name—the class name—cannot be referred to in code, the member elements—defined by the member declarators `First` and `Last`—can be used in code and IntelliSense works for all the elements of the projection (see Figure 1.3).

Another nice feature added to anonymous types is the overloaded `ToString` method. If you look at the MSIL or the output from Listing 1.2, you will see that the field names and field values, neatly formatted, are returned from the emitted `ToString` method. This is useful for debugging.

### Adding Behaviors to Anonymous Composite Types

If you try to add a behavior to an anonymous type at initialization—for instance, by using an anonymous delegate—the compiler reports an error. However, it is possible with a little bending and twisting to add behaviors to anonymous types. The next section shows you how.



## LISTING 1.4 Adding a Behavior to an Anonymous Type

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;

namespace AnonymousTypeWithMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            // adding method possibility
            Func<string, string, string> Concat1 =
                delegate(string first, string last)
                {
                    return last + ", " + first;
                };

            // whacky method but works
            Func<Type, Object, string> Concat2 =
                delegate(Type t, Object o)
                {
                    PropertyInfo[] info = t.GetProperties();
                    return (string)info[1].GetValue(o, null) +
                        ", " + (string)info[0].GetValue(o, null);
                };

            var dena = new {First="Dena", Last="Swanson", Concat=Concat1};
            //var dena = new {First="Dena", Last="Swanson", Concat=Concat2};
            Console.WriteLine(dena.Concat(dena.First, dena.Last));
            //Console.WriteLine(dena.Concat(dena.GetType(), dena));
            Console.ReadLine();
        }
    }
}

```

The technique consists of defining an anonymous delegate and assigning that anonymous delegate to the generic `Func` class. In the example, `Concat` was defined as an anonymous delegate that accepts two strings, concatenates them, and returns a string. You can assign that delegate to a variable defined as an instance of `Func` that has the three string parameter types. Finally, you assign the variable `Concat` to a member declarator in the anonymous type definition (referring to `var dena = new {First="Dena", Last="Swanson", Concat=Concat1};` now).

After the plumbing is in place, you can use IntelliSense to see that the behavior—Concat—is, in fact, part of the anonymous type dena, and you can invoke it in the usual manner.

## Using Anonymous Type Indexes in For Statements

The `var` keyword can be used to initialize the index of a `for` loop or the recipient object of a `foreach` loop. The former is a simple anonymous type and the latter becomes a useful construct when the container to iterate over is something more than a sample collection. Listing 1.5 shows a `for` statement, and Listing 1.6 shows the `foreach` statement, both using the `var` construct.

LISTING 1.5 Demonstrating How to Iterate Over an Array of Integers—Using the Fibonacci Numbers from Listing 1.1—and the `var` Keyword to Initialize the Index

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AnonymousForLoop
{
    class Program
    {
        static void Main(string[] args)
        {
            var fibonacci = new int[]{ 1, 1, 2, 3, 5, 8, 13, 21 };
            for( var i=0; i<fibonacci.Length; i++)
                Console.WriteLine(fibonacci[i]);
            Console.ReadLine();
        }
    }
}
```

---

LISTING 1.6 Demonstrating Basically the Same Code but Using the More Convenient `foreach` Construct

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AnonymousForEachLoop
{
    class Program
    {
        static void Main(string[] args)
```

## LISTING 1.6 Continued

```
{
    var fibonacci = new int[]{ 1, 1, 2, 3, 5, 8, 13, 21 };
    foreach( var fibo in fibonacci)
        Console.WriteLine(fibo);
    Console.ReadLine();
}
}
```

The only requirement that must be met for an object to be the iterand in a `foreach` statement is that it must functionally represent an object that implements `IEnumerable` or `IEnumerable<T>`—the generic equivalent. Incidentally, this is also the same requirement for bindability, as in binding to a `GridView`.

**TIP**

At any time, you can branch in `for` or `foreach` statements with the `break` or `continue` keywords or the `goto`, `return`, or `throw` statements.

An all-too-common use of the `for` construct is to copy a subset of elements from one collection of objects to a new collection, for example, copying all the customers in the 48843 ZIP code to a `customersToCallOn` collection. In C# 2.0, the `yield return` and `yield break` key phrases actually played this role. For example, `yield return` signaled the compiler to emit a *state machine* in MSIL—in essence, it emitted the copy collection for you.

In .NET 3.5, the ability to query collections, datasets, and XML to essentially ask questions about data or copy some elements is one of those things that LINQ does very well. Listing 1.7 shows code that uses a LINQ statement to return just the numbers in the Fibonacci short sequence that are divisible by 3. (For now, don't worry about understanding all of the elements of the query.)

LISTING 1.7 A `foreach` Statement Whose Iterand Is Derived from a LINQ Query

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AnonymousForEachLoopFromExpression
{
    class Program
    {
```

LISTING 1.7 Continued

---

```

static void Main(string[] args)
{
    var fibonacci = new int[]{ 1, 1, 2, 3, 5, 8, 13, 21, 33, 54, 87 };
    // uses LINQ query
    foreach( var fibo in from f in fibonacci where f%3==0 select f)
        Console.WriteLine(fibo);
    Console.ReadLine();
}
}
}

```

---

The LINQ query—used as the iterand in the `foreach` statement—makes up this part of the Listing 1.7:

```
from f in fibonacci where f % 3 == 0 select f
```

For now, it is enough to know that this query meets the requirement that it returns an *enumerable* result, in fact, `IEnumerable<T>` where *T* is an *int* type.

If this is your first experience with LINQ, the query might look strange. The capability and power and this book will quickly make them familiar and desirable friends. For now, it is enough to know that queries meet the requirement of an enumerable resultset and can be used in a `foreach` statement.

## Anonymous Types and Using Statements

The `using` statement is shorthand notation for `try...finally`. With `try...finally` and `using`, the purpose is to ensure resources are cleaned up before the `using` block exits or the `finally` block is run. This is accomplished by calling `Dispose`, which implies that items created in `using` statements implement `IDisposable`. Employ `using` when the created types implement `IDisposable`—like `SqlConnection`s—and use `try...finally` when you need to do some kind of cleanup work, but do not necessarily need to invoke `Dispose` (see Listing 1.8).

LISTING 1.8 Using Statement and `var` Work Because `SqlConnection` Implements `IDisposable`

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace AnonymousUsingStatement

```

## LISTING 1.8 Continued

```

{
    class Program
    {
        static void Main(string[] args)
        {
            string connectionString =
                "Data Source=BUTLER;Initial Catalog=AdventureWorks2000;" +
                "Integrated Security=True";
            using( var connection = new SqlConnection(connectionString))
            {
                connection.Open();
                Console.WriteLine(connection.State);
                Console.ReadLine();
            }
        }
    }
}

```

The help documentation will verify that `SqlConnection` is derived from `DBConnection`, which, in turn, implements `IDisposable`. You can use a tool like Anakrino or Reflector—free decompilers and disassemblers—to see that `Dispose` in `DBConnection` invokes the `Close` method on a connection.

To really understand how things are implemented, you can use ILDASM—or one of the previously mentioned decompilers—and look at the MSIL that is emitted. If you look at the code in Listing 1.8's IL, you can clearly see the substitution of `using` for a properly configured `try...finally` block. (The `try` element—after `SqlConnection` creation—and the `finally` block invoking `Dispose` are shown in bold font in Listing 1.9.)

LISTING 1.9 The MSIL for the `var` and `using` Statement in Listing 1.8

```

.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size          66 (0x42)
    .maxstack 2
    .locals init ([0] string connectionString,
                  [1] class [System.Data]System.Data.SqlClient.SqlConnection connection,
                  [2] bool CS$4$0000)
    IL_0000: nop
    IL_0001: ldstr      "Data Source=BUTLER;Initial Catalog=AdventureWorks2"
    + "000;Integrated Security=True"
    IL_0006: stloc.0

```



## LISTING 1.9 Continued

---

```

IL_0007: ldloc.0
IL_0008: newobj      instance void
↳[System.Data]System.Data.SqlClient.SqlConnection::.ctor(string)
IL_000d: stloc.1
    .try
    {
        IL_000e: nop
        IL_000f: ldloc.1
        IL_0010: callvirt instance void
[System.Data]System.Data.Common.DbConnection::Open()
        IL_0015: nop
        IL_0016: ldloc.1
        IL_0017: callvirt      instance valuetype[System.Data]System.Data.ConnectionState
[System.Data]System.Data.Common.DbConnection::get_State()
        IL_001c: box          [System.Data]System.Data.ConnectionState
        IL_0021: call          void [mscorlib]System.Console::WriteLine(object)
        IL_0026: nop
        IL_0027: call          string [mscorlib]System.Console::ReadLine()
        IL_002c: pop
        IL_002d: nop
        IL_002e: leave.s      IL_0040
    } // end .try
    finally
    {
        IL_0030: ldloc.1
        IL_0031: ldnull
        IL_0032: ceq
        IL_0034: stloc.2
        IL_0035: ldloc.2
        IL_0036: brtrue.s      IL_003f
        IL_0038: ldloc.1
        IL_0039: callvirt      instance void [mscorlib]System.IDisposable::Dispose()
        IL_003e: nop
        IL_003f: endfinally
    } // end handler
IL_0040: nop
IL_0041: ret
} // end of method Program::Main

```

---

You don't have to master IL to use .NET effectively, but you can learn from it and writing .NET emitters—code that emits IL directly—is supported in the .NET Framework. As shown in the MSIL, you can infer, for example, that the proper way to use `try...finally` is to create the protected object, try to use it, and, finally, clean it up. If you read a little further—in the `finally` block starting with IL 0030—you can see that the compiler also

put a check in to ensure that the protected object, the `SqlConnection`, is compared with null before `Dispose` is called. This code is demonstrated in IL 0030, IL 0031, IL 0032, and the branch statement on IL 0036.

## Returning Anonymous Types from Functions

Anonymous types can be returned from functions because the garbage collector (GC) cleans up any objects, but outside of the defining scope, the anonymous type is an instance of an object. Unfortunately, returning an object defeats the value of the IntelliSense system and the strongly typed nature of anonymous types. Although you could use reflection to rediscover the capabilities of the anonymous type, again you are taking a feature intended to make life more convenient and making it somewhat inconvenient again. Listing 1.10 puts these elements together, but as a practical matter, it is best to design solutions to use anonymous types within the defining scope. (Ironically, using objects within the defining scope was a style issue used in C++ to reduce the probability of memory leaks. Those familiar with C++ won't find this slight quirk of anonymous types any more inconvenient.)

LISTING 1.10 Returning an Anonymous Type from a Method Defeats the Strongly Typed Utility of Anonymous Types

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;

namespace ReturnAnonymousTypeFromMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            var anon = GetAnonymous();
            Type t = anon.GetType();
            Console.WriteLine(t.GetProperty("Stock").GetValue(anon, null));
            Console.ReadLine();
        }

        public static object GetAnonymous()
        {
            var stock = new {Stock="MSFT", Price="32.45"};
            return stock;
        }
    }
}
```

Although it is intellectually satisfying to play with the reflection subsystem, writing code like that in Listing 1.10 is a slow and painful means to an end. (In addition, the code in Listing 1.10, as written, is fraught with the potentiality for bugs due to null values being returned from `GetType`, `GetProperty`, and `GetValue`.)

## Databinding Anonymous Types

Some interesting startups got blown up when the stock market bubble burst, such as PointCast. PointCast searched the web—based on criteria the user provided—and displayed stock prices on a ticker and news in a browsable environment. One of the possible kinds of data was streaming stock prices. (Thankfully, the 1990s day-trading craze is over, but the ability to get such data is still interesting.)

This section looks at how you can combine cool technologies, such as anonymous types, AJAX, `HttpRequests`, `HttpResponses`, and queries to Yahoo!'s stock-quoting capability, and assemble a web stock ticker. Aside from the code, a demonstration of data-binding anonymous types, and a brief description of what role the various technology elements are playing, this book doesn't elaborate in detail on features like AJAX (because of space and topic constraints). (For more information on web programming, see Stephen Walther's *ASP.NET 3.5 Unleashed*.)

The sample (in Listing 1.11) is actually very easy to complete, but uses some very cool technology and plumbing underneath. In the solution, a website project was created. The application contains a single `.aspx` web page. On that page, a `ScriptManager`, `UpdatePanel` (both AJAX controls), a `DataList`, `Label`, and AJAX `Timer` are added. The design-time view of the page is shown in Figure 1.4 and the runtime view is shown in Figure 1.5. (Listing 1.12 shows the settings for the Web controls.)

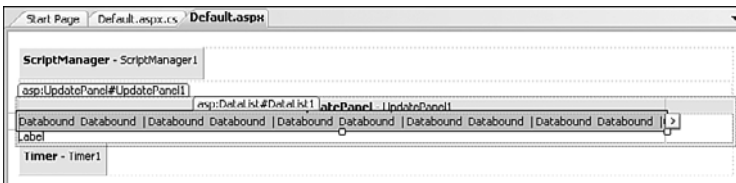


FIGURE 1.4 Just five controls and you have an asynchronous AJAX page.

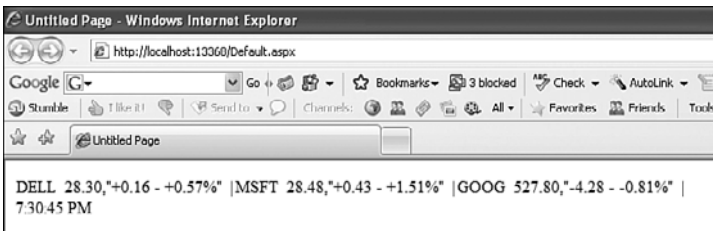


FIGURE 1.5 A very simple design but the code is actually updating the stock prices every 10 seconds with that postback page flicker.

Because of anonymous types, the code to actually query the stock process from Yahoo! is very short (see Listing 1.11).

**LISTING 1.11** This Code Uses `HttpRequest` and `HttpResponse` to Request Stock Quotes from Yahoo!

```
using System;
using System.Data;
using System.Diagnostics;
using System.Configuration;
using System.Collections;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;
using System.Web.Services ;
using System.Net;
using System.IO;
using System.Text;

namespace DataBindingAnonymousTypes
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            Update();
        }

        private void Update()
        {
            var quote1 = new {Stock="DELL", Quote=GetQuote("DELL")};
            var quote2 = new {Stock="MSFT", Quote=GetQuote("MSFT")};
            var quote3 = new {Stock="GOOG", Quote=GetQuote("GOOG")};

            var quotes = new object[] { quote1, quote2, quote3 };
            DataList1.DataSource = quotes;
            DataList1.DataBind();
            Label3.Text = DateTime.Now.ToLongTimeString();
        }

        protected void Timer1_Tick(object sender, EventArgs e)
        {
```

LISTING 1.11 Continued

---

```

        //Update();
    }

    public string GetQuote(string stock)
    {
        try
        {
            return InnerGetQuote(stock);
        }
        catch(Exception ex)
        {
            Debug.WriteLine(ex.Message);
            return "N/A";
        }
    }

    private string InnerGetQuote(string stock)
    {
        string url = @"http://quote.yahoo.com/d/quotes.csv?s={0}&f=pc";
        var request = HttpWebRequest.Create(string.Format(url, stock));

        using(var response = request.GetResponse())
        {
            using(var reader = new StreamReader(response.GetResponseStream(),
                Encoding.ASCII))
            {
                return reader.ReadToEnd();
            }
        }
    }
}

```

---

The method `InnerGetQuote` has a properly formatted uniform resource locator (URL) query for the Yahoo! stock-quoting feature. Next, an `HttpWebRequest` sends the URL query to Yahoo! Then, the `HttpWebResponse`—returned by `request.GetResponse`—is requested and a `StreamReader` reads the response. Easy, right?

All of this code is run by the `Update` method. `Update` creates anonymous types containing a `Stock` and `Quote` field (which are populated by the `GetQuote` and `InnerGetQuote` methods). An anonymous array of these quote objects is created and all of this is bound to the `DataList`. The `DataList` itself has template controls that are data bound to the `Stock` and `Quote` fields of the anonymous type. Figure 1.6 shows the template design of the `DataList`. The very easy binding statement is shown in Figure 1.7.

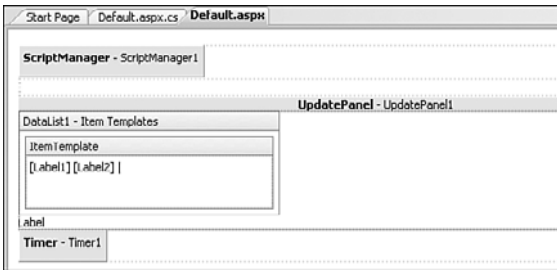


FIGURE 1.6 The template view of the DataList is two Label controls and the `!` character.

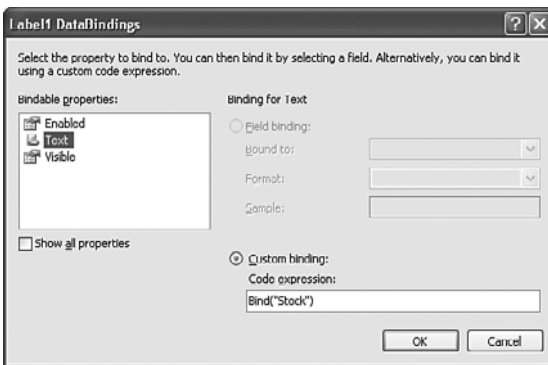


FIGURE 1.7 The binding statements for bound template controls have been very short (as shown) since Visual Studio 2005.

All of the special features, such as template editing and managing bindings, are accessible through the DataList Tasks button, which is shown to the right of the DataList in Figure 1.4. You can also edit elements such as binding statements directly in the ASP designer. Listing 1.12 shows the ASP/HTML for the web page.

LISTING 1.12 The ASP That Creates the Page Shown in Figure 1.4 (Design Time) and Figure 1.5 (Runtime)

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="DataBindingAnonymousTypes._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
```

LISTING 1.12 Continued

---

```

<body>
  <form id="form1" runat="server">
    <asp:ScriptManager ID="ScriptManager1" runat="server">
    </asp:ScriptManager>
    <div>

    </div>
    <asp:UpdatePanel ID="UpdatePanel1" runat="server" EnableViewState="False">
      <ContentTemplate>
        <asp:DataList ID="DataList1" runat="server" RepeatDirection="Horizontal">
          <itemtemplate>
            <asp:Label ID="Label1" runat="server" Text='<%= Bind("Stock") %>'>
              ➤</asp:Label>
            &nbsp;<asp:Label ID="Label2" runat="server" Text='<%= Bind("Quote") %>'>
              ➤</asp:Label>
            &nbsp;<|
          </itemtemplate>
        </asp:DataList>
        <asp:Label ID="Label3" runat="server" Text="Label"></asp:Label>
      </ContentTemplate>
      <triggers>
        <asp:asyncpostbacktrigger ControlID="Timer1" EventName="Tick" />
      </triggers>
    </asp:UpdatePanel>
    <asp:Timer ID="Timer1" runat="server" Interval="10000" ontick="Timer1_Tick">
    </asp:Timer>
  </form>
</body>
</html>

```

---

The really neat thing about this application (besides getting stock quotes) is that the postbacks happen transparently with AJAX. The way AJAX works is that an asynchronous postback happens and all of the code runs except the part that renders the new page data. Instead, text is sent back and JavaScript updates small areas of the page.

The underlying technology for AJAX is an XMLHttpRequest, and this technology in its raw form has been around for a while. But, the raw form required wiring up callbacks and spinning your own JavaScript. You can still handcraft AJAX code of course, but now there are web controls, such as the UpdatePanel and Timer, that take care of the AJAX plumbing for you.

The elements that initiate the AJAX behavior are called triggers. Triggers can really be any postback event. Listing 1.12 uses the AJAX Timer's Tick event. (And, if you want this to actually look like a ticker, play with some styles and add some color.)

## Testing Anonymous Type Equality

Anonymous type equality is defined very deliberately. If any two or more anonymous types have the same order, number, and member declaratory type and name, the same anonymous type class is defined. In this instance, it is permissible to use the referential equality operator on these types. If any of the order, number, and member declarator type and name is different, a different anonymous type definition is defined for each. And, of course, testing referential integrity produces a compiler error.

### NOTE

It is possible to use reflection to get type information about anonymous types, and you might want to do this, occasionally, for anonymous types returned from methods. However, the actual name of the anonymous type can vary between compilations, so devising a way to use the class name probably has no reliable uses.

If you want to test member equality, use the `Equals` method (defined by all objects). Anonymous types with the same order, type, and name, type, and value of member declarators also produce the same hash; the hash is the basis for the equality test. Listing 1.13 provides some samples of anonymous types followed by equality tests and comments indicating those that produce the same anonymous types and those that have member-wise equality.

LISTING 1.13 Various Anonymous Types with Annotations

```
var audioBook = new {Artist="Bob Dylan",  
    Song="I Shall Be Released"}; // anonymous type 1  
var songBook1 = new {Artist="Bob Dylan",  
    Song="I Shall Be Released"}; // also anonymous type 1  
var songBook2 = new {Singer="Bob Dylan",  
    Song="I Shall Be Released"}; // anonymous type 2  
var audioBook1 = new {Song="I Shall Be Released",  
    Artist="Bob Dylan"}; // anonymous type 3  
  
audioBook.Equals(songBook1);           // true everything the same  
audioBook.Equals(songBook2);           // first member declarators different  
songBook1.Equals(songBook2);           // member declarator-names differ  
audioBook1.Equals(audioBook);           // member declarators in different orders
```

The anonymous types `audioBook` and `songBook1` produce the same anonymous type. These are the only two that produce the same hash and, as a result, the `Equals` method returns true. The other anonymous types are similar, but either the member declarators are different—`songBook1` uses the member declarator `Artist` and `songBook2` uses `Singer`—or the order of the declarators are different—referring to `audioBook` and `audioBook1`.



## Using Anonymous Types with LINQ Queries

The most significant attribute of anonymous types in conjunction with LINQ is that they support *hierarchical data shaping* without writing all of the plumbing code or resorting to SQL. Data shaping is roughly transforming data from one composition to another. LINQ lets you do this with natural queries, and anonymous types give you a place to store the results of these queries.

This whole book is about LINQ, so Listing 1.14 shows a couple of LINQ examples without getting too far ahead in upcoming chapter material. Again, each example also has a brief description.

LISTING 1.14 A Couple of Simple LINQ Queries to Play With Demonstrating Future Topics Such as Sorting and Projections

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AnonymousTypeWithQuery
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new int[] {1, 2, 3, 4, 5, 6, 7};
            var all = from n in numbers orderby n descending select n;

            foreach(var n in all)
                Console.WriteLine(n);

            var songs = new string[]{"Let it be", "I shall be released"};
            var newType = from song in songs select new {Title=song};

            foreach(var s in newType)
                Console.WriteLine(s.Title);

            Console.ReadLine();
        }
    }
}
```

---

The first query—from `n in numbers orderby n descending select n`—sorts the integers 1 to 7 in reverse order and stuffs the results in the anonymous type `all`. The second query—from `song in songs select new {Title=song}`—shapes the array of strings in

songs to an enumerable collection of anonymous objects with a property `Title`. (The second example takes an array of strings and shapes it into an array of objects with a well-named property.)

## Introducing Generic Anonymous Methods

For newer programmers, word reuse can be confusing. For example, *anonymous* methods are unrelated to anonymous types except to the extent that it means the type of the method is unnamed. Anonymous methods are covered in this section because they are valuable and worth covering, but, for the most part, this section switches topics.

Anonymous methods behave like regular methods except that they are unnamed. They were introduced as an alternative to defining delegates that did very simple tasks, where full-blown methods amounted to more than just extra typing. Anonymous methods also evolved further into *Lambda Expressions*, which are even shorter (terse) methods. Chapter 5, “Understanding Lambda Expressions and Closures,” delves deeper into the evolution of methods. For now, this section takes an introductory look at anonymous *generic* methods.

An anonymous method is like a regular method but uses the `delegate` keyword, and doesn’t require a name, parameters, or return type. Listing 1.15 shows a regular method (used as a delegate for the `CancelKeyPress` event, Ctrl+C in a console application) and an anonymous delegate that performs the same role.

LISTING 1.15 A Regular Method and Anonymous Method Handling the `CancelKeyPress` Event in a Console Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AnonymousMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            // ctrl+c
            Console.CancelKeyPress += new ConsoleCancelEventHandler
                (Console_CancelKeyPress);

            // anonymous cancel delegate
            Console.CancelKeyPress +=
                delegate
                {
                    Console.WriteLine("Anonymous Cancel pressed");
                };
        }
    }
}
```

LISTING 1.15 Continued

---

```

        Console.ReadLine();

    }

    static void Console_CancelKeyPress(object sender, ConsoleCancelEventArgs e)
    {
        Console.WriteLine("Cancel pressed");
    }
}
}

```

---

**TIP**

To quickly stub out an event-handling method, type the ***object.eventname***, the += operator, and press the Tab key twice.

---

The regular method (used as a delegate) is named `ConsoleCancelEventHandler`. Although the double-Tab trick generates these stubbed delegates for you, they are overkill for one-line event handlers. The second statement that begins with the `Console.CancelKeyPress +=` delegate demonstrates an anonymous method (delegate) that is equivalent to the longer form of the method. Notice that because the parameters in the delegate aren't used, they are omitted from the anonymous delegate. You have the option of using the parameter types and names if they are needed in the delegate.

## Using Anonymous Generic Methods

Delegates are really just methods that are used (mostly) as event handlers. Generic methods are those that have parameterized types. (Think replaceable data types.) Therefore, anonymous generic delegates are anonymous methods that are associated with replaceable parameterized types. A very useful type is `Func<T>` (and `Func<T, T1, ... Tn>`, demonstrated in Listing 1.16). This generic delegate (defined in the `System` namespace) can be assigned to delegates and anonymous delegates with varying return types and parameters, which makes it a very flexible delegate holder.

### LISTING 1.16 Demonstrating How to Use `System.Func` to Define an Essentially Nested Implementation of the Factorial Function

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

```

## LISTING 1.16 Continued

```
namespace AnonymousGenericDelegate
{
    class Program
    {
        static void Main(string[] args)
        {
            System.Func<long, long> Factorial =
                delegate(long n)
                {
                    if(n==1) return 1;
                    long result=1;
                    for(int i=2; i<=n; i++)
                        result *= i;
                    return result;
                };

            Console.WriteLine(Factorial(6));
            Console.ReadLine();
        }
    }
}
```

For all intents and purposes, `Factorial` is a nested function. Listing 1.16 used `Func<long, long>`, where the first `long` parameter represents the return type and the second is the parameter. Notice that the listing also used a named parameter for the anonymous delegate.

## Implementing Nested Recursion

Now, you can have a little fun bending and twisting the `Factorial` function to use recursion. The challenge is that the named delegate is not named until after the delegate definition—the name being `Factorial`. Hence, you can't use the name in the anonymous delegate itself, but you can make it work.

There is a class called `StackFrame`. `StackFrame` permits getting methods (and information from the call stack) and you can use this class and reflection to invoke the anonymous delegate recursively. (This code is obviously esoteric—referred to this as programmer *esoterism*—but it is fun and demonstrates a lot of features of the framework in a little bit of space, as shown in Listing 1.17.)

## LISTING 1.17 Nested, Recursive Anonymous Generic Methods—as a Routine Practice

```
using System;
using System.Diagnostics;
using System.Collections.Generic;
using System.Linq;
```

## LISTING 1.17 Continued

---

```

using System.Text;
using System.Reflection;

namespace AnonymousGenericRecursiveDelegate
{
    class Program
    {
        static void Main(string[] args)
        {
            Func<long, long> Factorial =
                delegate(long n)
                {
                    return n > 1 ?
                        n * (long)(new StackTrace()
                            .GetFrame(0).GetMethod().Invoke(null, new object[] { n-1 }))
                        : n;
                };

            Console.WriteLine(Factorial(6));
            Console.ReadLine();
        }
    }
}

```

---

Again, writing code like the `Factorial` delegate in Listing 1.17 is only fun for the writer, but elements of it do have utility. For example, anonymous delegates like the `Factorial` can be useful for one-time, simple event handling. Assigning behaviors to the `Func<T>` delegate type effectively makes nested functions and reusable delegates that can be passed as arguments, a very dynamic way to program. Getting the `StackFrame` can be a great way to create a utility that tracks function calls during debugging—like writing the `StackTrace` to the Debug window in a way that is useful to you—and reflection has many uses.

Reflection can be useful for dynamically loaded assemblies, as demonstrated by NUnit and Visual Studio's unit testing.

## Summary

This chapter examined anonymous types in detail. Anonymous types are strong types where the compiler does the work of figuring out the actual type and writing the class implementation, if the anonymous type is a composite type.

As you see anonymous types used throughout the book for query results, remember anonymous types are immutable, the same type is code generated if the member declaratory—field name—type, number, and order are identical.

# Index

## NUMBERS

**101 LINQ Samples by Microsoft website, 203**

## A

**Action delegate, Lambda Expressions, 104-106**

**Active Directory, 243**

defining as data source, 248-252

helper attributes, 257-259

IQueryable provider

creating, 245-246

Smet, Bart De implementation, 245

IQueryProvider interface, implementing, 246-248

LINQ query conversions, 252-254

property assignments, 257

querying, 243-244, 260-262

RFCs, 244

schema entities, defining, 259-260

search filters, creating from Where Lambda Expressions, 254-256

**Active Directory Services Interfaces, 259**

**Add New Item template dialog, 488**

**ADO.NET**

- Entity Framework, 383
  - conceptual data models, 385
  - downloading, 387
- Entity Data Models (EDMs), 386
- Go Live estimation date, 388
- relational database solutions, 385
- StockHistory database, 401-411
  - web resources, 387-388
- objects, filling, 377

**ADO.NET 2.0**

- obtaining stock quotes, updating the database, 397-401
- StockHistory database
  - complete script, 394-397
  - defining, 389-390
  - foreign keys, adding, 393-394
  - quotes, adding, 390-392

**ADSI (Active Directory Services Interfaces), 259****Aggregate method, 151-153****aggregate operations**

- aggregation, 151-153
- averages, 154-157
- finding minimum and maximum elements, 157-159
- median values, 163-165
- overview, 151
- summing query results, 162-163

**aggregation, 151-153****AJAX, 22****All, 124-125****Allow access dialog (Outlook), 242****annotating nodes, 433-434****anonymous methods**

- CancelKeyPress events, 25
- delegate keyword, 25
- generic, 26-27
- nested recursion, 27-28
- regular method comparisons, 25

**anonymous types**

- arrays, initializing, 7-8
- composite
  - behaviors, adding, 9
  - creating, 9
  - methods, adding, 10-12
- databinding, 18, 22
  - binding statement, 20
  - elements, editing, 21
  - requesting stock quotes from Yahoo!, 19-20
- defining, 7
- equality, testing, 23
- hierarchical data shaping support, 24
- indexes in for statements, 12-14
- IntelliSense support, 6
- LINQ query examples, 24-25
- object initialization, 34-36
- overview, 6-7
- returning from functions, 17-18
- using statements, 14-16
- var keyword, 5

**Any, 124-125****API methods for raw device contexts, 201**

**arrays**

- anonymous types, initializing, 7-8
- Blackjack game, shuffling a deck of cards, 196-199
- initial capping words, 202-203
- select indexes, shuffling/unsorting, 194-195
- ToArray conversion operator, 51-53

**ascending order, sorting information in, 138-139**

**AsEnumerable conversion operator, 55-56**

**AsEnumerable method, 278-280**

**assigning Lambda Expressions, predefined generic delegates, 101**

**AssociationAttribute, 301**

**associations, adding to databases, 402**

**attributes**

- Active Directory helper, 257-259
- AssociationAttribute, 301
- InheritanceMappingAttribute, 295
- XElement class
  - adding to, 422
  - deleting from, 423
- XML documents, querying, 420-421

**auto-implemented properties, 34**

**automatic properties**

- creating custom objects with, 169
- Lambda Expressions, 102-103

**AutoSync property (ColumnAttribute class), 272**

**Average method, 154-157**

**averages, computing, 157**

- average file size, 154
- average length of words, 156
- simple averages, 154

**B**

**behaviors, adding to anonymous composite types, 9**

**Bill Blogs in C# website, 203**

**BinaryTree class, yield return, 89-93**

**binding control events to Lambda Expressions, 109-110**

**Blackjack game, 195-199**

**Bonaparte, Napoleon, 137**

**broadcast-listeners, 51-53**

**business objects, returning, 190-193**

**C**

**C#, Active Directory queries, 243-244**

**calculated values, projecting new types, 200**

**calling user-defined functions, 363-366**

**CanBeNull property (ColumnAttribute class), 272**

**Cartesian joins, 228**

**CAS (code access security), 205**

**Cast conversion operator, 54-55**

**ChangeConflicts collection, 372**

**child elements, LINQ to XML and XPath comparison, 441**

**classes**

- closures, 117-119
- ColoredPoint, 32-33
- ColoredPointList, 37
- ColumnAttribute, properties, 269-273
- Customer, 460



- DataContext, 191
- DataContext, 305, 356-357
- DirectoryAttributeAttribute, 258-259
- DirectoryQuery, 252-254
- DirectorySchemaAttribute, 257-258
- EntitySet, adding as properties, 300-305
- EventLog, 206
- Hypergraph, 39-46
- implementing, compound type initialization, 32-34
- IOrderedQueryable, 248-252
- LINQ to SQL Class designer, 285-286
- LINQ to SQL Object Relational Designer generated, customizing, 299-300
- MAPIFolder, 242
- mapping to tables, 269-272
- Person, 36
- StackFrame, 27
- StockHistoryEntities, 405
- Supplier, 190
- TransactionScope, 366
- WebClient, 400
- XDocument, loading XML documents, 416-419
- XElement
  - adding/deleting attributes, 422-423
  - node annotations, 433-434
  - XML documents, loading, 420
- XmlWriter
  - overview, 464-465
  - XML files, creating, 465-467

## clauses

- from, joins, 211-212
- group by, 145-149
- let, XML intermediate values, 432-433
- Where, XML documents, 429

## closures, 117-119

### code, compiling Lambda Expressions as

- assigned to Expression<TDelegate> instance emits IL example, 114, 116
- expression tree exploration, 116-117

### code access security (CAS), 205

### CodeDOM, 485

## collections

- ChangeConflicts, 372
- initializing, 36, 39

### ColoredPoint class, 32-33

### ColoredPointList class, 37

### ColumnAttribute class, 269-273

### columns, ignoring for conflict checks, 371

### COM (Component Object Model), 239

### comma-delimited text files, 453

### CompareTo method, 159

### compiling Lambda Expressions as code/data

- assigned to Expression<TDelegate> instance emits IL example, 114, 116
- expression tree exploration example, 116-117

### Component Object Model (COM), 239

### composite anonymous types

- behaviors, adding, 9
- creating, 9
- methods, adding, 10-12

**composite keys, defining joins, 237**

**composite resultsets, creating, 182-183**

**compound initialization**

anonymous types, 34-36

collections, 36, 39

named types, 31

auto-implemented properties, 34

classes, implementing, 32-34

default constructor and property assignment, 30

purpose-defined constructor, 30

**Concat, extension methods, 132-133**

**concurrency conflicts, 368**

catching

comparing member conflict states, 373-374

entities/tables associated with conflict, 372-373

ignoring columns for conflict checks, 371

retrieving conflict information, 372

handling, SubmitChanges method, 369-371

resolving, 375-376

**concurrency control, 368**

**Configure Behavior dialog, 362**

**conflicts**

catching

conflict information, retrieving, 372

entities/tables associated with conflict, 372-373

ignoring columns for conflict checks, 371

member conflict states, comparing, 373-374

concurrency conflicts, 368

concurrency control, 368

handling, SubmitChanges method, 369-371

optimistic concurrency, 368

pessimistic concurrency, 368

resolving, 375-376

**console applications, creating LINQ to XSD Preview, 487**

**contacts (Outlook), adding email addresses, 240-241**

**control events, binding to Lambda Expressions, 109-110**

**conversion operators**

AsEnumerable, 55-56

Cast, 54-55

OfType, 54

ToArray, 51-53

ToDictionary, 57-58

ToList, 56-57

ToLookup, 58-59

**converting CSV files to XML, 454-456**

**Count method, 157**

**counting elements, 157**

**"Creating Project Templates in .NET" website, 487**

**cross joins, implementing, 228, 236**

Northwind database customers and products example, 229-231

SQL as LINQ query example, 231-236

**CSV files (comma-delimited text files), XML**

converting to, 454-456

creating from, 457-458

**currying, 119-120**

**Customer class, 460****customizing**

- joins, 214
  - defining, 215-218
  - multiple predicates, 219-220
  - temporary range variables, 220-223
- LINQ to SQL Object Relational Designer
  - generated classes, 299-300
- objects, instantiating, 170-171
- select statement predicates, 190

**D****data**

- compiling Lambda Expressions as
  - assigned to Expression<TDelegate>
    - instance emits IL example, 114-116
  - expression tree exploration, 116-117
- LINQ to SQL
  - adding to, 349-352
  - deleting from, 352-354
  - updating in, 354-355

**data access layers, writing, 384****DataAccess class, 191****databases**

- creating with LINQ to SQL, 305-307
- relational model
  - C# programming problems, 384-385
  - data access layers, 384
  - Entity Framework solution, 385

**StockHistory (ADO.NET 2.0)**

- complete script, 394-397
- defining, 389-390
- foreign keys, adding, 393-394
- obtaining stock quotes, updating the
  - database, 397-401
- quotes, adding, 390-392

**StockHistory (ADO.NET Entity Framework)**

- associations, adding, 402
- creating EDMs, 401-402
- LINQ to XML and LINQ to Entities,
  - 407-411
- querying EDMs with Entity SQL,
  - 402-405
- querying EDMs with LINQ to Entities,
  - 405-406

**databinding**

- anonymous types, 18
- binding statement, 20
- elements, editing, 21
- requesting stock quotes from Yahoo!,
  - 19-20
- bindability, 345
- IEnumerable interface, 345
- listing example, 345-347

**DataContext class, 305, 356-357****DataContext object, 275-277****DataContextMapping property, 372****DataSets**

- DataTables
  - querying with Where clause, 280-281
  - selecting data from, 278-280
  - sorting against, 282

- joins, defining with, 282-284
- LINQ to DataSets
  - equijoins, 310-312
  - left outer joins, 313-315
  - nonequijoins, 312-313
  - right joins, 315-317
- overview, 277-278
- partitioning methods, 282
- DbType property (ColumnAttribute class), 272**
- de Gaulle, Charles, 151**
- debugging**
  - stored procedures, 392
  - XSLT documents, 450
- Decorator Structural pattern, 61**
- DefaultIfEmpty method, 127, 331**
- defining**
  - Active Directory schema entities, 259-260
  - anonymous types, 7
  - exclusive sets, 177-181
  - generic extension methods, 69-70, 73
  - joins
    - based on composite keys, 237
    - cross joins, 228-236
    - group joins, 224-226
    - left outer joins, 226-228
    - with DataSets, 282-284
  - nonequijoins, 215-218
  - multiple predicates, 219-220
  - temporary range variables, 220-223
  - partial methods, 79-84
  - stored procedures, 358-360
  - tables, 266-269
  - XML as strings, 424-425
- delegate keyword, anonymous methods, 25**
- delegates, Lambda Expressions**
  - Action, 104-106
  - Predicate<T>, 108-109
- deleting**
  - attributes, XElement class, 423
  - data, LINQ to SQL, 352-354
- descending keyword, 138-140**
- descending order, sorting information in, 139-141**
- design goals, LINQ to XSD, 486**
- dictionaries, list conversions, 57-58**
- DirectoryAttributeAttribute class, 258-259**
- DirectoryQuery class, 252-254**
- DirectorySchemaAttribute class, 257-258**
- distinct elements**
  - customer objects
    - defining Order object, 169
    - instantiating, 170-171
  - distinct lists of cities, sorting/returning, 173-177
  - finding, 167
  - IEqualityComparer interface, implementing, 171-172
  - median grade, determining from list of numbers, 167-168
  - object dumper, implementing, 172-173
- Distinct method, 167**

**DLLs (Dynamic Link Libraries),  
importing, 200****documents**

## XML documents

- creating from Yahoo! stock quotes, 426-427
- defining as strings, 424-425
- element navigation based on context, 430-431
- filtering, 429
- functional construction, 450-451
- intermediate values, 432-433
- loading, 415-416
- missing data, 425-426
- namespaces, 427-428
- nested queries, 428-429
- node annotations, 433-434
- querying, 416-421
- sorting, 431

## XSLT documents, debugging, 450

**downloading**

- Entity Framework, 387-388
- LINQ to XSD, 487

**duct typing, 64****Dump, overloading extension methods,  
62-63, 67-68****dynamic programming, Lambda Expressions**

- code/data, compiling as, 114-117
- OrderBy<T> method, 113
- Select<T> method, 110-112
- Where<T> method, 112-113

**E****EDMs (Entity Data Models), 386**

- creating, 401-402
- querying
  - Entity SQL, 402-405
  - LINQ to Entities, 405-406

**element operations, 131-132****elements**

- child elements, LINQ to XML and XPath comparison, 441
- counting, aggregate operations, 157
- distinct elements
  - defining Order object in customer objects, 169
  - determining median grade from list of numbers, 167-168
  - finding, 167
  - implementing IEqualityComparer interface, 171-172
  - implementing object Dumper, 172-173
  - instantiating customer objects, 170-171
  - sorting/returning distinct lists of cities, 173-177
- editing, databinding, 21
- filtering, LINQ to XML and XPath comparison, 442-443
- maximum elements, finding, 157-159
- minimum elements, finding, 157-159
- navigation based on context, XML documents, 430-431

obtaining specific elements from sequences, 131-132

sibling elements, LINQ to XML and XPath comparison, 442

**ElementsAfterSelf method, 430-431**

**email addresses, adding to Outlook contacts, 240-241**

**embedded LINQ queries, XML with, 458**

console application, 460-461

Customer class example, 459-460

literal XML with embedded expressions and LINQ, 461-462

**Empty, 127**

**entities**

Active Directory schema, defining, 259-260

associated with conflict, 372-373

LINQ to Entities

EDMs, querying, 405-406

StockHistory database, UpdatePriceHistory method, 407-411

nullable, 290-293

**Entity Data Models, 386**

**Entity Framework (ADO.NET), 383**

conceptual data models, 385

downloading, 387

EDMs, 386

Go Live estimation date, 388

relational database solutions, 385

StockHistory database

adding associations, 402

creating EDMs, 401-402

LINQ to XML and LINQ to Entities, 407-411

querying EDMs with Entity SQL, 402-405

querying EDMs with LINQ to Entities, 405-406

web resources

Entity SQL blog, 387

samples, 388

Wikipedia, 387

**Entity SQL**

blog, 387

EDMs, querying, 402-405

website, 405

**EntitySet classes, adding as properties, 300-305**

**equality testing, 23, 129-130**

**Equals method, anonymous types, 23**

**equijoins, 214**

LINQ to Datasets, 310-312

LINQ to SQL, 317-321

**esoterism, 27**

**Euclidean algorithm example, 186-189**

**EventLog class, 206**

**Except method, 177-181**

**exclusive sets, defining, 177-181**

**Expression property (ColumnAttribute class), 272**

**expression trees, 116**

**expressions**

Lambda

assigning to predefined generic delegates, 101

automatic properties, 102-103

capturing as generic actions, 104-106

- capturing as generic predicates, 108-109
- control events, binding, 109-110
- currying, 119-120
- delegate role listing, 100
- reading, 103-104
- string searches, 106-107
- Where, 254-256

- regular, adding to XML Schema files, 491-494

### **Extensible Stylesheet Language Transformations, 437**

#### **extension methods, 61-63, 151**

- Concat, 132-133
- defining generic extension methods, 69-70, 73
- defining with return type, 64-65
- implementing, 64-67
- LINQ, 73-77
- overloading, 67-68
- SequenceEqual, 130
- “talking” string extension methods, 78-79
- uses for, 63-64
- Where, 73, 76

## **F**

**Feynman, Richard, 179**

**Fibonacci numbers, 8, 177**

**Field method, 280-281**

### **filtering**

- elements, LINQ to XML and XPath comparison, 442-443
- information, 122-124
- OfType filters, 122-124
- XML documents, 429

### **finding**

- distinct elements, 167
- defining customer Order object, 169
- determining median grade from list of numbers, 167-168
- implementing IEqualityComparer interface, 171-172
- implementing object Dumper, 172-173
- instantiating custom objects, 170-171
- sorting and returning distinct list of cities, 173-177
- minimum and maximum elements, 157-159

### **for statements, anonymous type indexes, 12-14**

**foreign keys, adding to databases, 393-394**

**from clauses, joins, 211-212**

### **function pointers, listings**

- anonymous delegate, 99
- delegates in C#, 99
- FunctionPointer definition, 98
- Lambda Expression playing the delegate role, 100

**functional construction, 443, 450-451**

### **functions**

- anonymous types, returning, 17-18
- ProductsUnderThisUnitPrice, 363-366
- user-defined, calling, 363-366

## G

**GDI+, API methods for raw device contexts, 201**

**generation operations**

DefaultIfEmpty, 127

Empty, 127

Range, 127

Repeat, 128-129

**generic anonymous methods, 26-27**

**generic extension methods, defining, 69-70, 73**

**GetData method, 476**

**GetPoints method, 38**

**group by clause, 145-149**

**group joins**

defining, 224-226

LINQ to SQL, 321-331

**grouping information, 145-150**

**GroupJoin method, 321-331**

## H

**helper attributes, Active Directory, 257-259**

**Hypergraph class, 39-46**

broadcast-listener, 53

ColoredPoint class, 32-33

compound type initialization of objects

default constructor and property assignment, 30

Paint Event handler, 31

Pen object, 30

HypergraphController user control, 47-50

IHypergraph interface, 46-47

images, saving to files, 51-52

subject and observer interfaces, 50

## I

**IBindingList interface, databinding, 345**

**IComparable interface, 159**

**IDataReader interface methods, 215-218**

**IEnumerable{T}, 94**

**IEnumerable interface**

AsEnumerable conversion operator, 55

databinding, 345

**IEqualityComparer interface, implementing, 171-172**

**IHypergraph interface, 46-47**

**IL (Intermediate Language), 7**

**ILDASM (Intermediate Language Disassembler), 7**

**importing DLLs, 200**

**Inbox (Outlook), reading, 240-241**

**indexes**

anonymous type, for statements, 12-14

select, shuffling/unsorting arrays, 194-195

SelectMany method, 207

SelectMany methods, 208

**information filtering, 122-124**

**inheritance hierarchies, LINQ to SQL**

Object Relational Designer, creating with, 298

single-table mapping, 294-298



**InheritanceMappingAttribute, 295**

**initial capping words (arrays), 202-203**

**initializing**

- anonymous type arrays, 7-8
- collections, 36, 39
- objects with
  - anonymous types, 34-36
  - named types, 30-34

**inner joins, 213-214**

**InnerGetQuote method, 20**

**InsertCustomer methods, 362-363**

**InsertQuote stored procedure, 390-392**

**IntelliSense, anonymous types support, 6**

**interfaces**

- IBindingList, databinding, 345
- IComparable, 159
- IDataReader, methods, 215-218
- IEnumerable
  - AsEnumerable conversion operator, 55
  - databinding, 345
- IEqualityComparer, implementing, 171-172
- IHypergraph, 46-47
- IQueryProvider, implementing, 246-248
- projecting interfaces, support for, 159-161

**Intermediate Language, 7**

**Intermediate Language Disassembler, 7**

**intermediate values, XML documents, 432-433**

**Intersect method, 177-181**

**IOrderedQueryable class, 248-252**

**IQueryable, 73**

**IQueryable provider**

- creating, 245-246
- Smet, Bart De implementation, 245

**IQueryProvider interface, implementing, 246-248**

**IsDbGenerated property (ColumnAttribute class), 273**

**IsDiscriminator property (ColumnAttribute class), 273**

**IsPrimaryKey property (ColumnAttribute class), 273**

**IsVersion property (ColumnAttribute class), 273**

## J

**joins**

- based on composite keys, 237
- cross
  - implementing, 228, 236
  - Northwind database customers and products example, 229-231
  - SQL as LINQ query example, 231-236
- DataSets, defining with, 282-284
- equijoins, 214
  - LINQ to DataSets, 310-312
  - LINQ to SQL, 317-321
- group
  - defining, 224-226
  - LINQ to SQL, 321-331
- inner, 213-214

- left, LINQ to SQL, 331-340
- left outer, 224
  - implementing, 226-228
  - LINQ to DataSets, 313-315
- multiple from clauses, 211-212
- nonequijoins, 214
  - defining, 215-218
  - LINQ to DataSets, 312-313
  - multiple predicates, 219-220
  - temporary range variables, 220-223
- right joins, LINQ to DataSets, 315-317

## K–L

**Kernighan, Brian, 98**

### keys

- composite, 237
- foreign, adding to databases, 393-394

### keywords

- delegate, anonymous, 25
- descending, 138-140
- orderby, 137
- var, anonymous types, 5

### Lambda Expressions

- automatic properties, 102-103
- capturing as
  - generic actions, 104-106
  - generic predicates, 108-109
- closures, 117-119
- control events, binding, 109-110
- currying, 119-120

- delegate role listing, 100
- dynamic programming
  - compiling as code/data, 114-117
  - OrderBy<T> method, 113
  - Select<T> method, 110-112
  - Where<T> method, 112-113
- predefined generic delegates, assigning to, 101
- reading, 103-104
- string searches, 106-107
- Where, converting to Active Directory search filters, 254-256

### LDAP (Lightweight Directory Access Protocol), 244

**left joins, LINQ to SQL, 331-340**

**left outer joins, 224**

- implementing, 226-228
- LINQ to Datasets, 313-315

**Leonardo of Pisa, 177**

**let clause, XML intermediate values, 432-433**

**LINQ (Language INtegrated Query), 121**

- constructing queries, 122
- equality testing, 129-130
- extension methods, 73-77

**LINQ to DataSets, joins**

- equijoins, 310-312
- left outer joins, 313-315
- nonequijoins, 312-313
- right joins, 315-317

**LINQ to Entities**

- EDMs, querying, 405-406
- StockHistory database
  - UpdatePriceHistory method, 407-411

**LINQ to SQL**

## data

- adding, 349-352
- deleting, 352-354
- updating, 354-355

## databases, creating, 305-307

## databinding

- bindability, 345
- IEnumerable interface, 345
- listing example, 345-347

## inheritance hierarchies

- creating with Object Relational Designer, 298
- single-table mapping, 294-298

## joins

- equijoins, 317-321
- group, 321-331
- left, 331-340

## n-tier applications, 376

- client with reference to the service, 380-381
- service contract for serializing Customer objects, 377-379
- service contract, implementing, 379
- WCF middle tier, 377

## Object Relational Designer generated classes, customizing, 299-300

## views, querying, 342-344

## Visual Designer, mapping stored procedures, 360-363

**LINQ to SQL Class designer, 285-286****LINQ to XML**

- node annotations, 433-434
- StockHistory database
- UpdatePriceHistory method, 407-411

## XML documents

- creating from Yahoo! stock quotes, 426-427
- element navigation based on context, 430-431
- filtering, 429
- intermediate values, 432-433
- namespaces, 427-428
- nested queries, 428-429
- sorting, 431

## XPath, compared, 438

- child elements, 441
- filtering elements, 442-443
- namespaces, 439-441
- sibling elements, 442

## XSLT, compared, 443

- debugging XSLT documents, 450
- HTML documents, 444-449

**LINQ to XSD**

- design goals, 486
- downloading/installing, 487
- object queries, 496-498
- overview, 485
- Preview console applications, creating, 487
- regular expressions added to XML Schema files, 491-494
- XML files, defining, 488-490
- XML Schema files, defining, 490-491

**listings**

## Active Directory

- DirectorySchemaAttribute class, 257
- LINQ query conversions to Active Directory queries, 253-254
- property assignments, 257

- querying, 260-262
  - schema entities, 259-260
  - search filters created with Where  
Lambda Expressions query, 254-256
- Active Directory queries with straight  
C# code, 243-244
- anonymous methods handling  
CancelKeyPress event, 25
- anonymous types
  - adding behaviors to, 10
  - equality testing, 23
  - indexes in for statements, 12-13
  - initializing, 7, 35-36
  - returning from functions, 17
  - using statements, 14-15
- AsEnumerable conversion operator, 55
- ASP for AJAX page, 21
- behaviors, adding to anonymous type, 10
- Blackjack game
  - jack namespace, 439
  - shuffling a deck of cards, 196-199
  - statistics saved to XML file, 438
- Cast conversion operator, 54
- composite anonymous types, 9
- concurrency conflicts
  - comparing member conflict  
states, 373
  - conflict information, retrieving, 372
  - entities/tables associated with  
conflict, 372
  - handling, 369
  - ignoring columns for conflict  
checks, 371
  - resolving, 375
- data
  - adding, 350-352
  - deleting, 352-354
  - updating, 354
- databinding with LINQ to SQL, 345-347
- DataContext class, 356-357
- DirectoryAttributeAttribute class, 258-259
- EntitySet classes as properties, adding,  
301-305
- function pointers
  - anonymous delegate, 99
  - delegates in C#, 99
  - FunctionPointer definition, 98
  - Lambda Expression playing the  
delegate role, 100
- functional construction, 451
- GDI API methods for raw device  
contexts, 201
- generic anonymous methods, 26
- Hypergraph
  - broadcast-listener, 53
  - ColoredPoint class, 32-33
  - ColoredPointList class, 37
  - default constructor and property  
assignment, 30
  - Hypergraph class, 39-46
  - HypergraphController user control,  
47-50
  - IHypergraph interface, 46-47
  - named types, 31
  - purpose-defined constructor, 30
  - saving images to files, 51-52
  - subject and observer interfaces, 50
- IOrderedQueryable class, 249-252

IQueryProvider interface, 246

joins

- based on composite keys, 237
- cross join for Northwind database customers and products, 229-231
- cross join SQL as LINQ query, 231-236
- group joins, 224-226
- inner, 213-214
- left outer joins, 227
- multiple from clauses, 211
- nonequijoins with multiple predicates, 219
- nonequijoins with temporary range variables, 220-223
- nonequijoins, defining, 215-218

Lambda Expressions

- assigned to Expression<TDelegate> instance emits IL example, 114-116
- assigning to predefined generic delegates, 101
- automatic properties, 102
- capturing as generic actions, 104-105
- capturing as generic predicates, 108
- closures, 118
- control events, binding, 109
- currying, 119
- demonstrating explicit argument types, 103
- expression tree exploration, 116-117
- OrderBy<T> method, 113
- Select<T> method, 110-111
- string searches, 106-107
- Where<T> method, 112

LINQ to DataSets

- equijoins, 310-311
- left outer joins, 314-315
- nonequijoins, 312-313
- right outer joins, 316

LINQ to SQL

- creating databases, 305
- customizing Object Relational Designer generated classes, 299-300
- equijoins, 317-321
- group joins, 321-331
- inheritance hierarchies, 294-298
- left joins, 331-340
- regular expressions added to XML Schema files, 492-494

LINQ to XML and XPath comparison

- child elements, 441
- filtering elements, 442
- namespaces, 440
- sibling elements, 442

LINQ to XML and XSLT comparison, HTML documents, 444-449

LINQ to XSD

- queries, 496-497
- XML files, creating, 488-490
- XML Schema files, creating, 490

n-tier applications with LINQ to SQL

- client with reference to the service, 380-381
- service contract for serializing Customer objects, 377-379
- service contract, implementing, 379

nested recursive anonymous generic methods, 27

- nullable type entities, 290-293
- OfType conversion operator, 54
- Outlook
  - updating contacts, 240-241
  - Inbox, 240-241
- PetCemetery.XML file, 417-419
- query examples with anonymous types, 24
- requesting stock quotes from Yahoo!, 19-20
- select statements
  - customizing predicates, 190
  - function call effects, 186-189
  - indexes for shuffling/unsorting arrays, 194-195
  - initial capping words in arrays, 202-203
  - projecting types, 203
  - returning custom business objects, 191-193
- SelectMany methods
  - comparing Windows Registry sections, 206-207
  - indexes, 207
  - projecting types, 203
- SQL to XML conversions, 473-475
  - Northwind DataContext example, 472
  - Northwind object-relational map example, 471-472
  - TreeView output of XML document, 476
- SQL updates from XML
  - examining inserted data, 481
  - inserting data, 480-481
  - osql.exe scripting output, 482-483
  - sample XML file, 478-479
- StockHistory database
  - adding quotes, 390-392
  - Company table, 390
  - complete script, 394-395, 397
  - foreign keys, adding, 394
  - LINQ to XML and LINQ to Entities, 407-411
  - obtaining stock quotes to update the database, 397-401
  - PriceHistory table, 390
  - querying EDMs with Entity SQL, 404-405
  - querying EDMs with LINQ to Entities, 405
- stored procedures
  - defining, 358-360
  - mapping with LINQ to SQL Visual Designer, 362-363
  - UpdateCustomer example, 357-358
- ToDictionary conversion operator, 57
- ToList conversion operator, 56
- ToLookup conversion operator, 58
- transactions, deleting parent/child rows, 366-368
- user-defined functions, calling, 363-365
- views
  - building with SQL Server, 342
  - querying with LINQ to SQL, 342-344
- XElement class, adding/deleting attributes, 422-423
- XML
  - creating from CSV files, 454-456
  - defining as strings, 424
  - missing data, 425-426
  - text files, creating, 457-458

**XML documents**

- creating from Yahoo! stock quotes, 426-427
- element navigation based on context, 430-431
- filtering, 429
- intermediate values, 432-433
- namespaces, 427-428
- nested queries, 428-429
- node annotations, 433-434
- querying, 416-417, 420-421
- sorting, 431

**XML with embedded LINQ queries in VB**

- console application, 460-461
- Customer class example, 459-460
- literal XML with embedded expressions and LINQ, 461-462

XmlWriter class for creating XML files, 465-467

**lists, converting**

- dictionaries, to, 57-58
- query results to, 56-57

**literal XML in VB with embedded expressions and LINQ, 461-462****LongCount method, 157****lookups, IEnumerable object conversions, 58-59****luncheon menu example**

- luncheon days collection and regular expression incorporation, 497
- possible weekdays XML document, 492-494
- XML file, 488
- XML Schema file, 490

**M****MAPIFolder class, 242****mapping**

- classes to tables, 269-272
- LINQ to SQL inheritance hierarchies
  - creating with Object Relational Designer, 298
  - single-table mappings, 294-298
- stored procedures, LINQ to SQL Visual Designer, 360-363

**Max method, 157-159****maximum elements, finding, 157-159****McCarthy, Dan, 177****Median method, 163-165****median grade, determining from list of numbers, 167-168****median values, 163-165****member conflict states, comparing, 373-374****methods**

- Aggregate, 151-153
- anonymous composite types, adding to, 10-12
- anonymous methods
  - CancelKeyPress events, 25
  - delegate keyword, 25
  - generic, 26-27
  - nested recursion, 27-28
  - regular method comparisons, 25
- API for raw device contexts, 201
- AsEnumerable, 278-280
- Average, 154-157
- CompareTo, 159

- Count, 157
- DefaultIfEmpty, 331
- Distinct, 167
- ElementsAfterSelf, 430-431
- Equals, anonymous types, 23
- Except, 177-181
- extension methods, 61-63, 151
  - Concat, 132-133
  - defining generic extension methods, 69-70, 73
  - defining with return type, 64-65
  - implementing, 64-67
  - LINQ, 73-77
  - overloading, 67-68
  - SequenceEqual, 130
  - “talking” string extension methods, 78-79
  - uses for, 63-64
  - Where, 73, 76
- Field, 280-281
- GetData, 476
- GetPoints, 38
- GroupJoin, 321-331
- IDataReader interface, 215-218
- InnerGetQuote, 20
- InsertCustomer, 362-363
- Intersect, 177-181
- LongCount, 157
- Max, 157-159
- Median, 163-165
- Min, 157-159
- ObjectChangeConflict.Resolve, 375

- OrderBy<T>, Lambda Expressions, 113
- OrderByDescending, 140
- partial methods, 79-84
- partitioning methods, 282
- ReadSuppliers, 191
- Reverse, 144-145
- Select<T>, Lambda Expressions, 110-112
- SelectMany
  - indexes, 207-208
  - types, projecting, 203-205
  - Windows Registry sections, comparing, 206-207
- SubmitChanges, 369-371
- Sum, 162-163
- ThenBy, 138
- ThenByDescending, 141
- ToLookup, 150
- Union, 182-183
- Update, databinding anonymous types, 20
- UpdatePriceHistory, 400
- Where<T>, Lambda Expressions, 112-113
- Microsoft Intermediate Language, 7**
- Microsoft XML Team WebLog website, 487**
- Min method, 157-159**
- minimum elements, finding, 157-159**
- missing data (XML), 425-426**
- MSIL (Microsoft Intermediate Language), 7**
- MyPoint property, 34**



## N

### **n-tier applications, 376**

- client with reference to the service, 380-381
- service contracts
  - implementing, 379
  - serializing Customer objects, 377-379
- WCF middle tier, 377

### **Name property (ColumnAttribute class), 273**

### **named types, object initialization, 31**

- auto-implemented properties, 34
- classes, implementing, 32-34
- default constructor and property assignment, 30
- purpose-defined constructor, 30

### **namespaces**

- LINQ to XML and XPath comparison, 439-441
- XML documents, 427-428

### **nanotechnology, 179**

### **Napoleon, 137**

### **nested queries, XML documents, 428-429**

### **nested recursive anonymous generic methods, 27-28**

### **New Association dialog, 402**

### **nodes**

- annotations, 433-434
- XComment, SQL to XML conversions, 475

### **nonequijoins, 214**

- defining, 215-218
- LINQ to Datasets, 312-313

- multiple predicates, 219-220
- temporary range variables, 220-223

### **Northwind Customers table object-relational map, 472**

### **Northwind database**

- cross join of customers and products, 229-231

### **customers**

- adding, 350-352
- deleting, 352-354, 366-368
- table object-relational map, 471

### **Customers table object-relational map, 471**

### **data, updating, 354**

### **DataContext example, 472**

### **examining inserted data, 481**

### **InsertCustomer methods, 362-363**

### **inserting data, 480-481**

### **new customers XML file, 478-479**

### **orders, deleting, 366-368**

### **osql.exe scripting output, 482-483**

### **ProductsUnderThisUnitPrice function, 363-366**

### **stored procedure for CustomerIDs, 358-360**

### **UpdateCustomer stored procedures, 357-358**

### **views**

### **Orders/Order Details tables, 342**

### **querying, 342-344**

### **nullable types, 289-293**

## O

### **Object Relational Designer, LINQ to SQL**

- generated classes, customizing, 299-300
- inheritance hierarchies, 298

### **object-relational maps, XML conversions, 470-472**

### **ObjectChangeConflict.Resolve method, 375**

### **objects**

- ADO.NET, filling with, 377
- compound initialization with anonymous types, 34-36
- compound initialization with named types, 31
  - auto-implemented properties, 34
  - classes, implementing, 32-34
  - default constructor and property type, 30
  - purpose-defined constructor, 30
- custom business, returning, 190-193
- custom objects, instantiating, 170-171
- DataContext, 275-277
- LINQ to XML queries, 496-498
- object dumper, implementing, 172-173
- Order, defining, 169
- tables
  - defining, 266-269
  - mapping classes to, 269-272
- XNamespace, 427-428

### **OfType conversion operator, 54**

### **OfType filter, 122-124**

### **operations**

- element operations, 131-132
- generation operations
  - DefaultIfEmpty, 127
  - Empty, 127
  - Range, 127
  - Repeat, 128-129

### **optimistic concurrency, 368**

### **Order object, defining, 169**

### **orderby keyword, 137**

### **OrderBy<T> method, Lambda Expression, 113**

### **OrderByDescending method, 140**

### **osql.exe command line, examining inserted data, 482-483**

### **Outlook**

- Allow access dialog, 242
- contacts, adding email addresses, 240-241
- Inbox/contacts, reading, 240-241
- instances, creating, 242

### **overloading extension methods, 67-68**

## P

### **partial methods, defining, 79-84**

### **partitioning, 282**

- Skip, 126-127
- Take, 126-127

### **Person class, 36**

### **pessimistic concurrency, 368**

**PetCemetery.XML file example, 417-419**

**phishing, 205**

**Predicate<T> delegate, Lambda Expressions, 108-109**

**predicates**

nonequijoins, defining, 219-220

select statements, customizing, 190

**prime number algorithm examples, 186-189**

**PRINT statements, debugging stored procedures, 392**

**ProductsUnderThisUnitPrice function, 363-366**

**profiling code, yield return, 93-94**

**programming**

anonymous types

arrays, initializing, 7-8

composite, 9-12

composite, creating, 9

defining, 7

indexes in for statements, 12-14

returning from functions, 17-18

using statements, 14-16

dynamic programming, Lambda Expressions, 110

**LINQ to XSD**

downloading/installing, 487

object queries, 496-498

Preview console applications, creating, 487

regular expressions added to XML Schema files, 491-494

XML files, defining, 488-490

XML Schema files, defining, 490-491

**“Programming for Fun and Profit—Using the Card.dll” website, 439**

**projecting interfaces, support for, 159-161**

**projecting new types, 200, 203-205**

**projections, 35, 203**

**properties**

Active Directory, assigning, 257

auto-implemented, 34

automatic properties

creating custom objects with, 169

Lambda Expressions, 102-103

ColumnAttribute class, 269-273

DataContextMapping, 372

EntitySet classes as, 300-305

MyPoint, 34

**providers, IQueryable**

creating, 245-246

Smet, Bart De implementation, 245

## Q

**quantifiers, 126**

All, 124-125

Any, 124-125

**querying**

Active Directory, 243-244, 252-254, 260-262

converting

results to lists, 56-57

to Active Directory queries, 252-254

**EDMs**

Entity SQL, 402-405

LINQ to Entities, 405-406

- embedded with XML in VB, 458
  - console application, 460-461
  - Customer class example, 459-460
  - literal XML with embedded expressions and LINQ, 461-462

## joins

- based on composite keys, 237
- cross, 228-236
- equijoins, 214
- group, 224-226
- inner, 213-214
- left outer, 224-228
- multiple from clauses, 211-212
- nonequijoins, 214-223

- LINQ queries, constructing, 122

- LINQ to XSD, 496, 498

- LINQ with anonymous types, 24-25

- nested, LINQ to XML, 428-429

- results, summing, 162-163

- text, viewing, 273-275

- views, LINQ to SQL, 342, 344

- XML documents

- attributes, 420-421

- XDocument class, 416-419

- XElement class, 420

## R

**Range, 127**

**range variables, defining nonequijoins, 220-223**

**ReaderHelper, 73**

**ReadSuppliers method, 191**

## Registry

- overview, 205

- two section comparison, 206-207

**regular expressions, adding to XML Schema files, 491-494**

**relational data, connecting to, 275-277**

## relational database models

- C# programming problems, 384-385

- data access layers, 384

- Entity Framework solution, 385

**Repeat, 128-129**

**requesting stock quotes from Yahoo!, 19-20**

**Requests for Comments, 244**

**resolving conflicts, 375-376**

**resources (web), ADO.NET Entity Framework**

- downloads, 387

- Entity SQL blog, 387

- samples, 388

- Wikipedia, 387

**results of queries, summing, 162-163**

**resultsets, creating composite resultsets, 182-183**

**return type, defining extension methods, 64-65**

**Reverse method, 144-145**

**reversing item order, 144-145**

**RFCs (Requests for Comments), 244**

**right joins, LINQ to Datasets, 315-317**

**Ritchie, Dennis, 98**

**rules for yield return, 88**

## S

**Santana, Carlos, 119**

**ScottGu's Blog website, 203**

**secondary sorts, 141-144**

**security, CAS (code access security), 205**

**select indexes, shuffling/unsorting arrays, 194-195**

**select statements**

- custom business objects, returning, 190-193
- function call effects, 186-189
- initial capping words in arrays, 202-203
- predicates, customizing, 190
- types, projecting, 203-205

**Select<T> method, Lambda Expression, 110-112**

**SelectMany method**

- indexes, 207-208
- types, projecting, 203-205
- Windows Registry sections comparisons, 206-207

**SequenceEqual, 130**

**sequences, appending with Concat, 132-133**

**services oriented architecture, 462**

**set operations**

- composite resultsets, creating, 182-183
- distinct elements, finding, 167
  - defining custom Order object, 169
  - determining median grade from list of numbers, 167-168
  - implementing IEqualityComparer interface, 171-172

implementing object dumper, 172-173

instantiating custom objects, 170-171

sorting and returning distinct list of cities, 173-177

exclusive sets, defining, 177-181

overview, 167

**shaping, 35**

**shared source code, 86**

**shuffling a deck of cards (Blackjack game), 196-199**

**sibling elements, LINQ to XML and XPath comparison, 442**

**sieve of Atkin algorithm, 189**

**sieve of Eratosthenes algorithm example, 186-189**

**single-table mapping, LINQ to SQL inheritance hierarchies, 294-298**

**Skip, partitioning, 126-127**

**SOA (services oriented architectures), 462**

**sorting**

- against DataTables, 282
- distinct list of cities, 173-177
- information
  - in ascending order, 138-139
  - in descending order, 139-141
  - overview, 137
  - reversing order of items, 144-145
  - secondary sorts, 141-144
- XML queries, 431

**source code (shared), 86**

**Space Invaders website, 463**

**sprocs (stored procedures), 223**

**SQL (Structured Query Language)****LINQ to SQL**

- adding data, 349-352
- customizing Object Relational Designer generated classes, 299-300
- databases, creating, 305-307
- databinding, 345-347
- deleting data, 352-354
- equijoins, 317-321
- group joins, 321-331
- inheritance hierarchies, 294-298
- left joins, 331-340
- n-tier applications, 376-381
- querying views, 342-344
- updating data, 354-355

**LINQ to SQL Class designer, 285-286****LINQ to SQL Visual Designer, mapping stored procedures, 360-363****statements, executing in Visual Studio, 481****XML, creating, 469, 473-474**

- object-relational maps, defining, 470-472

**TreeView output of XML document, 475-478****XComment node, 475****XML, updating from**

- examining inserted data, 481
- inserting data, 480-481
- osql.exe scripting output, 482-483
- sample XML file, 478-479

**SQL Server, building views, 340-342****SqlMetal, 285****StackFrame class, 27****statements**

- anonymous types, 14-16
  - binding statements in, 20
  - using, 14-16
- binding statements, anonymous types, 20
- for statements, anonymous type indexes, 12-14

**PRINT, debugging stored procedures, 392****select**

- custom business objects, returning, 190-193
- customizing predicates, 190
- function call effects, 186-189
- initial capping words in arrays, 202-203
- projecting types, 203-205

**SQL, executing in Visual Studio, 481****StockHistory database****ADO.NET 2.0**

- adding foreign keys, 393-394
- adding quotes, 390-392
- complete script, 394-397
- defining, 389-390

**Entity Framework (ADO.NET)**

- adding associations, 402
- creating EDMs, 401-402
- LINQ to XML and LINQ to Entities, 407-411

querying EDMs with Entity SQL,  
402-405

querying EDMs with LINQ to Entities,  
405-406

obtaining stock quotes, updating the  
database, 397, 399-401

**StockHistoryEntities class, 405**

**Storage property (ColumnAttribute  
class), 273**

**stored procedures, 223**

debugging, 392

defining, 358-360

InsertQuote, 390-392

mapping, LINQ to SQL Visual Designer,  
360-363

overview, 355

UpdateCustomer example, 357-358

**strings**

searching, Lambda Expressions, 106-107

XML defined as, 424-425

**SubmitChanges method, 369-371**

**Sum method, 162-163**

**summing query results, 162-163**

**Supplier class, 190**

**System.Ling namespace, 73**

## T

**tables**

associated with conflict, 372-373

DataTables

querying with Where clause, 280-281

selecting data from, 278-280

sorting against, 282

defining, 266-269

mapping classes to, 269-272

**Take, partitioning, 126-127**

**“talking” string extension methods,  
implementing, 78-79**

**testing**

anonymous types equality, 23

equality testing, 129-130

**text (queries), viewing, 273-275**

**text files, creating from XML, 457-458**

**ThenBy method, 138**

**ThenByDescending method, 141**

**ToArray conversion operator, 51-53**

**ToDictionary conversion operator, 57-58**

**ToList conversion operator, 56-57**

**ToLookup conversion operator, 58-59**

**ToLookup method, 150**

**transactions**

parent/child rows, deleting, 366-368

TransactionScope class, 366

**TreeView output of XML document, 475-478**

**triggers, 22**

**types**

anonymous, 203-205

named, 30

nullable, 289-293

projecting, 200, 203-205

## U

**Union method, 182-183**

**Update method, databinding anonymous  
types, 20**

**UpdateCheck** property (**ColumnAttribute** class), 273

**UpdateCustomer** stored procedure, 357-358

**UpdatePriceHistory** methods, 400

**updating**

data, LINQ to SQL, 354-355

SQL from XML

examining inserted data, 481

inserting data, 480-481

osql.exe scripting output, 482-483

sample XML file, 478-479

**user controls**, **HypergraphController**, 47-50

**user-defined functions**, calling, 363-366

**using** statements, anonymous types, 14-16

## V

**var** keyword, anonymous types, 5

**variables (range)**, defining nonequi joins, 220-223

**VB (Visual Basic)**

*VB Today* website, 203

XML with embedded LINQ queries, 458

console application, 460-461

Customer class example, 459-460

literal XML with embedded expressions and LINQ, 461-462

**views**, 340

querying with LINQ to SQL, 342-344

SQL Server, building with, 340-342

**Visual Designer (LINQ to SQL)**, mapping stored procedures, 360-363

**Visual Studio**

SQL statements, executing, 481

stored procedures, defining, 360

## W

**Wagner, Bill**, 80

**WCF (Windows Communication Foundation)**, 377

**WebClient** class, 400

**websites**, 438

101 LINQ Samples by Microsoft, 203

Bill Blogs in C#, 203

Creating Project Templates in .NET (quotes), 487

Entity Framework download, 387

Entity Framework Go Live estimation date, 388

Entity SQL blog, 387

Entity SQL reference, 405

Microsoft XML Team WebLog, 487

Programming for Fun and Profit—Using the Card.dll, 439

ScottGu's Blog, 203

Smet, Bart De IQueryable provider implementation, 245

Space Invaders, 463

*VB Today*, 203

Wikipedia, 387

Yahoo! stock quotes, 426



**West, David, 78**

**Where, extension methods, 73, 76**

**Where clauses, XML documents, 429**

**Where Lambda Expressions, converting to  
Active Directory search filters, 254-256**

**Where<T> method, Lambda Expression,  
112-113**

**Wikipedia, 387**

**Wilde, Oscar, 167**

**Windows Communication Foundation, 377**

**Windows Registry**

overview, 205

two section comparison, 206-207

## X–Z

**XComment node, SQL to XML  
conversions, 475**

**XDocument class, loading XML documents,  
416-419**

**XElement class**

attributes

adding, 422

deleting, 423

node annotations, 433-434

XML documents, loading, 420

**XML**

.csv files, creating from, 454-456

documents

creating from Yahoo! stock quotes,  
426-427

defining as strings, 424-425

element navigation based on context,  
430-431

filtering, 429

functional construction, 450-451

intermediate values, 432-433

loading, 415-416

missing data, 425-426

namespaces, 427-428

nested queries, 428-429

node annotations, 433-434

querying, 416-421

sorting, 431

embedded LINQ queries in VB, 458

console application, 460-461

Customer class example, 459-460

literal XML with embedded  
expressions and LINQ, 461-462

files, creating with

LINQ to XSD, 488-490

XmlWriter class, 465-467

LINQ to XML

StockHistory database

UpdatePriceHistory method, 407-411

XPath, compared, 438-443

XSLT, compared, 443-450

Path Language, 437

Schema files

creating with LINQ to XSD, 490-491

regular expressions, adding, 491-494

SQL, creating from, 469, 473-474

object-relational maps, defining,  
470-472

TreeView output of XML document,  
475-478

XComment node, 475

SQL, updating

- examining inserted data, 481

- inserting data, 480-481

- osql.exe scripting output, 482-483

- sample XML file, 478-479

- text files, creating, 457-458

### **XmlWriter class**

- overview, 464-465

- XML files, creating, 465-467

### **XNamespace object, 427-428**

### **XPath (XML Path Language), LINQ to XML, 437-438**

- child elements, 441

- filtering elements, 442-443

- namespaces, 439-441

- sibling elements, 442

### **XPath (XML Path Language), 437**

### **XQuery, filtering elements, 442**

### **XSLT (Extensible Stylesheet Language Transformations), LINQ to XML, 437, 443**

- debugging documents, 450

- HTML documents, 444-449

### **Yahoo! stock quotes website, 426**

### **yield return, 85-86**

- BinaryTree, 89-93

- demonstration of, 87-88

- profiling code, 93-94

- rules for, 88

- yield break, 95