CHAPTER 1

Programming with Anonymous Types

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

"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

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.)

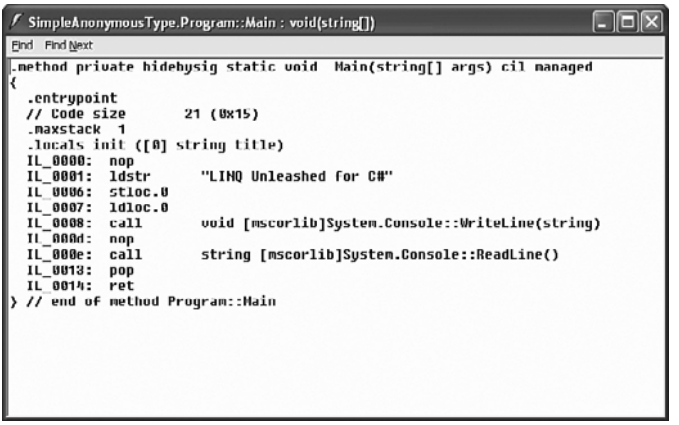

FIGURE 1.1 Looking at the .locals init statement and the `Console::Write(string)` statement in the MSIL, it is clear that title is emitted as a string.

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.

FIGURE 1.2 Anonymous types save a lot of programming time when it comes to composite types, as shown by the elements emitted to MSIL.

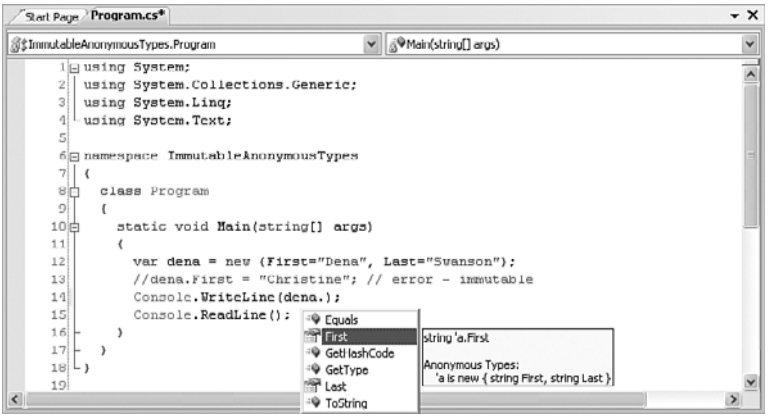

FIGURE 1.3 IntelliSense works quite well with anonymous types.

Adding Methods to Anonymous Types

To really understand language possibilities, it's helpful to bend and twist a language to make it do things it might not have been intended to do directly. One of these things is adding behaviors (aka methods). Although it might be harder to find a practical use for anonymous type–behaviors, Listing 1.4 shows you how to add a behavior to and use that behavior with an anonymous type. (The generic delegate `Func` in bold in the listing is used to initial the anonymous type's method.)

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 AnonysmousTypeWithMethod
{
  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=Concat};` 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 `SqlConnections`—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:  callvirtinstance 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, `HttpWebRequests`, `HttpWebResponses`, 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.)

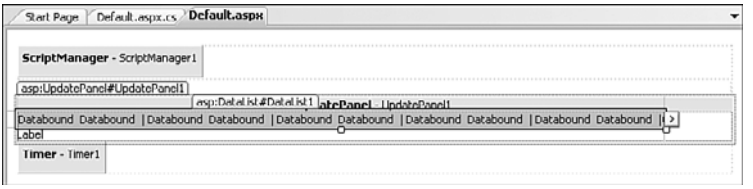

FIGURE 1.4 Just five controls and you have an asynchronous AJAX page.

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 `HttpWebRequest` and `HttpWebResponse` 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.

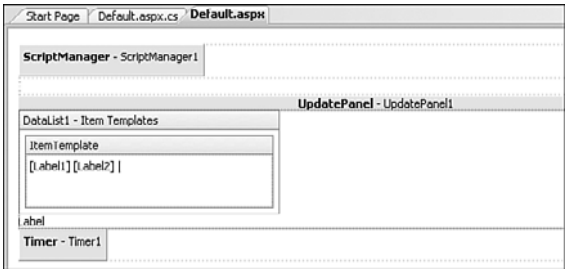

FIGURE 1.6 The template view of the `DataList` is two `Label` controls and the ¦ character.

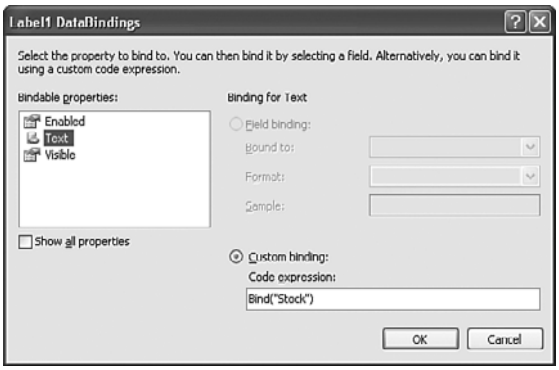

FIGURE 1.7 The binding statements for bound template controls have been very short (as shown) since Visual Studio 2005.

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>
             <asp:Label ID="Label2" runat="server" Text='<%# Bind("Quote") %>'
            ➥</asp:Label>
             ¦
          </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 `XHTMLRequest`, 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 *esoterrorism*—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.