CHAPTER 8

C# 4.0 FEATURES

Goals of this chapter:

- Define new C# 4.0 language features.
- Demonstrate the new language features in the context of LINQ to Objects.

C# is an evolving language. This chapter looks at the new features added into C# 4.0 that combine to improve code readability and extend your ability to leverage LINQ to Object queries over dynamic data sources. The examples in this chapter show how to improve the coding model for developers around reading data from various sources, including text files and how to combine data from a **COM-Interop** source into a LINQ to Objects query.

Evolution of C#

C# is still a relatively new language (circa 2000) and is benefiting from continuing investment by Microsoft's languages team. The C# language is an ECMA and ISO standard. (ECMA is an acronym for European Computer Manufacturers Association, and although it changed its name to Ecma International in 1994, it kept the name Ecma for historical reasons.[1]) The standard ECMA-334 and ISO/IEC 23270:2006 is freely available online at the Ecma International website[2] and describes the language syntax and notation. However, Microsoft's additions to the language over several versions take some time to progress through the standards process, so Microsoft's release cycle leads Ecma's acceptance by at least a version.

Each version of C# has a number of new features and generally a major theme. The major themes have been generics and nullable types in C# 2.0,

LINQ in C# 3.0, and dynamic types in C# 4.0. The major features added in each release are generally considered to be the following:

- **C# 2.0—**Generics (.NET Framework support was added, and C# benefited from this); iterator pattern (the `yield` keyword); anonymous methods (the `delegate` keyword), nullable types, and the null coalescing operator (`??`).
- **C# 3.0—**Anonymous types, extension methods, object initializers, collection initializers, implicitly typed local variables (`var` keyword), lambda expressions (`=>`), and the LINQ query expression pattern.
- **C# 4.0—**Optional Parameters and Named Arguments, Dynamic typing (`dynamic` type), improved COM-Interop, and Contra and Co-Variance.

The new features in C# 3.0 that launched language support for LINQ can be found in Chapter 2, "Introducing LINQ to Objects," and this chapter documents each of the major new features in C# 4.0 from the perspective of how they impact the LINQ story.

Optional Parameters and Named Arguments

A long-requested feature for C# was to allow for method parameters to be optional. Two closely related features in C# 4.0 fulfill this role and enable us to either omit arguments that have a defined default value when calling a method, and to pass arguments by name rather than position when calling a method.

OPTIONAL PARAMETERS OR OPTIONAL ARGUMENTS? Optional parameters and named parameters are sometimes called optional arguments and named arguments. These names are used interchangeably in this book, and in most literature, including the C# 4.0 specification that uses both, sometimes in the same section. I use "argument" when referring to a value passed in from a method call and "parameter" when referring to the method signature.

The main benefit of these features is to improve **COM-Interop** programming (which is covered shortly) and to reduce the number of method overloads created to support a wide range of parameter overloads. It is a

common programming pattern to have a master method signature containing all parameters (with the actual implementation) chained to a number of overloaded methods that have a lesser parameter signature set calling the master method with hard-coded default values. This common coding pattern becomes unnecessary when optional parameters are used in the definition of the aforementioned master method signature, arguably improving code readability and debugging by reducing clutter. (See Listing 8-2 for an example of the old and new way to create multiple overloads.)

There has been fierce debate on these features on various email lists and blogs. Some C# users believe that these features are not necessary and introduce uncertainty in versioning. For example if version 2 of an assembly changes a default parameter value for a particular method, client code that was assuming a specific default might break. This is true, but the existing chained method call pattern suffers from a similar issue—default values are coded into a library or application somewhere, so thinking about when and how to handle these hard-coded defaults would be necessary using either the existing chained method pattern or the new optional parameters and named arguments. Given that optional parameters were left out of the original C# implementation (even when the .NET Runtime had support and VB.NET utilized this feature), we must speculate that although this feature is unnecessary for general programming, coding COM-Interop libraries without this feature is unpleasant and at times infuriating—hence, optional parameters and specifying arguments by name has now made its way into the language.

COM-Interop code has always suffered due to C#'s inability to handle optional parameters as a concept. Many Microsoft Office Component Object Model (**COM**) libraries, like those built to automate Excel or Word for instance, have method signatures that contain 25 optional parameters. Previously you had no choice but to pass dummy arguments until you reached the "one" you wanted and then fill in the remaining arguments until you had fulfilled all 25. Optional parameters and named arguments solve this madness, making coding against COM interfaces much easier and cleaner. The code shown in Listing 8-1 demonstrates the before and after syntax of a simple Excel COM-Interop call to open an Excel spreadsheet. It shows how much cleaner this type of code can be written when using C# 4.0 versus any of its predecessors.

Listing 8-1 Comparing the existing way to call COM-Interop and the new way using optional parameters

```
// Old way - before optional parameters
var excel = new Microsoft.Office.Interop.Excel.Application();
try
{
    Microsoft.Office.Interop.Excel.Workbook workBook =
            excel.Workbooks.Open(fileName, Type.Missing,
                Type.Missing, Type.Missing, Type.Missing,
                Type.Missing, Type.Missing, Type.Missing,
                Type.Missing, Type.Missing, Type.Missing,
                Type.Missing);

    // do work with Excel...

    workBook.Close(false, fileName);
}
finally
{
    excel.Quit();
}

// New Way - Using optional parameters
var excel = new Microsoft.Office.Interop.Excel.Application();
try
{
    Microsoft.Office.Interop.Excel.Workbook workBook =
            excel.Workbooks.Open(fileName);

    // do work with Excel...

    workBook.Close(false, fileName);
}
finally
{
    excel.Quit();
}
```

The addition of object initializer functionality in C# 3.0 took over some of the workload of having numerous constructor overloads by allowing public properties to be set in line with a simpler constructor (avoiding having a

constructor for every Select projection needed). Optional parameters and named arguments offer an alternative way to simplify coding a LINQ Select projection by allowing variations of a type's constructor with a lesser set of parameters. Before diving into how to use these features in LINQ queries, it is necessary to understand the syntax and limitations of these new features.

Optional Parameters

The first new feature allows default parameters to be specified in a method signature. Callers of methods defined with default values can omit those arguments without having to define a specific overload matching that lesser parameter list for convenience.

To define a default value in a method signature, you simply add a constant expression as the default value to use when omitted, similar to member initialization and constant definitions. A simple example method definition that has one mandatory parameter (`p1`, just like normal) and an optional parameter definition (`p2`) takes the following form:

```
public void MyMethod( int p1, int p2 = 5 );
```

The following invocations of method `MyMethod` are legal (will compile) and are functionally equivalent as far as the compiler is concerned:

```
MyMethod( 1, 5 );
MyMethod( 1 ); // the declared default for p2 (5) is used
```

The rules when defining a method signature that uses optional parameters are:

1. Required parameters cannot appear after any optional parameter.
2. The default specified must be a constant expression available at compile time or a value type constructor without parameters, or `default(T)` where `T` is a value type.
3. The constant expression must be implicitly convertible by an identity (or nullable conversion) to the type of the parameter.
4. Parameters with a `ref` or `out` modifier cannot be optional parameters.
5. Parameter arrays (`params`) can occur after optional parameters, but these cannot have a default value assigned. If the value is omitted by the calling invocation, an empty parameter array is used in either case, achieving the same results.

Valid optional parameter definitions take the following form:

```
public void M1(string s, int i = 1) { }
public void M2(Point p = new Point()) { }
public void M3(Point p = default(Point)) { }
public void M4(int i = 1, params string[] values) { }
```

The following method definitions using optional parameters will *not* compile:

```
//"Optional parameters must appear after all required parameters"
public void M1 (int i = 1, string s) {}

//"Default parameter value for 'p' must be a compile-time constant"
//(Can't use a constructor that has parameters)
public void M2(Point p = new Point(0,0)) {}

//"Default parameter value for 'p' must be a compile-time constant"
//(Must be a value type (struct or built-in value types only))
public void M5(StringBuilder p = new StringBuilder()) {}

//"A ref or out parameter cannot have a default value"
public void M6(int i = 1, out string s = "") {}

//"Cannot specify a default value for a parameter array"
public void M7(int i = 1, params string[] values = "test") {}
```

To understand how optional parameters reduce our code, Listing 8-2 shows a traditional overloaded method pattern and the equivalent optional parameter code.

Listing 8-2 Comparing the traditional cascaded method overload pattern to the new optional parameter syntax pattern

```
// Old way - before optional parameters
public class OldWay
{
    // multiple overloads call the one master
    // implementation of a method that handles all inputs
```

```
    public void DoSomething(string formatString)
    {
        // passing 0 as param1 default,
        // and true as param2 default.
        DoSomething(formatString, 0, true);
    }

    public void DoSomething(string formatString, int param1)
    {
        DoSomething(formatString, param1, true);
    }

    public void DoSomething(string formatString, bool param2)
    {
        DoSomething(formatString, 0, param2);
    }

    // the actual implementation. All variations call this
    // method to implement the methods function.
    public void DoSomething(
        string formatString,
        int param1,
        bool param2)
    {
        Console.WriteLine(
            String.Format(formatString, param1, param2));
    }
}

// New Way - Using optional parameters
public class NewWay
{
    // optional parameters have a default specified.
    // optional parameters must come after normal params.
    public void DoSomething(
        string formatString,
        int param1 = 0,
        bool param2 = true)
    {
        Console.WriteLine(
            String.Format(formatString, param1, param2));
    }
}
```

Named Arguments

Traditionally, the position of the arguments passed to a method call identified which parameter that value matched. It is possible in C# 4.0 to specify arguments by name, in addition to position. This is helpful when many parameters are optional and you need to target a specific parameter without having to specify all proceeding optional parameters.

Methods can be called with any combination of positionally specified and named arguments, as long as the following rules are observed:

1. If you are going to use a combination of positional and named arguments, the positional arguments must be passed first. (They cannot come after named arguments.)
2. All non-optional parameters must be specified somewhere in the invocation, either by name or position.
3. If an argument is specified by position, it cannot then be specified by name as well.

To understand the basic syntax, the following example creates a `System.Drawing.Point` by using named arguments. It should be noted that there is no constructor for this type that takes the y-size, x-size by position—this reversal is solely because of named arguments.

```
// reversing the order of arguments.
Point p1 = new Point(y: 100, x: 10);
```

The following method invocations will not compile:

```
//"Named argument 'x' specifies a parameter for which a
// positional argument has already been given"
Point p3 = new Point(10, x: 10);

// "Named argument specifications must appear after all
// fixed arguments have been specified"
Point p4 = new Point(y: 100, 10);

// "The best overload for '.ctor' does not have a
// parameter named 'x'"
Point p5 = new Point(x: 10);
```

To demonstrate how to mix and match optional parameters and named arguments within method or constructor invocation calls, the code shown in Listing 8-3 calls the method definition for `NewWay` in Listing 8-2.

Listing 8-3 Mixing and matching positional and named arguments in a method invocation for methods that have optional and mandatory parameters

```
NewWay newWay = new NewWay();

// skipping an optional parameter
newWay.DoSomething(
     "({0},{1}) New way - param1 skipped.",
     param2: false);

// any order, but if it doesn't have a default
// it must be specified by name somewhere!
newWay.DoSomething(
     param2: false,
     formatString: "({0},{1}) New way - params specified" +
                   " by name, in any order.",
     param1: 5);
```

Using Named Arguments and Optional Parameters in LINQ Queries

Named arguments and optional parameters offer an alternative way to reduce code in LINQ queries, especially regarding flexibility in what parameters can be omitted in an object constructor.

Although anonymous types make it convenient to project the results of a query into an object with a subset of defined properties, these anonymous types are scoped to the local method. To share a type across methods, types, or assemblies, a concrete type is needed, meaning the accumulation of simple types or constructor methods just to hold variations of data shape projections. Object initializers reduce this need by allowing a concrete type to have a constructor without parameters and public properties used to assign values in the Select projection. Object-oriented purists take issue with a parameterless constructor being a requirement; it can lead to invalid objects being created by users who are unaware that certain

properties must be set before an object is correctly initialized for use—an opinion I strongly agree with. (You can't compile using the object initialization syntax unless the type concerned has a parameterless constructor, even if there are other constructors defined that take arguments.)

Optional parameters and named arguments can fill this gap. Data can be projected from queries into concrete types, and the author of that concrete type can ensure that the constructor maintains integrity by defining the default values to use when an argument is omitted. Many online discussions have taken place discussing if this is a good pattern; one camp thinks it doesn't hurt code readability or maintainability to use optional parameters in a constructor definition, and the other says refactoring makes it an easy developer task to define the various constructors required in a given type, and hence of no value. I see both sides of that argument and will leave it up to you to decide where it should be employed.

To demonstrate how to use named arguments and optional parameters from a LINQ query, the example shown in Listing 8-4 creates a subset of contact records (in this case, contacts from California) but omits the email and phone details. The Console output from this example is shown in Output 8-1.

Listing 8-4 Example LINQ query showing how to use named arguments and optional parameters to assist in projecting a lighter version of a larger type—see Output 8-1

```
var q = from c in Contact.SampleData()
        where c.State == "CA"
        select new Contact(
            c.FirstName, c.LastName,
            state: c.State,
            dateOfBirth: c.DateOfBirth
            );

foreach (var c in q)
    Console.WriteLine("{0}, {1} ({2}) - {3}",
        c.LastName, c.FirstName,
        c.DateOfBirth.ToShortDateString(), c.State);

public class Contact
{
    // constructor defined with optional parameters
    public Contact(
        string firstName,
        string lastName,
```

```
        DateTime dateOfBirth,
        string email = "unknown",  // optional
        string phone = "",         // optional
        string state = "Other")    // optional
    {
        FirstName = firstName;
        LastName = lastName;
        DateOfBirth = dateOfBirth;
        Email = email;
        Phone = phone;
        State = state;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string State { get; set; }

    public static List<Contact> SampleData() ...
    // sample data the same as used in Table 2-1.
}
```

Output 8-1

```
Gottshall, Barney (10/19/1945) - CA
Deane, Jeffery (12/16/1950) - CA
```

Dynamic Typing

The wow feature of C# 4.0 is the addition of dynamic typing. Dynamic languages such as Python and Ruby have major followings and have formed a reputation of being super-productive languages for building certain types of applications.

The main difference between these languages and C# or VB.NET is the type system and specifically when (and how) member names and method names are resolved. C# and VB.NET require (or required, as you will see) that static types be available during compile time and will fail if a