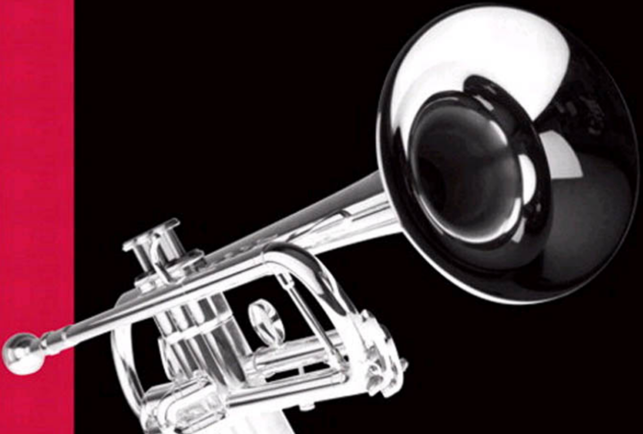
Effective SOFTWARE DEVELOPMENT SERIES
Scott Meyers, Consulting Editor

MORE *Effective* C#

50 Specific Ways to Improve Your C#



Bill Wagner

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Wagner, Bill.
More effective C# : 50 specific ways to improve your C# / Bill Wagner.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-48589-2 (pbk. : alk. paper)
1. C# (Computer program language) 2. Database management. 3.
Microsoft .NET. I. Title.

QA76.73.C154W343 2008
005.13'3—dc22

2008030878

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN-13: 978-0-321-48589-2
ISBN-10: 0-321-48589-0
Text printed in the United States on recycled paper at Donnelley in Crawfordsville, IN.
First printing, October 2008

Introduction

When Anders Hejlsberg first showed Language-Integrated Query (LINQ) to the world at the 2005 Professional Developers Conference (PDC), the C# programming world changed. LINQ justified several new features in the C# language: extension methods, local variable type inference, lambda expressions, anonymous types, object initializers, and collection initializers. C# 2.0 set the stage for LINQ by adding generics, iterators, static classes, nullable types, property accessor accessibility, and anonymous delegates. But all these features are useful outside LINQ: They are handy for many programming tasks that have nothing to do with querying data sources.

This book provides practical advice about the features added to the C# programming language in the 2.0 and 3.0 releases, along with advanced features that were not covered in my earlier *Effective C#: 50 Specific Ways to Improve Your C#* (Addison-Wesley, 2004). The items in *More Effective C#* reflect the advice I give developers who are adopting C# 3.0 in their professional work. There's a heavy emphasis on generics, an enabling technology for everything in C# 2.0 and 3.0. I discuss the new features in C# 3.0; rather than organize the topics by language feature, I present these tips from the perspective of recommendations about the programming problems that developers can best solve by using these new features.

Consistent with the other books in the Effective Software Development Series, this book contains self-contained items detailing specific advice about how to use C#. The items are organized to guide you from using C# 1.x to using C# 3.0 in the best way.

Generics are an enabling technology for all new idioms that are part of C# 3.0. Although only the first chapter specifically addresses generics, you'll find that they are an integral part of almost every item. After reading this book, you'll be much more comfortable with generics and metaprogramming.

Of course, much of the book discusses how to use C# 3.0 and the LINQ query syntax in your code. The features added in C# 3.0 are very useful in their own right, whether or not you are querying data sources. These changes in the language are so extensive, and LINQ is such a large part of the justification for those changes, that each warrants its own chapter. LINQ and C# 3.0 will have a profound impact on how you write code in C#. This book will make that transition easier.

Who Should Read This Book?

This book was written for professional software developers who use C#. It assumes that you have some familiarity with C# 2.0 and C# 3.0. Scott Meyers counseled me that an *Effective* book should be a developer's second book on a subject. This book does not include tutorial information on the new language features added as the language has evolved. Instead, I explain how you can integrate these features into your ongoing development activities. You'll learn when to leverage the new language features in your development activities, and when to avoid certain practices that will lead to brittle code.

In addition to some familiarity with the newer features of the C# language, you should have an understanding of the major components that make up the .NET Framework: the .NET CLR (Common Language Runtime), the .NET BCL (Base Class Library), and the JIT (Just In Time) compiler. This book doesn't cover .NET 3.0 components, such as WCF (Windows Communication Foundation), WPF (Windows Presentation Foundation), and WF (Windows Workflow Foundation). However, all the idioms presented apply to those components as well as any other .NET Framework components you happen to prefer.

About the Content

Generics are the enabling technology for everything else added to the C# language since C# 1.1. Chapter 1 covers generics as a replacement for `System.Object` and casts and then moves on to discuss advanced techniques such as constraints, generic specialization, method constraints, and backward compatibility. You'll learn several techniques in which generics will make it easier to express your design intent.

Multicore processors are already ubiquitous, with more cores being added seemingly every day. This means that every C# developer needs to have a solid understanding of the support provided by the C# language for multithreaded programming. Although one chapter can't cover everything you need to be an expert, Chapter 2 discusses the techniques you'll need every day when you write multithreaded applications.

Chapter 3 explains how to express modern design idioms in C#. You'll learn the best way to express your intent using the rich palette of C# language features. You'll see how to leverage lazy evaluation, create composable interfaces, and avoid confusion among the various language elements in your public interfaces.

Chapter 4 discusses how to use the enhancements in C# 3.0 to solve the programming challenges you face every day. You'll see when to use extension methods to separate contracts from implementation, how to use C# closures effectively, and how to program with anonymous types.

Chapter 5 explains LINQ and query syntax. You'll learn how the compiler maps query keywords to method calls, how to distinguish between delegates and expression trees (and convert between them when needed), and how to escape queries when you're looking for scalar results.

Chapter 6 covers those items that defy classification. You'll learn how to define partial classes, work with nullable types, and avoid covariance and contravariance problems with array parameters.

Regarding the Sample Code

The samples in this book are not complete programs. They are the smallest snippets of code possible that illustrate the point. In several samples the method names substitute for a concept, such as `AllocateExpensiveResource()`. Rather than read pages of code, you can grasp the concept and quickly apply it to your professional development. Where methods are elided, the name implies what's important about the missing method.

In all cases, you can assume that the following namespaces are specified:

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

Where types are used from other namespaces, I've explicitly included the namespace in the type.

In the first three chapters, I often show C# 2.0 and C# 3.0 syntax where newer syntax is preferred but not required. In Chapters 4 and 5 I assume that you would use the 3.0 syntax.

Making Suggestions and Providing Feedback

I've made every effort to remove all errors from this book, but if you believe you have found an error, please contact me at bill.wagner@srtsolutions.com. Errata will be posted to http://srtsolutions.com/blogs/MoreEffectiveCSharp.

Acknowledgments

A colleague recently asked me to describe what it feels like to finish a book. I replied that it gives you that same feeling of satisfaction and relief that shipping a software product gives you. It's very satisfying, and yet it's an incredible amount of work. Like shipping a software product, completing a book requires collaboration among many people, and all those people deserve thanks.

I was honored to be part of the Effective Software Development Series when I wrote *Effective C#* in 2004. To follow that up with *More Effective C#* and cover the numerous and far-reaching changes in the language since then is an even greater honor. The genesis of this book was a dinner I shared with Curt Johnson and Joan Murray at PDC 2005, when I expressed my excitement about the direction Hejlsberg and the rest of the C# team were presenting there. I was already taking notes about the changes and learning how they would affect the daily lives of C# developers.

Of course, it was some time before I felt comfortable in offering advice on all these new features. I needed to spend time using them and discussing different idioms with coworkers, customers, and other developers in the community. Once I felt comfortable with the new features, I began working on the new manuscript.

I was lucky enough to have an excellent team of technical reviewers. These people suggested new topics, modified the recommendations, and found scores of technical errors in earlier drafts. Bill Craun, Wes Dyer, Nick Paldino, Tomas Restrepo, and Peter Ritchie provided detailed technical feedback that made this book as useful as it is now. Pavin Podila reviewed those areas that mention WPF to ensure correctness.

Throughout the writing process, I discussed many ideas with members of the community and the C# team. The regulars at the Ann Arbor .NET Developers Group, the Great Lakes Area .NET User Group, the Greater Lansing User Group, the West Michigan .NET User Group, and the Toledo .NET User Group acted as prototype audiences for much of the advice presented here. In addition, CodeMash attendees helped me decide what to leave in and what to leave out. In particular, I want to single out Dustin Campbell, Jay Wren, and Mike Woelmer for letting me discuss ideas with them. In addition, Mads Torgersen, Charlie Calvert, and Eric Lippert joined me in several conversations that helped clarify the advice detailed here. In particular, Charlie Calvert has the great skill of mixing an engineer's understanding with a writer's gift of clarity. Without all those discussions, this manuscript would be far less clear, and it would be missing a number of key concepts.

Having been through Scott Meyers's thorough review process twice now, I'd recommend any book in his series sight unseen. Although he's not a C# expert, he's highly gifted and clearly cares about the books in his series. Responding to his comments takes quite a bit of time, but it results in a much better book.

Throughout the whole process, Joan Murray has been an incredible asset. As editor, she's always on top of everything. She prodded me when I needed prodding, she provided a great team of reviewers, and she helped shepherd the book from inception through outlines, manuscript drafts, and finally into the version you hold now. Along with Curt Johnson, she makes working with Addison-Wesley a joy.

The last step is working with a copy editor. Betsy Hardinger was somehow able to translate an engineer's jargon into English without sacrificing technical correctness. The book you're holding is much easier to read after her edits.

Of course, writing a book takes a large investment of time. During that time, Dianne Marsh, the other owner of SRT Solutions, kept the company moving forward. The greatest sacrifice was from my family, who saw much less of me than they'd like while I was writing this book. The biggest thanks go to Marlene, Lara, Sarah, and Scott, who put up with me as I did this again.

Simply return from the background thread procedure, and handle the error in the event handler for the foreground results.

Earlier I said that I often use `BackgroundWorker` in classes that aren't the `Form` class, and even in non-Windows Forms applications, such as services or Web services. This works fine, but it does have some caveats. When `BackgroundWorker` determines that it is running in a Windows Forms application and the form is visible, the `ProgressChanged` and `RunWorkerCompleted` events are marshaled to the graphical user interface (GUI) thread via a marshaling control and `Control.BeginInvoke` (see Item 16 later in this chapter). In other scenarios, those delegates are simply called on a free thread pool thread. As you will see in Item 16, that behavior may affect the order in which events are received.

Finally, because `BackgroundWorker` is built on `QueueUserWorkItem`, you can reuse `BackgroundWorker` for multiple background requests. You need to check the `IsBusy` property of `BackgroundWorker` to see whether `BackgroundWorker` is currently running a task. When you need to have multiple background tasks running, you can create multiple `BackgroundWorker` objects. Each will share the same thread pool, so you have multiple tasks running just as you would with `QueueUserWorkItem`. You need to make sure that your event handlers use the correct sender property. This practice ensures that the background threads and foreground threads are communicating correctly.

`BackgroundWorker` supports many of the common patterns that you will use when you create background tasks. By using it you can reuse that implementation in your code, adding any of those patterns as needed. You don't have to design your own communication protocols between foreground and background threads.

Item 13: Use lock() as Your First Choice for Synchronization

Threads need to communicate with each other. Somehow, you need to provide a safe way for various threads in your application to send and receive data. However, sharing data between threads introduces the potential for data integrity errors in the form of synchronization issues. Therefore, you need to be certain that the current state of every shared data item is consistent. You achieve this safety by using **synchronization primitives** to protect access to the shared data. Synchronization primitives ensure that the current thread is not interrupted until a critical set of operations is completed.

There are many primitives available in the .NET BCL that you can use to safely ensure that access to shared data is synchronized. Only one pair of them—`Monitor.Enter()` and `Monitor.Exit()`—was given special status in the C# language. `Monitor.Enter()` and `Monitor.Exit()` implement a **critical section** block. Critical sections are such a common synchronization technique that the language designers added support for them using the `lock()` statement. You should follow that example and make `lock()` your primary tool for synchronization.

The reason is simple: The compiler generates consistent code, but you may make mistakes some of the time. The C# language introduces the *lock* keyword to control synchronization for multithreaded programs. The lock statement generates exactly the same code as if you used `Monitor.Enter()` and `Monitor.Exit()` correctly. Furthermore, it's easier and it automatically generates all the exception-safe code you need.

However, under two conditions `Monitor` gives you necessary control that you can't get when you use `lock()`. First, be aware that `lock` is lexically scoped. This means that you can't enter a `Monitor` in one lexical scope and exit it in another when using the `lock` statement. Thus, you can't enter a `Monitor` in a method and exit it inside a lambda expression defined in that method (see Item 41, Chapter 5). The second reason is that `Monitor.Enter` supports a time-out, which I cover later in this item.

You can lock any reference type by using the `lock` statement:

```
public int TotalValue
{
    get
    {
        lock(syncHandle)
        {
            return total;
        }
    }
}

public void IncrementTotal()
{
    lock (syncHandle)
    {
        total++;
    }
}
```

The `lock` statement gets the exclusive monitor for an object and ensures that no other thread can access the object until the lock is released. The preceding sample code, using `lock()`, generates the same IL as the following version, using `Monitor.Enter()` and `Monitor.Exit()`:

```
public void IncrementTotal()
{
    object tmpObject = syncHandle;
    System.Threading.Monitor.Enter(tmpObject);
    try
    {
        total++;
    }
    finally
    {
        System.Threading.Monitor.Exit(tmpObject);
    }
}
```

The `lock` statement provides many checks that help you avoid common mistakes. It checks that the type being locked is a reference type, as opposed to a value type. The `Monitor.Enter` method does not include such safeguards. This routine, using `lock()`, doesn't compile:

```
public void IncrementTotal()
{
    lock (total) // compiler error: can't lock value type
    {
        total++;
    }
}
```

But this does:

```
public void IncrementTotal()
{
    // really doesn't lock total.
    // locks a box containing total.
    Monitor.Enter(total);
    try
    {
        total++;
    }
```

```
    finally
    {
        // Might throw exception
        // unlocks a different box containing total
        Monitor.Exit(total);
    }
}
```

`Monitor.Enter()` compiles because its official signature takes a `System.Object`. You can coerce `total` into an object by boxing it. `Monitor.Enter()` actually locks the box containing `total`. That's where the first bug lurks. Imagine that thread 1 enters `IncrementTotal()` and acquires a lock. Then, while incrementing `total`, the second thread calls `IncrementTotal()`. Thread 2 now enters `IncrementTotal()` and acquires the lock. It succeeds in acquiring a different lock, because `total` gets put into a different box. Thread 1 has a lock on one box containing the value of `total`. Thread 2 has a lock on another box containing the value of `total`. You've got extra code in place, and no synchronization.

Then you get bitten by the second bug: When either thread tries to release the lock on `total`, the `Monitor.Exit()` method throws a `SynchronizationLockException`. That's because `total` goes into yet another box to coerce it into the method signature for `Monitor.Exit`, which also expects a `System.Object` type. When you release the lock on this box, you unlock a resource that is different from the resource that was used for the lock. `Monitor.Exit()` fails and throws an exception.

Of course, some bright soul might try this:

```
public void IncrementTotal()
{
    // doesn't work either:
    object lockHandle = total;
    Monitor.Enter(lockHandle);
    try
    {
        total++;
    }
    finally
    {
        Monitor.Exit(lockHandle);
    }
}
```

This version doesn't throw any exceptions, but neither does it provide any synchronization protection. Each call to `IncrementTotal()` creates a new box and acquires a lock on that object. Every thread succeeds in immediately acquiring the lock, but it's not a lock on a shared resource. Every thread wins, and `total` is not consistent.

There are subtler errors that `lock` also prevents. `Enter()` and `Exit()` are two separate calls, so you can easily make the mistake of acquiring and releasing different objects. This action may cause a `SynchronizationLockException`. But if you happen to have a type that locks more than one synchronization object, it's possible to acquire two different locks in a thread and release the wrong one at the end of a critical section.

The `lock` statement automatically generates exception-safe code, something many of us humans forget to do. Also, it generates more-efficient code than `Monitor.Enter()` and `Monitor.Exit()`, because it needs to evaluate the target object only once. So, by default, you should use the `lock` statement to handle the synchronization needs in your C# programs.

However, there is one limitation to the fact that `lock` generates the same MSIL as `Monitor.Enter( )`. The problem is that `Monitor.Enter()` waits forever to acquire the lock. You have introduced a possible deadlock condition. In large enterprise systems, you may need to be more defensive in how you attempt to access critical resources. `Monitor.TryEnter( )` lets you specify a time-out for an operation and attempt a workaround when you can't access a critical resource.

```
public void IncrementTotal()
{
    if (!Monitor.TryEnter(syncHandle, 1000)) // wait 1 second
        throw new PreciousResourceException
            ("Could not enter critical section");
    try
    {
        total++;
    }
    finally
    {
        Monitor.Exit(syncHandle);
    }
}
```

You can wrap this technique in a handy little generic class:

```
public sealed class LockHolder<T> : IDisposable
    where T : class
{
    private T handle;
    private bool holdsLock;

    public LockHolder(T handle, int milliSecondTimeout)
    {
        this.handle = handle;
        holdsLock = System.Threading.Monitor.TryEnter(
            handle, milliSecondTimeout);
    }

    public bool LockSuccessful
    {
        get { return holdsLock; }
    }

    #region IDisposable Members
    public void Dispose()
    {
        if (holdsLock)
            System.Threading.Monitor.Exit(handle);
        // Don't unlock twice
        holdsLock = false;
    }
    #endregion
}
```

You would use this class in the following manner:

```
object lockHandle = new object();

using (LockHolder<object> lockObj = new LockHolder<object>
    (lockHandle, 1000))
{
    if (lockObj.LockSuccessful)
    {
        // work elided
    }
}
// Dispose called here.
```

The C# team added implicit language support for `Monitor.Enter()` and `Monitor.Exit()` pairs in the form of the `lock` statement because it is the most common synchronization technique that you will use. The extra checks that the compiler can make on your behalf make it easier to create synchronization code in your application. Therefore, `lock()` is the best choice for most synchronization between threads in your C# applications.

However, `lock` is not the only choice for synchronization. In fact, when you are synchronizing access to numeric types or are replacing a reference, the `System.Threading.Interlocked` class supports synchronizing single operations on objects. `System.Threading.Interlocked` has a number of methods that you can use to access shared data so that a given operation completes before any other thread can access that location. It also gives you a healthy respect for the kinds of synchronization issues that arise when you work with shared data.

Consider this method:

```
public void IncrementTotal()
{
    total++;
}
```

As written, interleaved access could lead to an inconsistent representation of the data. An increment operation is not a single machine instruction. The value of `total` must be fetched from main memory and stored in a register. Then the value of the register must be incremented, and the new value from the register must be stored back into the proper location in main memory. If another thread reads the value after the first thread, the second thread grabs the value from main memory but before storing the new value, thereby causing data inconsistency.

Suppose two threads interleave calls to `IncrementTotal`. Thread A reads the value of 5 from `total`. At that moment, the active thread switches to thread B. Thread B reads the value of 5 from `total`, increments it, and stores 6 in the value of `total`. At this moment, the active thread switches back to thread A. Thread A now increments the register value to 6 and stores that value in `total`. As a result, `IncrementTotal()` has been called twice—once by thread A, and once by thread B—but because of untimely interleaved access, the end effect is that only one update has occurred. These errors are hard to find, because they result from interleaved access at exactly the wrong moment.

You could use `lock()` to synchronize this operation, but there is a better way. The `Interlocked` class has a simple method that fixes the problem: `InterlockedIncrement`. If you rewrite `IncrementTotal` as follows, the increment operation cannot be interrupted and both increment operations will always be recorded:

```
public void IncrementTotal()
{
    System.Threading.Interlocked.Increment(ref total);
}
```

The `Interlocked` class contains other methods to work with built-in data types. `Interlocked.Decrement()` decrements a value. `Interlocked.Exchange()` switches a value with a new value and returns the current value. You'd use `Interlocked.Exchange()` to set new state and return the preceding state. For example, suppose you want to store the user ID of the last user to access a resource. You can call `Interlocked.Exchange()` to store the current user ID while at the same time retrieving the previous user ID.

Finally, there is the `CompareExchange()` method, which reads the value of a piece of shared data and, if the value matches a sought value, updates it. Otherwise, nothing happens. In either case, `CompareExchange` returns the preceding value stored at that location. In the next section, Item 14 shows how to use `CompareExchange` to create a private lock object inside a class.

The `Interlocked` class and `lock()` are not the only synchronization primitives available. The `Monitor` class also includes the `Pulse` and `Wait` methods, which you can use to implement a consumer/producer design. You can also use the `ReaderWriterLockSlim` class for those designs in which many threads are accessing a value that few threads are modifying. `ReaderWriterLockSlim` contains several improvements over the earlier version of `ReaderWriterLock`. You should use `ReaderWriterLockSlim` for all new development.

For most common synchronization problems, examine the `Interlocked` class to see whether you can use it to provide the capabilities you need. With many single operations, you can. Otherwise, your first choice is the `lock()` statement. Look beyond those only when you need special-purpose locking capability.

That introduces a breaking change in the application. This code snippet sets the value of `Marker` to 5:

```
MyType t = new MyType();
t.NextMarker(); // t.Marker == 5
```

You can't avoid this problem entirely, but you can minimize its effects. This sample was contrived to exhibit bad behavior. In production code, the behavior of the extension method should be semantically the same as that of the class method having the same signature. If you can create a better, more efficient algorithm in a class, you should do that. However, you must ensure that the behavior is the same. If you do that, then this behavior won't affect program correctness.

When you find that your design calls for making an interface definition that many classes will be forced to implement, consider creating the smallest possible set of members defined in the interface. Then provide an implementation of convenience methods in the form of extension methods. In that way, class designers who implement your interface will have the least amount of work to do, and developers using your interface can get the greatest possible benefit.

Item 29: Enhance Constructed Types with Extension Methods

You'll probably use a number of constructed generic types in your application. You'll create specific collection types: `List<int>`, `Dictionary<EmployeeID, Employee>`, and many other collections. The purpose of creating these collections is that your application has a specific need for a collection of a certain type and you want to have specific behavior defined for those specific constructed types. To implement that functionality in a low-impact way, you can create a set of extension methods on specific constructed types.

You can see this pattern in the `System.Linq.Enumerable` class. Item 28 (in this chapter) discusses the extension pattern used by `Enumerable<T>` to implement many common methods on sequences as extension methods on `IEnumerable<T>`. In addition, `Enumerable` contains a number of methods that are implemented specifically for particular constructed types that implement `IEnumerable<T>`. For example, several numeric methods are implemented on numeric sequences (`IEnumerable<int>`, `IEnumerable<double>`, `IEnumerable<long>`, and `IEnumerable<float>`). Here are a few of the extension methods implemented specifically for `IEnumerable<int>`:

```
public class Enumerable
{
    public static int Average(this IEnumerable<int>
        sequence);
    public static int Max(this IEnumerable<int> sequence);
    public static int Min(this IEnumerable<int> sequence);
    public static int Sum(this IEnumerable<int> sequence);

    // other methods elided
}
```

Once you recognize the pattern, you can see many ways you could implement the same kind of extensions for the constructed types in your own domain. If you were writing an e-commerce application and you wanted to send e-mail coupons to a set of customers, the method signature might look something like this:

```
public static void SendEmailCoupons(this
    IEnumerable<Customer>
    customers, Coupon specialOffer);
```

Similarly, you could find all customers with no orders in the past month:

```
public static IEnumerable<Customer> LostProspects(
    this IEnumerable<Customer> targetList);
```

If you didn't have extension methods, you could achieve a similar effect by deriving a new type from the constructed generic type you used. For example, the `Customer` methods just shown could be implemented like this:

```
public class CustomerList : List<Customer>
{
    public void SendEmailCoupons(Coupon specialOffer);
    public static IEnumerable<Customer> LostProspects();

}
```

It works, but it is actually much more limiting than extension methods on `IEnumerable<Customer>` to the users of this list of customers. The difference in the method signatures provides part of the reason. The extension methods use `IEnumerable<Customer>` as the parameter, but the methods added to the derived class are based on `List<Customer>`. They mandate a particular storage model. For that reason, they can't be composed as a set of iterator methods (see Item 17, Chapter 3). You've placed unnecessary design constraints on the users of these methods. That's a misuse of inheritance.

Another reason to prefer the extension methods as a way to implement this functionality has to do with the way queries are composed. The `LostProspects()` method probably would be implemented something like this:

```
public static IEnumerable<Customer> LostProspects(
    IEnumerable<Customer> targetList)
{
    IEnumerable<Customer> answer =
        from c in targetList
        where DateTime.Now - c.LastOrderDate >
            TimeSpan.FromDays(30)
        select c;
    return answer;
}
```

Item 34 (later in this chapter) discusses why lambda expressions are preferred over methods in queries. Implementing these features as extension methods means that they provide a reusable query expressed as a lambda expression. You can reuse the entire query rather than try to reuse the predicate of the `where` clause.

If you examine the object model for any application or library you are writing, you'll likely find many constructed types used for the storage model. You should look at these constructed types and decide what methods logically would be added to each of them. It's best to create the implementation for those methods as extension methods by using either the constructed type or a constructed interface implemented by the type. You'll turn a simple generic instantiation into a class having all the behavior you need. Furthermore, you'll create that implementation in a manner that decouples the storage model from the implementation to the greatest extent possible.

Item 30: Prefer Implicitly Typed Local Variables

Implicitly typed local variables were added to the C# language to support anonymous types. A second reason for using implicitly typed locals is that

5 Working with LINQ

The driving force behind the language enhancements to C# 3.0 was LINQ. The new features and the implementation of those features were driven by the need to support deferred queries, translate queries into SQL to support LINQ to SQL, and add a unifying syntax to the various data stores. Chapter 4 shows you how the new language features can be used for many development idioms in addition to data query. This chapter concentrates on using those new features for querying data, regardless of source.

A goal of LINQ is that language elements perform the same work no matter what the data source is. However, even though the syntax works with all kinds of data sources, the query provider that connects your query to the actual data source is free to implement that behavior in a variety of ways. If you understand the various behaviors, it will make it easier to work with various data sources transparently. If you need to, you can even create your own data provider.

Item 36: Understand How Query Expressions Map to Method Calls

LINQ is built on two concepts: a query language, and a translation from that query language into a set of methods. The C# compiler converts query expressions written in that query language into method calls.

Every query expression has a mapping to a method call or calls. You should understand this mapping from two perspectives. From the perspective of a class user, you need to understand that your query expressions are nothing more than method calls. A `where` clause translates to a call to a method named `Where()`, with the proper set of parameters. As a class designer, you should evaluate the implementations of those methods provided by the base framework and determine whether you can create better implementations for your types. If not, you should simply defer to the base library versions. However, when you can create a better version, you must make sure that you fully understand the translation from query expressions into method calls. It's your responsibility to ensure that your method signatures correctly handle every translation case. For some of the query expressions, the correct path is rather obvious. However, it's a little more difficult to comprehend a couple of the more complicated expressions.

The full **query expression pattern** contains eleven methods. The following is the definition from *The C# Programming Language,* Third Edition, by Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde (Microsoft Corporation, 2009), §7.15.3 (reprinted with permission from Microsoft Corporation):

```
delegate R Func<T1,R>(T1 arg1);
delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);
class C
{
    public C<T> Cast<T>();
}

class C<T> : C
{
    public C<T> Where(Func<T,bool> predicate);
    public C<U> Select<U>(Func<T,U> selector);
    public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
        Func<T,U,V> resultSelector);
    public C<V> Join<U,K,V>(C<U> inner,
        Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector,
        Func<T,U,V> resultSelector);
    public C<V> GroupJoin<U,K,V>(C<U> inner,
        Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector,
        Func<T,C<U>,V> resultSelector);
    public O<T> OrderBy<K>(Func<T,K> keySelector);
    public O<T> OrderByDescending<K>(Func<T,K> keySelector);
    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);
    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}

class O<T> : C<T>
{
```

```
    public O<T> ThenBy<K>(Func<T,K> keySelector);
    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T> : C<T>
{
    public K Key { get; }
}
```

The .NET base library provides two general-purpose reference implementations of this pattern. `System.Linq.Enumerable` provides extension methods on `IEnumerable<T>` that implement the query expression pattern. `System.Linq.Queryable` provides a similar set of extension methods on `IQueryable<T>` that supports a query provider's ability to translate queries into another format for execution. (For example, the LINQ to SQL implementation converts query expressions into SQL queries that are executed by the SQL database engine.) As a class user, you are probably using one of those two reference implementations for most of your queries.

Second, as a class author, you can create a data source that implements `IEnumerable<T>` or `IQueryable<T>` (or a closed generic type from `IEnumerable<T>` or `IQueryable<T>`), and in that case your type already implements the query expression pattern. Your type has that implementation because you're using the extension methods defined in the base library.

Before we go further, you should understand that the C# language does not enforce any execution semantics on the query expression pattern. You can create a method that matches the signature of one of the query methods and does anything internally. The compiler cannot verify that your `Where` method satisfies the expectations of the query expression pattern. All it can do is ensure that the syntactic contract is satisfied. This behavior isn't any different from that of any interface method. For example, you can create an interface method that does anything, whether or not it meets users' expectations.

Of course, this doesn't mean that you should ever consider such a plan. If you implement any of the query expression pattern methods, you should ensure that its behavior is consistent with the reference implementations, both syntactically and semantically. Except for performance differences, callers should not be able to determine whether your method is being used or the reference implementations are being used.

Translating from query expressions to method invocations is a complicated iterative process. The compiler repeatedly translates expressions to methods until all expressions have been translated. Furthermore, the compiler has a specified order in which it performs these translations, although I'm not explaining them in that order. The compiler order is easy for the compiler and is documented in the C# specification. I chose an order that makes it easier to explain to humans. For our purposes, I discuss some of the translations in smaller, simpler examples.

In the following query, let's examine the `where`, `select`, and `range` variables:

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var smallNumbers = from n in numbers
                   where n < 5
                   select n;
```

The expression `from n in numbers` binds the range variable `n` to each value in `numbers`. The `where` clause defines a filter that will be translated into a `where` method. The expression `where n < 5` translates to the following:

```
numbers.Where((n) => n < 5);
```

`Where` is nothing more than a filter. The output of `Where` is a proper subset of the input sequence containing only those elements that satisfy the predicate. The input and output sequences must contain the same type, and a correct `Where` method must not modify the items in the input sequence. (User-defined predicates may modify items, but that's not the responsibility of the query expression pattern.)

That `where` method can be implemented either as an instance method accessible to `numbers` or as an extension method matching the type of `numbers`. In the example, `numbers` is an array of `int`. Therefore, `n` in the method call must be an integer.

`Where` is the simplest of the translations from query expression to method call. Before we go on, let's dig a little deeper into how this works and what that means for the translations. The compiler completes its translation from query expression to method call before any overload resolution or type binding. The compiler does not know whether there are any candidate methods when the compiler translates the query expression to a method call. It doesn't examine the type, and it doesn't look for any candidate extension methods. It simply translates the query expression into the method call. After all queries have been translated into method call syntax, the compiler performs the work of searching for candidate methods and then determining the best match.

Next, you can extend that simple example to include the `select` expression in the query. `Select` clauses are translated into `Select` methods. However, in certain special cases the `Select` method can be optimized away. The sample query is a **degenerate select,** selecting the `range` variable. Degenerate select queries can be optimized away, because the output sequence is not equal to the input sequence. The sample query has a `where` clause, which breaks that identity relationship between the input sequence and the output sequence. Therefore, the final method call version of the query is this:

```
var smallNumbers = numbers.Where(n => n < 5);
```

The `select` clause is removed because it is redundant. That's safe because the `select` operates on an immediate result from another query expression (in this example, `where`).

When the `select` does not operate on the immediate result of another expression, it cannot be optimized away. Consider this query:

```
var allNumbers = from n in numbers select n;
```

It will be translated into this method call:

```
var allNumbers = numbers.Select(n => n);
```

While we're on this subject, note that `select` is often used to transform or project one input element into a different element or into a different type. The following query modifies the value of the result:

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var smallNumbers = from n in numbers
                   where n < 5
                   select n * n;
```

Or you could transform the input sequence into a different type as follows:

```
int [] numbers = {0,1,2,3,4,5,6,7,8,9};
var squares = from n in numbers
              select new { Number = n, Square = n * n};
```

The `select` clause maps to a `Select` method that matches the signature in the query expression pattern:

```
var squares = numbers.Select(n =>
    new { Number = n, Square = n * n});
```

`Select` transforms the input type into the output type. A proper `select` method must produce exactly one output element for each input element. Also, a proper implementation of `Select` must not modify the items in the input sequence.

That's the end of the simpler query expressions. Now we discuss some of the less obvious transformations.

Ordering relations map to the `OrderBy` and `ThenBy` methods, or `OrderByDescending` and `ThenByDescending`. Consider this query:

```
var people = from e in employees
             where e.Age > 30
             orderby e.LastName, e.FirstName, e.Age
             select e;
```

It translates into this:

```
var people = employees.Where(e => e.Age > 30).
    OrderBy(e => e.LastName).
    ThenBy(e => e.FirstName).
    ThenBy(e => e.Age);
```

Notice in the definition of the query expression pattern that `ThenBy` operates on a sequence returned by `OrderBy` or `ThenBy`. Those sequences can contain markers that enable `ThenBy` to operate on the sorted subranges when the sort keys are equal.

This transformation is not the same if the `orderby` clauses are expressed as different clauses. The following query sorts the sequence entirely by `LastName`, then sorts the entire sequence again by `FirstName`, and then sorts again by `Age`:

```
// Not correct. Sorts the entire sequence three times.
var people = from e in employees
             where e.Age > 30
             orderby e.LastName
             orderby e.FirstName
             orderby e.Age
             select e;
```

As separate queries, you could specify that any of the `orderby` clauses use descending order:

```
var people = from e in employees
             where e.Age > 30
             orderby e.LastName descending
             thenby e.FirstName
             thenby e.Age
             select e;
```

The `OrderBy` method creates a different sequence type as its output so that `thenby` clauses can be more efficient and so that the types are correct for the overall query. `OrderBy` cannot operate on an unordered sequence, only on a sorted sequence (typed as `O<T>` in the sample). Subranges are already sorted and marked. If you create your own `orderby` and `thenby` methods for a type, you must adhere to this rule. You'll need to add an identifier to each sorted subrange so that any subsequent `thenby` clause can work properly. `ThenBy` methods need to be typed to take the output of an `OrderBy` or `ThenBy` method and then sort each subrange correctly.

Everything I've said about `OrderBy` and `ThenBy` also applies to `OrderByDescending` and `ThenByDescending`. In fact, if your type has a custom version of any of those methods, you should almost always implement all four of them.

The remaining expression translations involve multiple steps. Those queries involve either groupings or multiple `from` clauses that introduce continuations. Query expressions that contain continuations are translated into nested queries. Then those nested queries are translated into methods. Following is a simple query with a continuation:

```
var results = from e in employees
              group e by e.Department into d
              select new { Department = d.Key,
              Size = d.Count() };
```

Before any other translations are performed, the continuation is translated into a nested query:

```
var results = from d in
    from e in employees group e by e.Department
    select new { Department = d.Key, Size = d.Count()};
```

Once the nested query is created, the methods translate into the following:

```
var results = employees.GroupBy(e => e.Department).
    Select(d => new { Department = d.Key, Size = d.Count()});
```

The foregoing query shows a `GroupBy` that returns a single sequence. The other `GroupBy` method in the query expression pattern returns a sequence of groups in which each group contains a key and a list of values:

```
var results = from e in employees
              group e by e.Department into d
              select new { Department = d.Key,
              Employees = d.AsEnumerable()};
```

That query maps to the following method calls:

```
var results2 = employees.GroupBy(e => e.Department).
    Select(d => new { Department = d.Key,
        Employees = d.AsEnumerable()});
```

`GroupBy` methods produce a sequence of key/value list pairs; the keys are the group selectors, and the values are the sequence of items in the group. The query `select` clause may create new objects for the values in each group. However, the output should always be a sequence of key/value pairs in which the value contains some element created by each item in the input sequence that belongs to that particular group.

The final methods to understand are `SelectMany`, `Join`, and `GroupJoin`. These three methods are complicated, because they work with multiple input sequences. The methods that implement these translations perform the enumerations across multiple sequences and then flatten the resulting sequences into a single output sequence. `SelectMany` performs a cross join on the two source sequences. For example, consider this query:

```
int[] odds = {1,3,5,7};
int[] evens = {2,4,6,8};
var pairs = from oddNumber in odds
            from evenNumber  in evens
            select new {oddNumber, evenNumber,
            Sum=oddNumber+evenNumber};
```

It produces a sequence having 16 elements:

```
1,2, 3
1,4, 5
```

```
1,6, 7
1,8, 9
3,2, 5
3,4, 7
3,6, 9
3,8, 11
5,2, 7
5,4, 9
5,6, 11
5,8, 13
7,2, 9
7,4, 11
7,6, 13
7,8, 15
```

Query expressions that contain multiple `select` clauses are translated into a `SelectMany` method call. The sample query would be translated into the following `SelectMany` call:

```
int[] odds = { 1, 3, 5, 7 };
int[] evens = { 2, 4, 6, 8 };
var values = odds.SelectMany(oddNumber => evens,
    (oddNumber, evenNumber) =>
    new { oddNumber, evenNumber,
    Sum = oddNumber + evenNumber });
```

The first parameter to `SelectMany` is a function that maps each element in the first source sequence to the sequence of elements in the second source sequence. The second parameter (the output selector) creates the projections from the pairs of items in both sequences.

`SelectMany()` iterates the first sequence. For each value in the first sequence, it iterates the second sequence, producing the result value from the pair of input values. The output selected is called for each element in a flattened sequence of every combination of values from both sequences. One possible implementation of `SelectMany` is as follows:

```
static IEnumerable<TOutput> SelectMany<T1, T2, TOutput>(
    this IEnumerable<T1> src,
    Func<T1, IEnumerable<T2>> inputSelector,
    Func<T1, T2, TOutput> resultSelector)
{
    foreach (T1 first in src)
```

```
    {
        foreach (T2 second in inputSelector(first))
            yield return resultSelector(first, second);
    }
}
```

The first input sequence is iterated. Then the second input sequence is iterated using the current value on the input sequence. That's important, because the input selector on the second sequence may depend on the current value in the first sequence. Then, as each pair of elements is generated, the result selector is called on each pair.

If your query has more expressions and if `SelectMany` does not create the final result, then `SelectMany` creates a tuple that contains one item from each input sequence. Sequences of that tuple are the input sequence for later expressions. For example, consider this modified version of the original query:

```
int[] odds = { 1, 3, 5, 7 };
int[] evens = { 2, 4, 6, 8 };
var values = from oddNumber in odds
             from evenNumber in evens
             where oddNumber > evenNumber
             select new { oddNumber, evenNumber,
             Sum = oddNumber + evenNumber };
```

It produces this `SelectMany` method call:

```
odds.SelectMany(oddNumber => evens,
    (oddNumber, evenNumber) =>
    new {oddNumber, evenNumber});
```

The full query is then translated into this statement:

```
var values = odds.SelectMany(oddNumber => evens,
    (oddNumber, evenNumber) =>
    new { oddNumber, evenNumber }).
    Where(pair => pair.oddNumber > pair.evenNumber).
    Select(pair => new {
        pair.oddNumber,
        pair.evenNumber,
        Sum = pair.oddNumber + pair.evenNumber });
```

You can see another interesting property in the way `SelectMany` gets treated when the compiler translates multiple `from` clauses into `SelectMany` method calls. `SelectMany` composes well. More than two `from` clauses will produce more than one `SelectMany()` method call. The resulting pair from the first `SelectMany()` call will be fed into the second `SelectMany()`, which will produce a triple. The triple will contain all combinations of all three sequences. Consider this query:

```
var triples = from n in new int[] { 1, 2, 3 }
              from s in new string[] { "one", "two",
                  "three" }
              from r in new string[] { "I", "II", "III" }
              select new { Arabic = n, Word = s, Roman = r };
```

It will be translated into the following method calls:

```
var numbers = new int[] {1,2,3};
var words = new string[] {"one", "two", "three"};
var romanNumerals = new string[] { "I", "II", "III" };
var triples = numbers.SelectMany(n => words,
    (n, s) => new { n, s}).
    SelectMany(pair => romanNumerals,
    (pair,n) =>
        new { Arabic = pair.n, Word = pair.s, Roman = n });
```

As you can see, you can extend from three to any arbitrary number of input sequences by applying more `SelectMany()` calls. These later examples also demonstrate how `SelectMany` can introduce anonymous types into your queries. The sequence returned from `SelectMany()` is a sequence of some anonymous type.

Now let's look at the two other translations you need to understand: `Join` and `GroupJoin`. Both are applied on join expressions. `GroupJoin` is always used when the join expression contains an `into` clause. `Join` is used when the join expression does not contain an `into` clause.

A join without an `into` looks like this:

```
var numbers = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var labels = new string[] { "0", "1", "2", "3", "4", "5" };
var query = from num in numbers
            join label in labels on num.ToString() equals
                label
            select new { num, label };
```

It translates into the following:

```
var query = numbers.Join(labels, num => num.ToString(),
    label => label, (num, label) => new { num, label });
```

The `into` clause creates a list of subdivided results:

```
var groups = from p in projects
             join t in tasks on p equals t.Parent
                 into projTasks
             select new { Project = p, projTasks };
```

That translates into a `GroupJoin`:

```
var groups = projects.GroupJoin(tasks,
    p => p, t => t.Parent, (p, projTasks) =>
        new { Project = p, TaskList = projTasks });
```

The entire process of converting all expressions into method calls is complicated and often takes several steps.

The good news is that for the most part, you can happily go about your work secure in the knowledge that the compiler does the correct translation. And because your type implements `IEnumerable<T>`, users of your type are getting the correct behavior.

But you may have that nagging urge to create your own version of one or more of the methods that implement the query expression pattern. Maybe your collection type is always sorted on a certain key, and you can short-circuit the `OrderBy` method. Maybe your type exposes lists of lists, and this means that you may find that `GroupBy` and `GroupJoin` can be implemented more efficiently.

More ambitiously, maybe you intend to create your own provider and you'll implement the entire pattern. That being the case, you need to understand the behavior of each query method and know what should go into your implementation. Refer to the examples, and make sure you understand the expected behavior of each query method before you embark on creating your own implementations.

Many of the custom types you define model some kind of collection. The developers who use your types will expect to use your collections in the same way that they use every other collection type, with the built-in query syntax. As long as you support the `IEnumerable<T>` interface for any type that models a collection, you'll meet that expectation. However, your types may be able to improve on the default implementation by using the internal specifics in your type. When you choose to do that, ensure that your type matches the contract from the query pattern in all forms.

Item 37: Prefer Lazy Evaluation Queries

When you define a query, you don't actually get the data and populate a sequence. You are actually defining only the set of steps that you will execute when you choose to iterate that query. This means that each time you execute a query, you perform the entire recipe from first principles. That's usually the right behavior. Each new enumeration produces new results, in what is called **lazy evaluation.** However, often that's not what you want. When you grab a set of variables, you want to retrieve them once and retrieve them now, in what is called **eager evaluation.**

Every time you write a query that you plan to enumerate more than once, you need to consider which behavior you want. Do you want a snapshot of your data, or do you want to create a description of the code you will execute in order to create the sequence of values?

This concept is a major change in the way you are likely accustomed to working. You probably view code as something that is executed immediately. However, with LINQ queries, you're injecting code into a method. That code will be invoked at a later time. More than that, if the provider uses expression trees instead of delegates, those expression trees can be combined later by combining new expressions into the same expression tree.

Let's start with an example to explain the difference between lazy and eager evaluation. The following bit of code generates a sequence and then iterates that sequence three times, with a pause between iterations.

```
private static IEnumerable<TResult>
    Generate<TResult>(int number, Func<TResult> generator)
{
    for (int i = 0; i < number; i++)
        yield return generator();
}

private static void LazyEvaluation()
{
    Console.WriteLine("Start time for Test One: {0}",
        DateTime.Now);
    var sequence = Generate(10, () => DateTime.Now);
```

```
            orderby p.GoalsScored
            select p).Skip(2).First();
```

I chose `First()` rather than `Take()` to emphasize that I wanted exactly one element, and not a sequence containing one element. Note that because I use `First()` instead of `FirstOrDefault()`, the compiler assumes that at least three forwards have scored goals.

However, once you start looking for an element in a specific position, it's likely that there is a better way to construct the query. Are there different properties you should be looking for? Should you look to see whether your sequence supports `IList<T>` and supports index operations? Should you rework the algorithm to find exactly the one item? You may find that other methods of finding results will give you much clearer code.

Many of your queries are designed to return one scalar value. Whenever you query for a single value, it's best to write your query to return a scalar value rather than a sequence of one element. Using `Single()` means that you expect to always find exactly one item. `SingleOrDefault()` means zero or one item. `First` and `Last` mean that you are pulling one item out of a sequence. Using any other method of finding one item likely means that you haven't written your query as well as you should have. It won't be as clear for developers using your code or maintaining it later.

Item 44: Prefer Storing Expression<> to Func<>

In Item 42 (earlier in this chapter) I briefly discuss how query providers such as LINQ to SQL examine queries before execution and translate them into their native format. LINQ to Objects, in contrast, implements queries by compiling lambda expressions into methods and creating delegates that access those methods. It's plain old code, but the access is implemented through delegates.

LINQ to SQL (and any other query provider) performs this magic by asking for query expressions in the form of a `System.Linq.Expressions.Expression` object. `Expression` is an abstract base class that represents an expression. One of the classes derived from `Expression` is `System.Linq.Expressions.Expression<TDelegate>`, where `TDelegate` is a delegate type. `Expression<TDelegate>` represents a lambda expression as a data structure. You can analyze it by using the `Body`, `NodeType`, and `Parameters` properties. Furthermore, you can compile it into a delegate by using the `Expression<TDelegate>.Compile()` method.

That makes `Expression<TDelegate>` more general than `Func<T>`. Simply put, `Func<T>` is a delegate that can be invoked. `Expression<TDelegate>` can be examined, or it can be compiled and then invoked in the normal way.

When your design includes the storage of lambda expressions, you'll have more options if you store them using `Expression<T>`. You don't lose any features; you simply have to compile the expression before invoking it:

```
Expression<Func<int, bool>> compound = val =>
    (val % 2 == 1) && (val > 300);
Func<int, bool> compiled = compound.Compile();
Console.WriteLine(compiled(501));
```

The `Expression` class provides methods that allow you to examine the logic of an expression. You can examine an expression tree and see the exact logic that makes up the expression. The C# team provides a reference implementation for examining an expression with the C# samples delivered with Visual Studio 2008. The Expression Tree Visualizer sample, which includes source code, provides code that examines each node type in an expression tree and displays the contents of that node. It recursively visits each subnode in the tree; this is how you would examine each node in a tree in an algorithm to visit and modify each node.

Working with expressions and expression trees instead of functions and delegates can be a better choice, because expressions have quite a bit more functionality: You can convert an `Expression` to a `Func`, and you can traverse expression trees, meaning that you can create modified versions of the expressions. You can use `Expression` to build new algorithms at runtime, something that is much harder to do with `Func`.

This habit helps you by letting you later combine expressions using code. In this way, you build an expression tree that contains multiple clauses. After building the code, you can call `Compile()` and create the delegate when you need it.

Here is one way to combine two expressions to form a larger expression:

```
Expression<Func<int, bool>> IsOdd = val => val % 2 == 1;
Expression<Func<int, bool>> IsLargeNumber = val => val > 300;

InvocationExpression callLeft = Expression.Invoke(IsOdd,
Expression.Constant(5));
InvocationExpression callRight = Expression.Invoke(
    IsLargeNumber,
    Expression.Constant(5));
```

```
BinaryExpression Combined =
    Expression.MakeBinary(ExpressionType.And,
    callLeft, callRight);

// Convert to a typed expression:
Expression<Func<bool>> typeCombined =
    Expression.Lambda<Func<bool>>(Combined);

Func<bool> compiled = typeCombined.Compile();
bool answer = compiled();
```

This code creates two small expressions and combines them into a single expression. Then it compiles the larger expression and executes it. If you're familiar with either CodeDom or Reflection.Emit, the `Expression` APIs can provide similar metaprogramming capabilities. You can visit expressions, create new expressions, compile them to delegates, and finally execute them.

Working with expression trees is far from simple. Because expressions are immutable, it's a rather extensive undertaking to create a modified version of an expression. You need to traverse every node in the tree and either (1) copy it to the new tree or (2) replace the existing node with a different expression that produces the same kind of result. Several implementations of expression tree visitors have been written, as samples and as open source projects. I don't add yet another version here. A Web search for "expression tree visitor" will find several implementations.

The `System.Linq.Expressions` namespace contains a rich grammar that you can use to build algorithms at runtime. You can construct your own expressions by building the complete expression from its components. The following code executes the same logic as the previous example, but here I build the lambda expression in code:

```
// The lambda expression has one parameter:
ParameterExpression parm = Expression.Parameter(
    typeof(int), "val");
// We'll use a few integer constants:
ConstantExpression threeHundred = Expression.Constant(300,
    typeof(int));
ConstantExpression one = Expression.Constant(1, typeof(int));
ConstantExpression two = Expression.Constant(2, typeof(int));
```

```
// Creates (val > 300)
BinaryExpression largeNumbers =
    Expression.MakeBinary(ExpressionType.GreaterThan,
    parm, threeHundred);

// creates (val % 2)
BinaryExpression modulo = Expression.MakeBinary(
    ExpressionType.Modulo,
    parm, two);
// builds ((val % 2) == 1), using modulo
BinaryExpression isOdd = Expression.MakeBinary(
    ExpressionType.Equal,
    modulo, one);
// creates ((val % 2) == 1) && (val > 300),
// using isOdd and largeNumbers
BinaryExpression lambdaBody =
    Expression.MakeBinary(ExpressionType.AndAlso,
    isOdd, largeNumbers);

// creates val => (val % 2 == 1) && (val > 300)
// from lambda body and parameter.
LambdaExpression lambda = Expression.Lambda(lambdaBody, parm);

// Compile it:
Func<int, bool> compiled = lambda.Compile() as
    Func<int, bool>;
// Run it:
Console.WriteLine(compiled(501));
```

Yes, using `Expression` to build your own logic is certainly more complicated than creating the expression from the `Func<>` definitions shown earlier. This kind of metaprogramming is an advanced topic. It's not the first tool you should reach for in your toolbox.

Even if you don't build and modify expressions, libraries you use might do so. You should consider using `Expression<>` instead of `Func<>` when your lambda expressions are passed to unknown libraries whose implementations might use the expression tree logic to translate your algorithms into a different format. Any `IQueryProvider`, such as LINQ to SQL, would perform that translation.

Also, you might create your own additions to your type that would be better served by expressions than by delegates. The justification is the same: You can always convert expressions into delegates, but you can't go the other way.

You may find that delegates are an easier way to represent lambda expressions, and conceptually they are. Delegates can be executed. Most C# developers understand them, and often they provide all the functionality you need. However, if your type will store expressions and passing those expressions to other objects is not under your control, or if you will compose expressions into more-complex constructs, then you should consider using expressions instead of funcs. You'll have a richer set of APIs that will enable you to modify those expressions at runtime and invoke them after you have examined them for your own internal purposes.

Index

C

D

E

F

G

J

K

L

X

Y