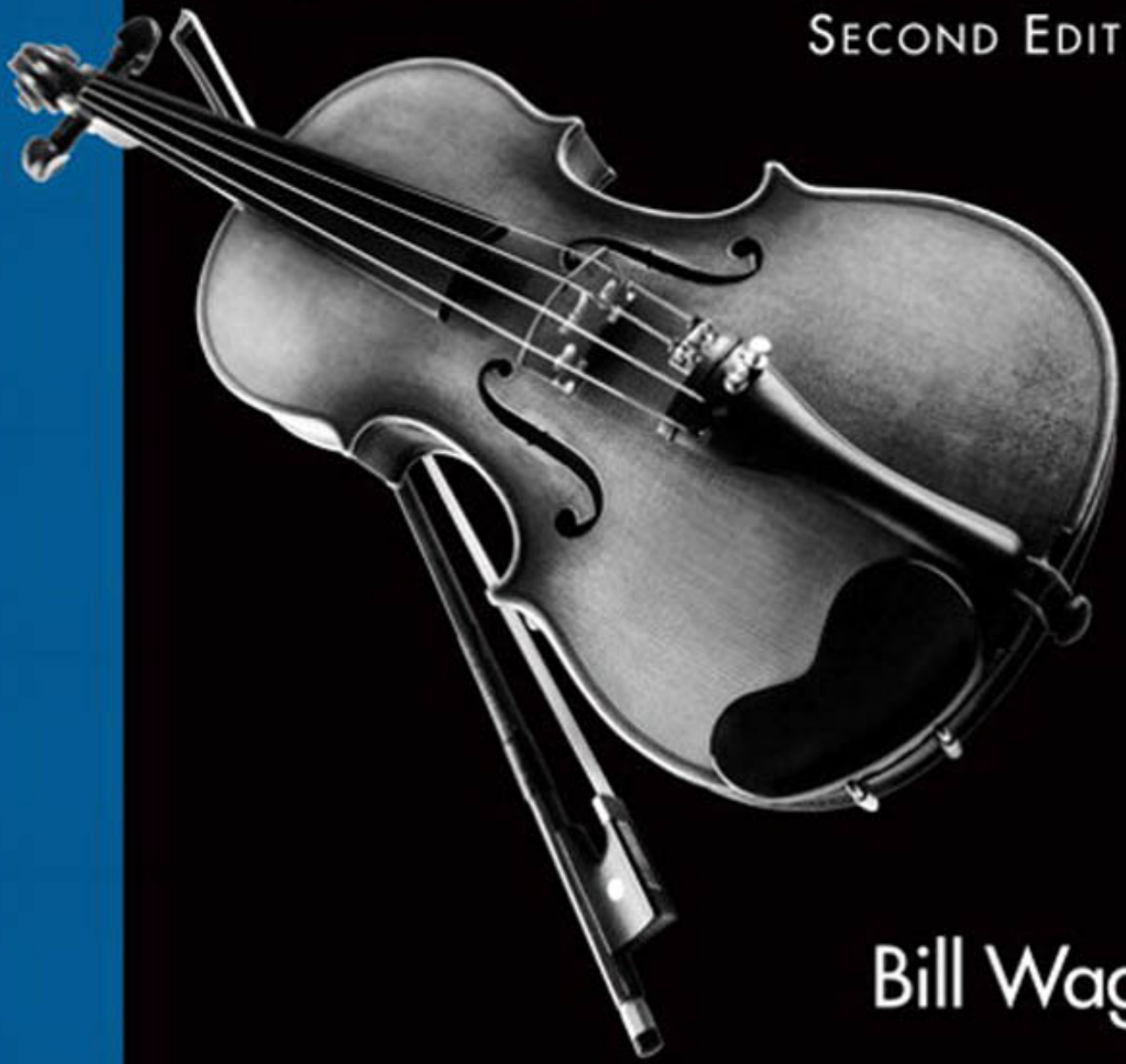
Effective SOFTWARE DEVELOPMENT SERIES

Scott Meyers, Consulting Editor

Effective C#

50 Specific Ways to Improve Your C#

SECOND EDITION



Bill Wagner

Library of Congress Cataloging-in-Publication Data

Wagner, Bill.
Effective C# : 50 specific ways to improve your C# / Bill Wagner.–2nd ed.
p. cm.
Includes index.
ISBN 978-0-321-65870-8 (pbk. : alk. paper)
1. C# (Computer program language) 2. Database management. 3. Microsoft .NET Framework. I. Title.
QA76.73.C154W343 2010
005.13'3–dc22

2009052199

ISBN-13: 978-0-321-65870-8
ISBN-10: 0-321-65870-1
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, March 2010

Introduction

The C# community is very different in 2010 than it was in 2004 when the first edition of *Effective C#* was published. There are many more developers using C#. A large contingent of the C# community is now seeing C# as their first professional language. They aren't approaching C# with a set of ingrained habits formed using a different language. The community has a much broader range of experience. New graduates all the way to professionals with decades of experience are using C#.

The C# language has also grown in the last five years. The language I covered in the first edition did not have generics, lambda expressions, LINQ, and many of the other features we now take for granted. C# 4.0 adds new features that change our toolset again. And yet, even with all the growth in the C# language, much of the original advice is as relevant now as it was in the C# 1.x days. Viewed in hindsight, the changes to the C# language appear to be natural and obvious extensions to what we had in C# 1.0. New editions give us new ways of solving problems, without invalidating previous idioms.

I organized this second edition of *Effective C#* by taking into account both the changes in the language and the changes in the C# community. *Effective C#* does not take you on a historical journey through the changes in the language. Rather, I provide advice on how to use the current C# language. The items that have been removed from this second edition are those that aren't as relevant in today's C# language. The new items cover the new language and framework features, and those practices the community has learned from building several versions of software products using C#. Overall, these items are a set of recommendations that will help you use C# 4.0 more effectively as a professional developer.

This book covers C# 4.0, but it is not an exhaustive treatment of the new language features. Like all books in the Effective Software Development Series, this book offers practical advice on how to use these features to solve problems you're likely to encounter every day. Many of the items are equally valid in the 3.0 and even earlier versions of the language.

Who Should Read This Book?

Effective C# was written for professional developers who use C# as part of their daily toolset. It assumes you are familiar with the C# syntax and the language's features. The second edition assumes you understand the new syntax added in C# 4.0, as well as the syntax available in the previous versions of the language. This book does not include tutorial instruction on language features. Instead, this book discusses how you can integrate all the features of the current version of the C# language into your everyday development.

In addition to language features, I assume you have some knowledge of the Common Language Runtime (CLR) and Just-In-Time (JIT) compiler.

About the Content

There are language constructs you'll use every day in almost every C# program you write. Chapter 1, "C# Language Idioms," covers those language idioms you'll use so often they should feel like well-worn tools in your hands. These are the building blocks of every type you create and every algorithm you implement.

Working in a managed environment doesn't mean the environment absolves you of all your responsibilities. You still must work with the environment to create correct programs that satisfy the stated performance requirements. It's not just about performance testing and performance tuning. Chapter 2, ".NET Resource Management," teaches you the design idioms that enable you to work with the environment to achieve those goals before detailed optimization begins.

In many ways, we write programs to satisfy human readers rather than a compiler. All the compiler cares about is that a program is valid. Our colleagues want to understand our intent as well. Chapter 3, "Expressing Designs in C#," discusses how the C# language can be applied to express your design intent. There are always several ways to solve a problem. The recommendations in Chapter 3 will help you choose the solution that best expresses your design intent to fellow developers.

C# is a small language, supported by a rich framework library. Chapter 4, "Working with the Framework," covers the portions of the .NET Base Class Library (BCL) that support your core algorithms. In addition, I cover

some of the common idioms that you'll encounter throughout the framework. Multicore processors are a way of life, and the Parallel Task Library provides a step forward in creating multithreaded programs on the .NET platform. I cover the most common practices for the Parallel Task Library in this chapter.

Chapter 5, "Dynamic Programming in C#," discusses how to use C# as a dynamic language. C# is a strongly typed, statically typed language. However, more and more programs contain both dynamic and static typing. C# provides ways for you to leverage dynamic programming idioms without losing the benefits of static typing throughout your entire program. You'll learn how to use dynamic features and how to avoid having dynamic types leak through your entire program.

Chapter 6, "Miscellaneous," covers those items that somehow continue to defy classification. These are the techniques you'll use often to create robust programs that are easier to maintain and extend.

Code Conventions

We no longer look at code in monochrome, and we shouldn't in books either. While it's impossible to replicate the experience of using a modern IDE on paper, I've tried to provide a better experience reading the code in the book. Where the medium supports it, the code samples use the standard Visual Studio IDE colors for all code elements. Where I am pointing to particular changes in samples, those changes are highlighted.

Showing code in a book still requires making some compromises for space and clarity. I've tried to distill the samples down to illustrate the particular point of the sample. Often that means eliding other portions of a class or a method. Sometimes that will include eliding error recovery code for space. Public methods should validate their parameters and other inputs, but that code is usually elided for space. Similar space considerations remove validation of method calls, and `try`/`finally` clauses that would often be included in complicated algorithms.

I also usually assume most developers can find the appropriate namespace when samples use one of the common namespaces. You can safely assume that every sample implicitly includes the following using statements:

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Dynamic;
using System.Threading;
```

Finally, I use the #region/#endregion directives to denote interface implementations. While that's not necessary, and some dislike the region directive in code, it does make it easy to see which methods implement interface methods in static text. Any other option would be nonstandard and take more space.

Providing Feedback

Despite my best efforts, and the efforts of the people who have reviewed the text, errors may have crept into the text or samples. If you believe you have found an error, please contact me at bill.wagner@srtsolutions.com. Errata will be posted at http://srtsolutions.com/blogs/effectivecsharp. Many of the items in this book, and *More Effective C#*, are the result of email conversations with other C# developers. If you have questions or comments about the recommendations, please contact me. Discussions of general interest will be covered on my blog at http://srtsolutions.com/blogs/billwagner.

Acknowledgments

There are many people to whom I owe thanks for their contributions to this book. I've been privileged to be part of an amazing C# community over the years. Everyone on the C# Insiders mailing list (whether inside or outside Microsoft) has contributed ideas and conversations that made this a better book.

I must single out a few members of the C# community who directly helped me with ideas, and with turning ideas into concrete recommendations. Conversations with Charlie Calvert, Eric DeCarufel, Justin Etheredge, Marc Gravell, Mike Gold, and Doug Holland are the basis for many new ideas in this edition.

I also had great email conversations with Stephen Toub and Michael Wood on the Parallel Task Library and its implications on C# idioms.

I had a wonderful team of technical reviewers for this edition. Jason Bock, Claudio Lassala, and Tomas Petricek pored over the text and the samples to

ensure the quality of the book you now hold. Their reviews were thorough and complete, which is the best anyone can hope for. Beyond that, they added recommendations that helped me explain many of the topics better.

The team at Addison-Wesley is a dream to work with. Joan Murray is a fantastic editor, taskmaster, and the driving force behind anything that gets done. She leans on Olivia Basegio heavily, and so do I. Their contributions created the quality of the finished manuscript from the front cover to the back, and everything in between. Curt Johnson and Brandon Prebynski continue to do an incredible job marketing technical content. No matter what format you chose, Curt and Brandon have had something to do with its existence for this book. Geneil Breeze poured over the entire manuscript improving explanations and clarifying the wording in several places.

It's an honor, once again, to be part of Scott Meyer's series. He goes over every manuscript and offers suggestions and comments for improvement. He is incredibly thorough, and his experience in software, although not in C#, means he finds any areas where I haven't explained an item clearly or fully justified a recommendation. His feedback, as always, is invaluable.

I've also had the privilege of bouncing ideas off the other consultants at SRT Solutions. From the most experienced to the youngest, they are an incredibly smart group of people with great insight. They are also not afraid to express their opinions. Countless conversations with Ben Barefield, Dennis Burton, Marina Fedner, Alex Gheith, Darrell Hawley, Chris Marinos, Dennis Matveyev, Anne Marsan, Dianne Marsh, Charlie Sears, Patrick Steele, Mike Woelmer, and Jay Wren sparked ideas and samples. Later conversations helped clarify how to explain and justify different recommendations.

As always, my family gave up time with me so that I could finish this manuscript. My children Lara, Sarah, and Scott, put up with the times I hid in the home office and didn't join in other activities. My wife, Marlene, gave up countless hours while I went off to write or create samples. Without their support, I never would have finished this or any other book. Nor would it be as satisfying to finish.

About the Author

With more than twenty years of experience, Bill Wagner, SRT Solutions cofounder, is a recognized expert in software design and engineering,

specializing in C#, .NET, and the Azure platform. He serves as Michigan's Regional Director for Microsoft and is a multiyear winner of Microsoft's MVP award. An internationally recognized writer, Bill is the author of the first edition of this book and *More Effective C#* (Addison-Wesley, 2009) and currently writes a column on the Microsoft C# Developer Center. Bill earned a B.S. in computer science from the University of Illinois at Champaign-Urbana.

case in which an upgrade to a base class now collides with a member that you previously declared in your class.

Of course, over time, your users might begin wanting to use the BaseWidget .NormalizeValues() method. Then you are back to the original problem: two methods that look the same but are different. Think through all the long-term ramifications of the `new` modifier. Sometimes, the short-term inconvenience of changing your method is still better.

The new modifier must be used with caution. If you apply it indiscriminately, you create ambiguous method calls in your objects. It's for the special case in which upgrades in your `base` class cause collisions in your class. Even in that situation, think carefully before using it. Most importantly, don't use it in any other situations.

Item 34: Avoid Overloading Methods Defined in Base Classes

When a base class chooses the name of a member, it assigns the semantics to that name. Under no circumstances may the derived class use the same name for different purposes. And yet, there are many other reasons why a derived class may want to use the same name. It may want to implement the same semantics in a different way, or with different parameters. Sometimes that's naturally supported by the language: Class designers declare virtual functions so that derived classes can implement semantics differently. Item 33 covered why using the `new` modifier could lead to hard-to-find bugs in your code. In this item, you'll learn why creating overloads of methods that are defined in a base class leads to similar issues. You should not overload methods declared in a base class.

The rules for overload resolution in the C# language are necessarily complicated. Possible candidate methods might be declared in the target class, any of its base classes, any extension method using the class, and interfaces it implements. Add generic methods and generic extension methods, and it gets very complicated. Throw in optional parameters, and I'm not sure anyone could know exactly what the results will be. Do you really want to add more complexity to this situation? Creating overloads for methods declared in your base class adds more possibilities to the best overload match. That increases the chance of ambiguity. It increases the chance that your interpretation of the spec is different than the compilers, and it will certainly confuse your users. The solution is simple: Pick a different method name. It's your class, and you certainly have enough brilliance to

come up with a different name for a method, especially if the alternative is confusion for everyone using your types.

The guidance here is straightforward, and yet people always question if it really should be so strict. Maybe that's because overloading sounds very much like overriding. Overriding virtual methods is such a core principle of object-oriented languages; that's obviously not what I mean. Overloading means creating multiple methods with the same name and different parameter lists. Does overloading base class methods really have that much of an effect on overload resolution? Let's look at the different ways where overloading methods in the base class can cause issues.

There are a lot of permutations to this problem. Let's start simple. The interplay between overloads in base classes has a lot to do with base and derived classes used for parameters. For all the following examples, any class that begins with "B" is the base class, and any class that begins with "D" is the derived class. The samples use this class hierarchy for parameters:

```
public class B2 { }
public class D2 : B2 {}
```

Here's a class with one method, using the derived parameter (D2):

```
public class B
{
    public void Foo(D2 parm)
    {
        Console.WriteLine("In B.Foo");
    }
}
```

Obviously, this snippet of code writes "In B.Foo":

```
var obj1 = new D();
obj1.Bar(new D2());
```

Now, let's add a new derived class with an overloaded method:

```
public class D : B
{
    public void Foo(B2 parm)
    {
        Console.WriteLine("In D.Foo");
    }
}
```

Now, what happens when you execute this code?

```
var obj2 = new D();
obj2.Foo(new D2());
obj2.Foo(new B2());
```

Both lines print "in D.Foo". You always call the method in the derived class. Any number of developers would figure that the first call would print "in B.Foo". However, even the simple overload rules can be surprising. The reason both calls resolve to D.Foo is that when there is a candidate method in the most derived compile-time type, that method is the better method. That's still true when there is even a better match in a base class. Of course, this is very fragile. What do you suppose this does:

```
B obj3 = new D();
obj3.Foo(new D2());
```

I chose the words above very carefully because obj3 has the compile-time type of B (your Base class), even though the runtime type is D (your Derived class). Foo isn't virtual; therefore, obj3.Foo() must resolve to B.Foo.

If your poor users actually want to get the resolution rules they might expect, they need to use casts:

```
var obj4 = new D();
((B)obj4).Foo(new D2());
obj4.Foo(new B2());
```

If your API forces this kind of construct on your users, you've failed. You can easily add a bit more confusion. Add one method to your base class, B:

```
public class B
{
    public void Foo(D2 parm)
    {
        Console.WriteLine("In B.Foo");
    }

    public void Bar(B2 parm)
    {
        Console.WriteLine("In B.Bar");
    }
}
```

Clearly, the following code prints "In B.Bar":

```
var obj1 = new D();
obj1.Bar(new D2());
```

Now, add a different overload, and include an optional parameter:

```
public class D : B
{
    public void Foo(B2 parm)
    {
        Console.WriteLine("In D.Foo");
    }

    public void Bar(B2 parm1, B2 parm2 = null)
    {
        Console.WriteLine("In D.Bar");
    }
}
```

Hopefully, you've already seen what will happen here. This same snippet of code now prints "In D.Bar" (you're calling your derived class again):

```
var obj1 = new D();
obj1.Bar(new D2());
```

The only way to get at the method in the base class (again) is to provide a cast in the calling code.

These examples show the kinds of problems you can get into with one parameter method. The issues become more and more confusing as you add parameters based on generics. Suppose you add this method:

```
public class B
{
    public void Foo(D2 parm)
    {
        Console.WriteLine("In B.Foo");
    }

    public void Bar(B2 parm)
    {
        Console.WriteLine("In B.Bar");
    }
```

```
    public void Foo2(IEnumerable<D2> parm)
    {
        Console.WriteLine("In B.Foo2");
    }
}
```

Then, provide a different overload in the derived class:

```
public class D : B
{
    public void Foo(B2 parm)
    {
        Console.WriteLine("In D.Foo");
    }

    public void Bar(B2 parm1, B2 parm2 = null)
    {
        Console.WriteLine("In D.Bar");
    }
    public void Foo2(IEnumerable<B2> parm)
    {
        Console.WriteLine("In D.Foo2");
    }
}
```

Call Foo2 in a manner similar to before:

```
var sequence = new List<D2> { new D2(), new D2() };
var obj2 = new D();

obj2.Foo2(sequence);
```

What do you suppose gets printed this time? If you've been paying attention, you'd figure that "In D.Foo2" gets printed. That answer gets you partial credit. That is what happens in C# 4.0. Starting in C# 4.0, generic interfaces support covariance and contravariance, which means D.Foo2 is a candidate method for an IEnumerable<D2> when its formal parameter type is an IEnumerable<B2>. However, earlier versions of C# do not support generic variance. Generic parameters are invariant. In those versions, D.Foo2 is not a candidate method when the parameter is an IEnumerable<D2>. The only candidate method is B.Foo2, which is the correct answer in those versions.

The code samples above showed that you sometimes need casts to help the compiler pick the method you want in many complicated situations. In the real world, you'll undoubtedly run into situations where you need to use casts because class hierarchies, implemented interfaces, and extension methods have conspired to make the method you want, not the method the compiler picks as the "best" method. But the fact that real-world situations are occasionally ugly does not mean you should add to the problem by creating more overloads yourself.

Now you can amaze your friends at programmer cocktail parties with a more in-depth knowledge of overload resolution in C#. It can be useful information to have, and the more you know about your chosen language the better you'll be as a developer. But don't expect your users to have the same level of knowledge. More importantly, don't rely on everyone having that kind of detailed knowledge of how overload resolution works to be able to use your API. Instead, don't overload methods declared in a base class. It doesn't provide any value, and it will only lead to confusion among your users.

Item 35: Learn How PLINQ Implements Parallel Algorithms

This is the item where I wish I could say that parallel programming is now as simple as adding AsParallel() to all your loops. It's not, but PLINQ does make it much easier than it was to leverage multiple cores in your programs and still have programs that are correct. It's by no means trivial to create programs that make use of multiple cores, but PLINQ makes it easier.

You still have to understand when data access must be synchronized. You still need to measure the effects of parallel and sequential versions of the methods declared in ParallelEnumerable. Some of the methods involved in LINQ queries can execute in parallel very easily. Others force more sequential access to the sequence of elements—or, at least, require the complete sequence (like Sort). Let's walk through a few samples using PLINQ and learn what works well, and where some of the pitfalls still exist. All the samples and discussions for this item use LINQ to Objects. The title even calls out "Enumerable," not "Queryable". PLINQ really won't help you parallelize LINQ to SQL, or Entity Framework algorithms. That's not really a limiting feature, because those implementations leverage the parallel database engines to execute queries in parallel.

```
                typeof(DynamicDictionary2).GetMethod(methodName),
                parameters),
            BindingRestrictions.GetTypeRestriction(Expression,
                LimitType));
    return getDictionaryEntry;
}
```

Before you go off and think this isn't that hard, let me leave you with some thoughts from the experience of writing this code. This is about as simple as a dynamic object can get. You have two APIs: property `get`, property `set`. The semantics are very easy to implement. Even with this very simple behavior, it was rather difficult to get right. Expression trees are hard to debug. They are hard to get right. More sophisticated dynamic types would have much more code. That would mean much more difficulty getting the expressions correct.

Furthermore, keep in mind one of the opening remarks I made on this section: Every invocation on your dynamic object will create a new DynamicMetaObject and invoke one of the Bind members. You'll need to write these methods with an eye toward efficiency and performance. They will be called a lot, and they have much work to do.

Implementing dynamic behavior can be a great way to approach some of your programming challenges. When you look at creating dynamic types, your first choice should be to derive from System.Dynamic.DynamicObject. On those occasions where you must use a different base class, you can implement IDynamicMetaObjectProvider yourself, but remember that this is a complicated problem to take on. Furthermore, any dynamic types involve some performance costs, and implementing them yourself may make those costs greater.

Item 42: Understand How to Make Use of the Expression API

.NET has had APIs that enable you to reflect on types or to create code at runtime. The ability to examine code or create code at runtime is very powerful. There are many different problems that are best solved by inspecting code or dynamically generating code. The problem with these APIs is that they are very low level and quite difficult to work with. As developers, we crave an easier way to dynamically solve problems.

Now that C# has added LINQ and dynamic support, you have a better way than the classic Reflection APIs: expressions and expression trees. Expres-

sions look like code. And, in many uses, expressions do compile down to delegates. However, you can ask for expressions in an Expression format. When you do that, you have an object that represents the code you want to execute. You can examine that expression, much like you can examine a class using the Reflection APIs. In the other direction, you can build an expression to create code at runtime. Once you create the expression tree you can compile and execute the expression. The possibilities are endless. After all, you are creating code at runtime. I'll describe two common tasks where expressions can make your life much easier.

The first solves a common problem in communication frameworks. The typical workflow for using WCF, remoting, or Web services is to use some code generation tool to generate a client-side proxy for a particular service. It works, but it is a somewhat heavyweight solution. You'll generate hundreds of lines of code. You'll need to update the proxy whenever the server gets a new method, or changes parameter lists. Instead, suppose you could write something like this:

```
var client = new ClientProxy<IService>();
var result = client.CallInterface<string>(
    srver => srver.DoWork(172));
```

Here, the ClientProxy<T> knows how to put each argument and method call on the wire. However, it doesn't know anything about the service you're actually accessing. Rather than relying on some out of band code generator, it will use expression trees and generics to figure out what method you called, and what parameters you used.

The CallInterface() method takes one parameter, which is an Expression <Func<T, TResult>>. The input parameter (of type T) represents an object that implements IService. TResult, of course, is whatever the particular method returns. The parameter is an expression, and you don't even need an instance of an object that implements IService to write this code. The core algorithm is in the CallInterface() method.

```
public TResult CallInterface<TResult>(Expression<
    Func<T, TResult>> op)
{
    var exp = op.Body as MethodCallExpression;
    var methodName = exp.Method.Name;
    var methodInfo = exp.Method;
```

```
    var allParameters = from element in exp.Arguments
                        select processArgument(element);
    Console.WriteLine("Calling {0}", methodName);

    foreach (var parm in allParameters)
        Console.WriteLine(
            "\tParameter type = {0}, Value = {1}",
            parm.Item1, parm.Item2);

    return default(TResult);
}

private Tuple<Type, object> processArgument(Expression
    element)
{
    object argument = default(object);
    LambdaExpression l = Expression.Lambda(
        Expression.Convert(element, element.Type));
    Type parmType = l.ReturnType;
    argument = l.Compile().DynamicInvoke();
    return Tuple.Create(parmType, argument);
}
```

Starting from the beginning of CallInterface, the first thing this code does is look at the body of the expression tree. That's the part on the right side of the lambda operator. Look back at the example where I used CallInterface(). That example called it with srver.DoWork(172). It is a MethodCallExpression, and that MethodCallExpression contains all the information you need to understand all the parameters and the method name invoked. The method name is pretty simple: It's stored in the Name property of the Method property. In this example, that would be 'DoWork'. The LINQ query processes any and all parameters to this method. The interesting work in is processArgument.

processArgument evaluates each parameter expression. In the example above, there is only one argument, and it happens to be a constant, the value 172. However, that's not very robust, so this code takes a different strategy. It's not robust, because any of the parameters could be method calls, property or indexer accessors, or even field accessors. Any of the method calls could also contain parameters of any of those types. Instead of trying to parse everything, this method does that hard work by leveraging

the LambdaExpression type and evaluating each parameter expression. Every parameter expression, even the ConstantExpression, could be expressed as the return value from a lambda expression. ProcessArgument() converts the parameter to a LambdaExpression. In the case of the constant expression, it would convert to a lambda that is the equivalent of () => 172. This method converts each parameter to a lambda expression because a lambda expression can be compiled into a delegate and that delegate can be invoked. In the case of the parameter expression, it creates a delegate that returns the constant value 172. More complicated expressions would create more complicated lambda expressions.

Once the lambda expression has been created, you can retrieve the type of the parameter from the lambda. Notice that this method does not perform any processing on the parameters. The code to evaluate the parameters in the lambda expression would be executed when the lambda expression is invoked. The beauty of this is that it could even contain other calls to CallInterface(). Constructs like this just work:

```
client.CallInterface(srver => srver.DoWork(
    client.CallInterface(srv => srv.GetANumber())));
```

This technique shows you how you can use expression trees to determine at runtime what code the user wishes to execute. It's hard to show in a book, but because ClientProxy<T> is a generic class that uses the service interface as a type parameter, the CallInterface method is strongly typed. The method call in the lambda expression must be a member method defined on the server.

The first example showed you how to parse expressions to convert code (or at least expressions that define code) into data elements you can use to implement runtime algorithms. The second example shows the opposite direction: Sometimes you want to generate code at runtime. One common problem in large systems is to create an object of some destination type from some related source type. For example, your large enterprise may contain systems from different vendors each of which has a different type defined for a contact (among other types). Sure, you could type methods by hand, but that's tedious. It would be much better to create some kind of type that "figures out" the obvious implementation. You'd like to just write this code:

```
var converter = new Converter<SourceContact,
    DestinationContact>();
DestinationContact dest2 = converter.ConvertFrom(source);
```

You'd expect the converter to copy every property from the source to the destination where the properties have the same name and the source object has a public `get` accessor and the destination type has a public `set` accessor. This kind of runtime code generation can be best handled by creating an expression, and then compiling and executing it. You want to generate code that does something like this:

```
// Not legal C#, explanation only
TDest ConvertFromImaginary(TSource source)
{
    TDest destination = new TDest();
    foreach (var prop in sharedProperties)
        destination.prop = source.prop;
    return destination;
}
```

You need to create an expression that creates code that executes the pseudo code written above. Here's the full method to create that expression and compile it to a function. Immediately following the listing, I'll explain all the parts of this method in detail. You'll see that while it's a bit thorny at first, it's nothing you can't handle.

```
private void createConverterIfNeeded()
{
    if (converter == null)
    {
        var source = Expression.Parameter(typeof(TSource),
            "source");
        var dest = Expression.Variable(typeof(TDest),
            "dest");

        var assignments = from srcProp in
                          typeof(TSource).GetProperties(
                              BindingFlags.Public |
                              BindingFlags.Instance)
                          where srcProp.CanRead
                          let destProp = typeof(TDest).
                              GetProperty(
                              srcProp.Name,
                              BindingFlags.Public |
                              BindingFlags.Instance)
```

```
                                where (destProp != null) &&
                                    (destProp.CanWrite)
                                select Expression.Assign(
                                    Expression.Property(dest,
                                        destProp),
                                    Expression.Property(source,
                                        srcProp));

            // put together the body:
            var body = new List<Expression>();
            body.Add(Expression.Assign(dest,
                Expression.New(typeof(TDest))));
            body.AddRange(assignments);
            body.Add(dest);

            var expr =
                Expression.Lambda<Func<TSource, TDest>>(
                    Expression.Block(
                    new[] { dest }, // expression parameters
                    body.ToArray() // body
                    ),
                    source  // lambda expression
                );

            var func = expr.Compile();
            converter = func;
        }
    }
```

This method creates code that mimics the pseudo code shown before. First, you declare the parameter:

```
var source = Expression.Parameter(typeof(TSource), "source");
```

Then, you have to declare a local variable to hold the destination:

```
var dest = Expression.Variable(typeof(TDest), "dest");
```

The bulk of the method is the code that assigns properties from the source object to the destination object. I wrote this code as a LINQ query. The source sequence of the LINQ query is the set of all public instance properties in the source object where there is a `get` accessor:

```
from srcProp in typeof(TSource).GetProperties(
                BindingFlags.Public | BindingFlags.Instance)
                where srcProp.CanRead
```

The `let` declares a local variable that holds the property of the same name in the destination type. It may be null, if the destination type does not have a property of the correct type:

```
let destProp = typeof(TDest).GetProperty(
               srcProp.Name,
               BindingFlags.Public | BindingFlags.Instance)
               where (destProp != null) &&
                     (destProp.CanWrite)
```

The projection of the query is a sequence of assignment statements that assigns the property of the destination object to the value of the same property name in the source object:

```
select Expression.Assign(
    Expression.Property(dest, destProp),
    Expression.Property(source, srcProp));
```

The rest of the method builds the body of the lambda expression. The Block() method of the Expression class needs all the statements in an array of Expression. The next step is to create a List<Expression> where you can add all the statements. The list can be easily converted to an array.

```
var body = new List<Expression>();
body.Add(Expression.Assign(dest,
    Expression.New(typeof(TDest))));
body.AddRange(assignments);
body.Add(dest);
```

Finally, it's time to build a lambda that returns the destination object and contains all the statements built so far:

```
var expr =
    Expression.Lambda<Func<TSource, TDest>>(
        Expression.Block(
        new[] { dest }, // expression parameters
        body.ToArray() // body
        ),
        source  // lambda expression
    );
```

That's all the code you need. Time to compile it and turn it into a delegate that you can call:

```
var func = expr.Compile();
converter = func;
```

That is complicated, and it's not the easiest to write. You'll often find compiler-like errors at runtime until you get the expressions built correctly. It's also clearly not the best way to approach simple problems. But even so, the Expression APIs are much simpler than their predecessors in the Reflection APIs. That's when you should use the Expression APIs: When you think you want to use reflection, try to solve the problem using the Expression APIs instead.

The Expression APIs can be used in two very different ways: You can create methods that take expressions as parameters, which enables you to parse those expressions and create code based on the concepts behind the expressions that were called. Also, the Expression APIs enable you to create code at runtime. You can create classes that write code, and then execute the code they've written. It's a very powerful way to solve some of the more difficult general purpose problems you'll encounter.

Item 43: Use Expressions to Transform Late Binding into Early Binding

Late binding APIs use the symbol text to do their work. Compiled APIs do not need that information, because the compiler has already resolved symbol references. The Expression API enables you to bridge both worlds. Expression objects contain a form of abstract symbol tree that represents the algorithms you want to execute. You can use the Expression API to execute that code. You can also examine all the symbols, including the names of variables, methods, and properties. You can use the Expression APIs to create strongly typed compiled methods that interact with portions of the system that rely on late binding, and use the names of properties or other symbols.

One of the most common examples of a late binding API is the property notification interfaces used by Silverlight and WPF. Both Silverlight and WPF were designed to respond to bound properties changing so that user interface elements can respond when data elements change underneath the user interface. Of course, there is no magic; there is only code that you

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

X