

SECOND EDITION



IN DEPTH

MEAP

Sheet



MANNING

- Chapter 1: The changing face of C# development
- Chapter 2: Core foundations: Building on C# 1
- Chapter 3: Parameterized typing with generics
- Chapter 4: Saying nothing with nullable types
- Chapter 5: Fast-tracked delegates
- Chapter 6: Implementing iterators the easy way
- Chapter 7: Concluding C# 2: the final features
- Chapter 8: Cutting fluff with a smart compiler
- Chapter 9: Lambda expressions and expression trees
- Chapter 10: Extension methods
- Chapter 11: Query expressions and LINQ to Objects
- Chapter 12: LINQ beyond collections
- Chapter 13: Minor changes to simplify code
- Chapter 14: Dynamic binding in a static language
- Chapter 15: Framework features which change coding styles
- Chapter 16: Whither now?



MEAP Edition
Manning Early Access Program

Copyright 2009 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

13. Minor changes to simplify code	1
Optional parameters and named arguments	1
Optional parameters	2
Named arguments	7
Putting the two together	10
Improvements for COM interoperability	14
The horrors of automating Word before C# 4	14
The revenge of default parameters and named arguments	15
When is a ref parameter not a ref parameter?	16
Linking Primary Interop Assemblies	17
Generic variance for interfaces and delegates	20
Types of variance: covariance and contravariance	21
Using variance in interfaces	22
Using variance in delegates	25
Complex situations	25
Limitations and notes	27
Summary	29
14. Dynamic binding in a static language	31
What? When? Why? How?	31
What is dynamic typing?	32
When is dynamic typing useful, and why?	32
How does C# 4 provide dynamic typing?	33
The 5 minute guide to dynamic	34
Examples of dynamic typing	36
COM in general, and Microsoft Office in particular	36
Dynamic languages such as IronPython	38
Reflection	42
Looking behind the scenes	46
Introducing the Dynamic Language Runtime	47
DLR core concepts	50
How the C# compiler handles dynamic	52
The C# compiler gets even smarter	55
Restrictions on dynamic code	57
Implementing dynamic behavior	60
Using ExpandableObject	60
Using DynamicObject	64
Implementing IDynamicMetaObjectProvider	70
Summary	70

Chapter 13. Minor changes to simplify code

Just as in previous versions, C# 4 has a few minor features which don't really merit individual chapters to themselves. In fact, there's only one really *big* feature in C# 4 - dynamic typing - which we'll cover in the next chapter. The changes we'll cover here just make C# that little bit more pleasant to work with, particularly if you work with COM on a regular basis. We'll be looking at:

- Optional parameters (so that callers don't need to specify everything)
- Named arguments (to make code clearer, and to help with optional parameters)
- Streamlining `ref` parameters in COM (a simple compiler trick to remove drudgery)
- Embedding COM Primary Interop Assemblies (leading to simpler deployment)
- Generic variance for interfaces and delegates (in limited situations)

Will any of those make your heart race with excitement? It's unlikely. They're nice features all the same, and make some patterns simpler (or just more realistic to implement). Let's start off by looking at how we call methods.

Optional parameters and named arguments

These are perhaps the Batman and Robin¹ features of C# 4. They're distinct, but usually seen together. I'm going to keep them apart for the moment so we can examine each in turn, but then we'll use them together for some more interesting examples.

Parameters and Arguments

This section obviously talks about parameters and arguments a lot. In casual conversation, the two terms are often used interchangeably, but I'm going to use them in line with their formal definitions. Just to remind you, a *parameter* (also known as a *formal parameter*) is the variable which is part of the method or indexer declaration. An *argument* is an expression used when calling the method or indexer. So for example, consider this snippet:

```
void Foo(int x, int y)
{
    // Do something with x and y
}
...
int a = 10;
Foo(a, 20);
```

Here the parameters are `x` and `y`, and the arguments are `a` and `20`.

We'll start off by looking at optional parameters.

¹Or Cavalleria Rusticana and Pagliacci if you're feeling more highly cultured

Optional parameters

Visual Basic has had optional parameters for ages, and they've been in the CLR from .NET 1.0. The concept is as obvious as it sounds: some parameters are optional, so they don't have to be explicitly specified by the caller. Any parameter which hasn't been specified as an argument by the caller is given a default value.

Motivation

Optional parameters are usually used when there are several values required for an operation (often creating a new object), where the same values are used a lot of the time. For example, suppose you wanted to read a text file, you might want to provide a method which allows the caller to specify the name of the file and the encoding to use. The encoding is almost always UTF-8 though, so it's nice to be able to just use that automatically if it's all you need.

Historically the idiomatic way of allowing this in C# has been to use method overloading, with one "canonical" method and others which call it, providing default values. For instance, you might create methods like this:

```
public IList<Customer> LoadCustomers(string filename,
                                     Encoding encoding)
{
    ... ❶
}

public IList<Customer> LoadCustomers(string filename)
{
    return LoadCustomers(filename, Encoding.UTF8); ❷
}
```

❶ Do real work here

❷ Default to UTF-8

This works fine for a single parameter, but it becomes trickier when there are multiple options. Each extra option doubles the number of possible overloads, and if two of them are of the same type you can have problems due to trying to declare multiple methods with the same signature. Often the same set of overloads is also required for multiple parameter types. For example, the `XmlReader.Create()` method can create an `XmlReader` from a `Stream`, a `TextReader` or a `string` - but it also provides the option of specifying an `XmlReaderSettings` and other arguments. Due to this duplication, there are twelve overloads for the method. This could be significantly reduced with optional parameters. Let's see how it's done.

Declaring optional parameters and omitting them when supplying arguments

Making a parameter optional is as simple as supplying a default value for it. Figure 13.X shows a method with three parameters: two are optional, one is required². Listing 13.X implements the method and called in three slightly different ways.

²Note for editors, typesetters and MEAP readers: the figure should be to one side of the text, so there isn't the jarring "figure then listing" issue. Quite how we build that as a PDF remains to be seen.

Figure 13.1. Declaring optional parameters

void Foo (int x, int y=10)

required parameter

optional parameter

Example 13.1. Declaring a method with optional parameters and calling

```
static void Dump(int x, int y = 20, int z = 30) ❶
{
    Console.WriteLine("{0} {1} {2}", x, y, z);
}
...
Dump(1, 2, 3); ❷
Dump(1, 2); ❸
Dump(1); ❹
```

❶❶ Declares method with optional parameters**❷❷** Calls method with all arguments**❸❸** Omits one argument**❹❹** Omits two arguments

The *optional parameters* are the ones with default values specified **❶**. If the caller doesn't specify *y*, its initial value will be 20, and likewise *z* has a default value of 30. The first call **❷** explicitly specifies all the arguments; the remaining calls (**❸** and **❹**) omit one or two arguments respectively, so the default values are used. When there is one argument "missing" the compiler assumes it's the final parameter which has been omitted - then the penultimate one, and so on. The output is therefore:

```
x=1 y=2 z=3
x=1 y=2 z=30
x=1 y=20 z=30
```

Note that although the compiler *could* use some clever analysis of the types of the optional parameters and the arguments to work out what's been left out, it doesn't: it assumes that you are supplying arguments in the same order as the parameters³. This means that the following code is invalid:

```
static void TwoOptionalParameters(int x = 10,
                                   string y = "default")
{
    Console.WriteLine("x={0} y={1}", x, y);
}
```

³Unless you're using named arguments, of course - we'll learn about those soon.

```
...  
TwoOptionalParameters("second parameter"); ❶
```

❶ Error!

This tries to call the `TwoOptionalParametersMethod` specifying a string for the *first* argument. There's no overload with a first parameter which is convertible from a string, so the compiler issues an error. This is a good thing - overload resolution is tricky enough (particularly when generic type inference gets involved) without the compiler trying all kinds of different permutations to find something you *might* be trying to call. If you want to omit a value for one optional parameter but specify a later one, you need to use named arguments.

Restrictions on optional parameters

Now, there are a few rules for optional parameters. All optional parameters have to come after required parameters. The exception to this is a *parameter array* (as declared with the `params` modifier) which still has to come at the end of a parameter list, but can come after optional parameters. A parameter array can't be declared as an optional parameter - if the caller doesn't specify any values for it, an empty array will be used instead. Optional parameters can't have `ref` or `out` modifiers either.

The type of the optional parameter can be any type, but there are restrictions on the default value specified. You can always use a constant, including literals, `null`, references to other `const` members, and the `default(...)` operator. Additionally, for value types, you can call the parameterless constructor, although this is equivalent to using the `default(...)` operator anyway. There has to be an implicit conversion from the specified value to the parameter type, but this must *not* be a user-defined conversion. Here are some examples of valid declarations:

- `Foo(int x, int y = 10)` - numeric literals are allowed
- `Foo(decimal x = 10)` - implicit built-in conversion from `int` to `decimal` is allowed
- `Foo(string name = "default")` - string literals are allowed
- `Foo(DateTime dt = new DateTime())` - "zero" value of `DateTime`
- `Foo(DateTime dt = default(DateTime))` - another way of writing the same thing
- `Foo<T>(T value = default(T))` - the default value operator works with type parameters
- `Foo(int? x = null)` - nullable conversion is valid
- `Foo(int x, int y = 10, params int[] z)` - parameter array can come after optional parameters

And some invalid ones:

- `Foo(int x = 0, int y)` - required non-params parameter cannot come after optional parameter
- `Foo(DateTime dt = DateTime.Now)` - default values have to be constant
- `Foo(XName name = "default")` - conversion from `string` to `XName` is user-defined
- `Foo(params string[] names = null)` - parameter arrays can't be optional
- `Foo(ref string name = "default")` - `ref/out` parameters can't have default values

The fact that the default value has to be constant is a pain in two different ways. One of them is familiar from a slightly different context, as we'll see now.

Versioning and optional parameters

The restrictions on default values for optional parameters may remind you of the restrictions on `const` fields, and in fact they behave very similarly. In both cases, when the compiler references the value it copies it directly into the output. The generated IL acts exactly as if your original source code had contained the default value. This means if you ever *change* the default value without recompiling everything that references it, the old callers will still be using the old default value. To make this concrete, imagine this set of steps:

1. Create a class library (`Library.dll`) with a class like this:

```
public class LibraryDemo
{
    public static void PrintValue(int value = 10)
    {
        System.Console.WriteLine(value);
    }
}
```

2. Create a console application (`Application.exe`) which references the class library:

```
public class Program
{
    static void Main()
    {
        LibraryDemo.PrintValue();
    }
}
```

3. Run the application - it will print 10, predictably.
4. Change the declaration of `PrintValue` as follows, then recompile *just* the class library:

```
public static void PrintValue(int value = 20)
```

5. Rerun the application - it will still print 10. The value has been compiled directly into the executable.
6. Recompile the application and rerun it - this time it will print 20.

This versioning issue can cause bugs which are very hard to track down, because all the code *looks* correct. Essentially, you are restricted to using genuine constants which should never change as default values for optional parameters. Of course, this also means you can't use any values which can't be expressed as constants anyway - you can't create a method with a default value of "the current time."

Making defaults more flexible with nullity

Fortunately, there is a way round this. Essentially you introduce a "magic value" to represent the default, and then replace that magic value with the *real* default within the method itself. If the phrase "magic value" bothers you, I'm not surprised - except we're going to use `null` for the magic value, which already represents the absence of a "normal" value. If the parameter type would normally be a value type, we simply make it the corresponding nullable value type, at which point we can still specify that the default value is `null`.

As an example of this, let's look at a similar situation to the one I used to introduce the whole topic: allowing the caller to supply an appropriate text encoding to a method, but defaulting to UTF-8. We can't specify the default encoding as `Encoding.UTF8` as that's not a constant value, but we can treat a null parameter value as "use the default". To demonstrate how we can handle value types, we'll make the method append a timestamp to a text file with a message. We'll default the encoding to UTF-8 and the timestamp to the current time. Listing 13.X shows the complete code, and a few examples of using it.

Example 13.2. Using null default values to handle non-constant situations

```
static void AppendTimestamp(string filename, ❶
                                string message,
                                Encoding encoding = null, ❷
                                DateTime? timestamp = null)
{
    Encoding realEncoding = encoding ?? Encoding.UTF8; ❸
    DateTime realTimestamp = timestamp ?? DateTime.Now;
    using (TextWriter writer = new StreamWriter(filename,
                                                true,
                                                realEncoding))
    {
        writer.WriteLine("{0:s}: {1}", realTimestamp, message);
    }
}
...
AppendTimestamp("utf8.txt", "First message");
AppendTimestamp("ascii.txt", "ASCII", Encoding.ASCII);
AppendTimestamp("utf8.txt", "Message in the future", null, ❹
                new DateTime(2030, 1, 1));
```

- ❶ Two required parameters
- ❷ Two optional parameters
- ❸ Null coalescing operator for convenience
- ❹ Explicit use of null

Listing 13.X shows a few nice features of this approach. First, we've solved the versioning problem. The default values for the optional parameters are null ❶, but the *effective* values are "the UTF-8 encoding" and "the current date and time." Neither of these could be expressed as constants, and should we ever wish to change the effective default - for example to use the current UTC time instead of the local time - we could do so without having to recompile everything that called `AppendTimestamp`. Of course, changing the effective default changes the behavior of the method - you need to take the same sort of care over this as you would with any other code change.

We've also introduced an extra level of flexibility. Not only do optional parameters mean we can make the calls shorter, but having a specific "use the default" value means that should we ever wish to, we can *explicitly* make a call allowing the method to choose the appropriate value. At the moment this is the only way we know to specify the timestamp explicitly without also providing an encoding ❶, but that will change when we look at named arguments.

The optional parameter values are very simple to deal with thanks to the null coalescing operator ❸. I've used separate variables for the sake of formatting, but you could use the same expressions directly in the calls to the `StreamWriter` constructor and the `WriteLine` method.

There's one downside to this approach: it assumes that you don't want to use null as a "real" value. There are certainly occasions where you want null to mean null - and if you don't want that to be the default

value, you'll have to find a different constant or just make leave the parameter as a required one. However, in other cases where there isn't an obvious constant value which will clearly *always* be the right default, I'd recommend this approach to optional parameters as one which is easy to follow consistently and removes some of the normal difficulties.

We'll need to look at how optional parameters affect overload resolution, but it makes sense to visit that topic just once, when we've seen how named arguments work. Speaking of which...

Named arguments

The basic idea of named arguments is that when you specify an argument value, you can also specify the name of the parameter it's supplying the value for. The compiler then makes sure that there *is* a parameter of the right name, and uses the value for that parameter. Even on its own, this can increase readability in some cases. In reality, named arguments are most useful in cases where optional parameters are also likely to appear, but we'll look at the simple situation first.

Indexers, optional parameters and named arguments

You *can* use optional parameters and named arguments with indexers as well as methods. However, this is only useful for indexers with more than one parameter: you can't access an indexer without specifying at least one argument anyway. Given this limitation, I don't expect to see the feature used very much with indexers, and I haven't demonstrated it in the book.

I'm sure we've all seen code which looks something like this:

```
MessageBox.Show("Please do not press this button again", // text  
               "Ouch!"); // title
```

I've actually chosen a pretty tame example: it can get a lot worse when there are loads of arguments, especially if a lot of them are the same type. However, this is still realistic: even with just two parameters, I would find myself guessing which argument meant what based on the text when reading this code, unless it had comments like the ones I've got here. There's a problem though: comments can lie very easily. Nothing is checking them at all. Named arguments ask the compiler to help.

Syntax

All we need to do to the previous example is prefix each argument with the name of the corresponding parameter and a colon:

```
MessageBox.Show(text: "Please do not press this button again",  
               caption: "Ouch!");
```

Admittedly we now don't get to choose the name we find most meaningful (I prefer "title" to "caption") but at least I'll know if I get something wrong. Of course, the most common way in which we could "get something wrong" here is to get the arguments the wrong way round. Without named arguments, this would be a problem: we'd end up with the pieces of text switched in the message box. With named arguments, the position becomes largely irrelevant. We can rewrite the previous code like this:

```
MessageBox.Show(caption: "Ouch!",  
               text: "Please do not press this button again");
```

We'd still have the right text in the right place, because the compiler would work out what we meant based on the names.

To explore the syntax in a bit more detail, listing 13.X shows a method with three integer parameters, just like the one we used to start looking at optional parameters.

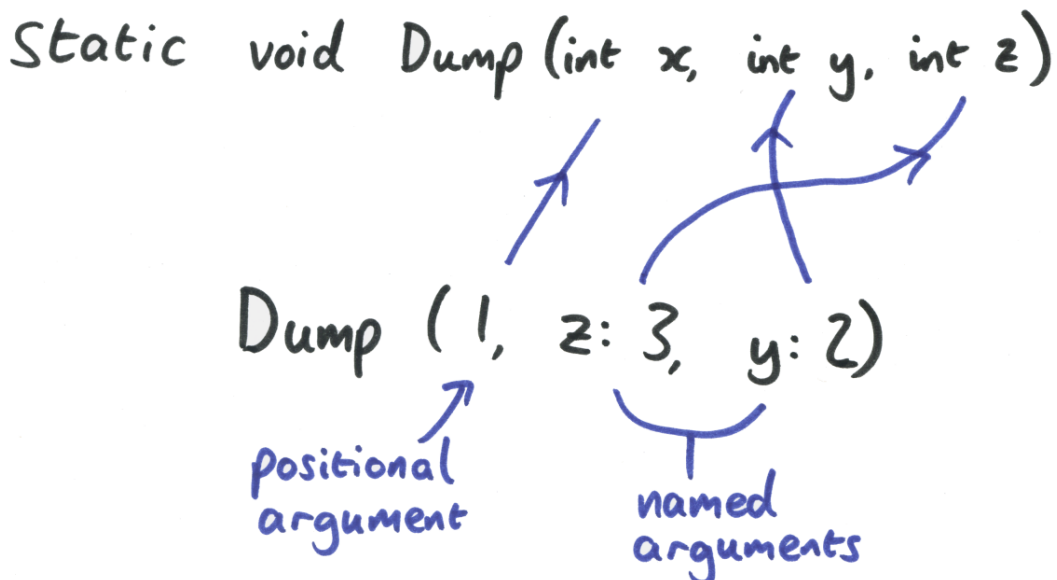
Example 13.3. Simple examples of using named arguments

```
static void Dump(int x, int y, int z) ❶
{
    Console.WriteLine("x={0} y={1} z={2}", x, y, z);
}
...
Dump(1, 2, 3); ❷
Dump(x: 1, y: 2, z: 3); ❸
Dump(z: 3, y: 2, x: 1);
Dump(1, y: 2, z: 3); ❹
Dump(1, z: 3, y: 2);
```

- ❶ Declares method as normal
- ❷ Calls method as normal
- ❸ Specifies names for all arguments
- ❹ Specifies names for some arguments

The output is the same for each call in listing 13.X: $x=1$, $y=2$, $z=3$. We've effectively made the same method call in five different ways. It's worth noting that there are no tricks in the method declaration ❶: you can use named arguments with any method which has at least one parameter. First we call the method in the normal way, without using any new features ❷. This is a sort of "control point" to make sure that the other calls really are equivalent. We then make two calls to the method using just named arguments ❸. The second of these calls reverses the order of the arguments, but the result is still the same, because the arguments are matched up with the parameters by name, not position. Finally there are two calls using a mixture of named arguments and *positional arguments* ❹. A positional argument is one which isn't named - so every argument in valid C# 3 code is a positional argument from the point of view of C# 4. Figure 13.X shows how the final line of code works.

Figure 13.2. Positional and named arguments in the same call



All named arguments have to come after positional arguments - you can't switch between the styles. Positional arguments *always* refer to the corresponding parameter in the method declaration - you can't make positional arguments "skip" a parameter by specifying it later with a named argument. This means that these method calls would both be invalid:

- `Dump(z: 3, 1, y: 2)` - positional arguments must come before named ones
- `Dump(2, x: 1, z: 3)` - `x` has already been specified by the first positional argument, so we can't specify it again with a named argument

Now, although in *this particular case* the method calls have been equivalent, that's not *always* the case. Let's take a look at why reordering arguments might change behaviour.

Argument evaluation order

We're used to C# evaluating its arguments in the order they're specified - which, until C# 4, has always been the order in which the parameters have been declared too. In C# 4, only the first part is still true: the arguments are still evaluated in order they're written, even if that's not the same as the order in which they're declared as parameters. This matters if evaluating the arguments has side effects. It's *usually* worth trying to avoid having side effects in arguments, but there are cases where it can make the code clearer. A more realistic rule is to try to avoid side effects which might interfere with each other. For the sake of demonstrating execution order, we'll break both of these rules. Please don't treat this as a recommendation that you do the same thing.

First we'll create a relatively harmless example, introducing a method which logs its input and returns it - a sort of "logging echo". We'll use the return values of three calls to this to call the `Dump` method (which isn't shown as it hasn't changed). Listing 13.X shows two calls to `Dump` which result in slightly different output.

Example 13.4. Logging argument evaluation

```
static int Log(int value)
{
    Console.WriteLine("Log: {0}", value);
    return value;
}
...
Dump(x: Log(1), y: Log(2), z: Log(3));
Dump(z: Log(3), x: Log(1), y: Log(2));
```

The results of running listing 13.X show what happens:

```
Log: 1
Log: 2
Log: 3
x=1 y=2 z=3
Log: 3
Log: 1
Log: 2
x=1 y=2 z=3
```

In both cases, the parameters in the `Dump` method are still 1, 2 and 3 in that order. However, we can see that while they were evaluated in that order in the first call (which was equivalent to just using positional arguments), the second call evaluated the value used for the `z` parameter first. We can make the effect even more significant by using side effects which change the results of the argument evaluation, as shown in listing 13.X, again using the same `Dump` method.

Example 13.5. Abusing argument evaluation order

```
int i = 0;
Dump(x: ++i, y: ++i, z: ++i);
i = 0;
Dump(z: ++i, x: ++i, y: ++i);
```

The results of listing 13.X may be best expressed in terms of the blood spatter pattern at a murder scene, after someone maintaining code like this has gone after the original author with an axe. Yes, *technically speaking* the last line prints out `x=2 y=3 z=1` but I'm sure you see what I'm getting at. Just say "no" to code like this. By all means reorder your arguments for the sake of readability: you may think that laying out a call to `MessageBox.Show` with the title coming above the text in the code itself reflects the on-screen layout more closely, for example. If you want to rely on a particular evaluation order for the arguments though, introduce some local variables to execute the relevant code in separate statements. The compiler won't care - it will follow the rules of the spec - but this reduces the risk of a "harmless refactoring" which inadvertently introduces a subtle bug.

To return to cheerier matters, let's combine the two features (optional parameters and named arguments) and see how much tidier the code can be.

Putting the two together

The two features work in tandem with no extra effort required on your part. It's not at all uncommon to have a bunch of parameters where there are obvious defaults, but where it's hard to predict which ones a caller will want to specify explicitly. Figure 13.X shows just about every combination: a required parameter, two optional parameters, a positional argument, a named argument and a "missing" argument for an optional parameter.

Figure 13.3. Mixing named arguments and optional parameters

Static void Dump (int x, int y=20, int z=30)

Dump (10, z: 3) 20

Going back to an earlier example in listing 13.X we wanted to append a timestamp to a file using the "default" encoding of UTF-8, but with a particular timestamp. Back then we just used `null` for the encoding argument, but now we can write the same code more simply, as shown in listing 13.X.

Example 13.6. Combining named and optional arguments

```
static void AppendTimestamp(string filename,
                           string message,
                           Encoding encoding = null,
                           DateTime? timestamp = null)
{
    ❶
}
...
AppendTimestamp("utf8.txt", "Message in the future", ❷
               timestamp: new DateTime(2030, 1, 1)); ❸
```

- ❶ Same implementation as before
- ❷ Encoding is omitted
- ❸ Named timestamp argument

In this fairly simple situation the benefit isn't particularly huge, but in cases where you want to omit three or four arguments but specify the final one, it's a real blessing.

We've seen how optional parameters reduce the need for huge long lists of overloads, but one specific pattern where this is worth mentioning is with respect to immutability.

Immutability and object initialization

One aspect of C# 4 which disappoints me somewhat is that it hasn't done much *explicitly* to make immutability easier. Immutable types are a core part of functional programming, and C# has been gradually supporting the functional style more and more... except for immutability. Object and collection initializers make it easy to work with *mutable* types, but immutable types have been left out in the cold. (Automatically implemented properties fall into this category too.) Fortunately, while it's not a feature which is particularly designed to aid immutability, named arguments and optional parameters allow you to write object-initializer-like code which just calls a constructor or other factory method. For instance, suppose we were creating a `Message` class, which required a "from" address, a "to" address and a body, with the subject and attachment being optional. (We'll stick with single recipients in order to keep the example as simple as possible.) We *could* create a mutable type with appropriate writable properties, and construct instances like this:

```
Message message = new Message {
    From = "skeet@pobox.com",
    To = "csharp-in-depth-readers@everywhere.com",
    Body = "I hope you like the second edition",
    Subject = "A quick message"
};
```

That has two problems: first, it doesn't enforce the required fields. We could force those to be supplied to the constructor, but then (before C# 4) it wouldn't be obvious which argument meant what:

```
Message message = new Message(
    "skeet@pobox.com",
    "csharp-in-depth-readers@everywhere.com",
    "I hope you like the second edition")
{
    Subject = "A quick message"
};
```

The second problem is that this construction pattern simply doesn't work for immutable types. The compiler has to call a property setter *after* it has initialized the object. However, we can use optional parameters and named arguments to come up with something that has the nice features of the first form (only specifying what you're interested in and supplying names) without losing the validation of which aspects of the message are required or the benefits of immutability. Listing 13.X shows a possible constructor signature and the construction step for the same message as before.

Example 13.7.

```
public Message(string from, string to,
               string body, string subject = null,
               byte[] attachment = null)
{
    ❶
}
...
Message message = new Message(
    from: "skeet@pobox.com",
    to: "csharp-in-depth-readers@everywhere.com",
    body: "I hope you like the second edition",
    subject: "A quick message"
);
```

❶ Normal initialization code goes here

I really like this in terms of readability and general cleanliness. You don't need hundreds of constructor overloads to choose from, just one with some of the parameters being optional. The same syntax will also work with static creation methods, unlike object initializers. The only downside is that it really relies on your code being consumed by a language which supports optional parameters and named arguments, otherwise callers will be forced to write ugly code to specify values for all the optional parameters. Obviously there's more to immutability than getting values to the initialization code, but this is a welcome step in the right direction nonetheless.

There are couple of final points to make around these features before we move on to COM, both around the details of how the compiler handles our code and the difficulty of good API design.

Overload resolution

Clearly both of these new features affect how the compiler resolves overloads - if there are multiple method signatures available with the same name, which should it pick? Optional parameters can *increase* the number of applicable methods (if some methods have more parameters than the number of specified arguments) and named arguments can *decrease* the number of applicable methods (by ruling out methods which don't have the appropriate parameter names).

For the most part, the changes are absolutely intuitive: to check whether any particular method is applicable, the compiler tries to build a list of the arguments it *would* pass in, using the positional arguments in order, then matching the named arguments up with the remaining parameters. If a required parameter hasn't been specified or if a named argument doesn't match any remaining parameters, the method isn't applicable. The specification gives a little more detail around this, but there are two situations I'd like to draw particular attention to.

First, if two methods are both applicable and one of them has been given *all* of its arguments explicitly while the other uses an optional parameter filled in with a default value, the method which doesn't use any default values will win. However, this *doesn't* extend to just comparing the number of default values used - it's a strict "does it use default values or not" divide. For example, consider the code below.


```
static void Foo(int x = 10) {}
static void Foo(int x = 10, int y = 20) {}
...
Foo(); ❶
Foo(1); ❷
Foo(y: 2); ❸
Foo(1, 2); ❹
```

- ❶❶ Ambiguous call
- ❷❷ Valid call - chooses first overload
- ❸❸ Argument name forces second overload
- ❹❹ Argument count forces second overload

In the first call ❶, both methods are applicable because of their default parameters. However, the compiler can't work out which one you meant to call: it will raise an error. In the second call ❷ both methods are still applicable, but the first overload is used because it can be applied without using any default values, whereas the second uses the default value for *y*. For both the third and fourth calls, only the second overload is applicable. The third call ❸ names the *y* argument, and the fourth call ❹ has two arguments; both of these mean the first overload isn't applicable.

The second point is that sometimes named arguments can be an alternative to casting in order to help the compiler resolve overloads. Sometimes a call can be ambiguous because the arguments can be converted to the parameter types in two different methods, but neither method is "better" than the other in all respects. For instance, consider the following method signatures and a call:

```
void Method(int x, object y) { ... }
void Method(object a, int b) { ... }
...
Method(10, 10); ❶
```

- ❶ Ambiguous call

Both methods are applicable, and neither is better than the other. There are two ways to resolve this, assuming you can't change the method names to make them unambiguous that way. (That's my preferred approach, by the way. Make each method name more informative and specific, and the general readability of the code can go up.) You can either cast one of the arguments explicitly, or use named arguments to resolve the ambiguity:

```
void Method(int x, object y) { ... }
void Method(object a, int b) { ... }
...
Method(10, (object) 10); ❶
Method(x: 10, y: 10); ❷
```

- ❶ Casting to resolve ambiguity
- ❷ Naming to resolve ambiguity

Of course this only works if the parameters have different names in the different methods - but it's a handy trick to know. Sometimes the cast will give more readable code, sometimes the name will. It's just an extra weapon in the fight for clear code. It does have a downside though, along with named arguments in general: it's another thing to be careful about when you change a method...

Contracts and overrides

In the past, parameter names haven't matter very much if you've only been using C#. Other languages may have cared, but in C# the only times that parameter names were important were when you were looking at

IntelliSense and when you were looking at the method code itself. Now, the parameter names of a method are effectively part of the API. If you change them at a later date, code can break - anything which was using a named argument to refer to one of your parameters will fail to compile if you decide to change it. This may not be much of an issue if your code is only consumed by itself anyway, but if you're writing a public API such as an Open Source class library, be aware that changing a parameter name is a big deal. It always has been really, but if everything calling the code was written in C#, we've been able to ignore that until now.

Renaming parameters is bad: switching the names round is worse. That way the calling code may still compile, but with a different meaning. A particularly evil form of this is to override a method and switch the parameter names in the overridden version. The compiler will always look at the "deepest" override it knows about, based on the static type of the expression used as the target of the method call. You really don't want to get into a situation where calling the same method implementation with the same argument list results in different behavior based on the static type of a variable. That's just evil.

Speaking of evil, let's move on to the new features relating to COM. I'm only kidding - mostly, anyway.

Improvements for COM interoperability

I'll readily admit to being far from a COM expert. When I tried to use it before .NET came along, I always ran into issues which were no doubt partially caused by my lack of knowledge and partially caused by the components I was working with being poorly designed or implemented. The overall impression of COM as a sort of "black magic" has lingered, however. I've been reliably informed that there's a lot to like about it, but unfortunately I haven't found myself going back to learn it in detail - and there seems to be a *lot* of detail to study.

This section is Microsoft-specific

The changes for COM interoperability won't make sense for all C# compilers, and a compiler can still be deemed compliant with the specification without implementing these features.

.NET has certainly made COM somewhat friendlier in general, but until now there have been distinct advantages to using it from Visual Basic instead of C#. The playing field has been leveled significantly by C# 4 though, as we'll see in this section. For the sake of familiarity, I'm going to use Word for the example in this chapter, and Excel in the next chapter. There's nothing Office-specific about the new features though; you should find the experience of working with COM to be nicer in C# 4 whatever you're doing.

The horrors of automating Word before C# 4

Our example is going to be very simple - it's just going to start Word, create a document with a single paragraph of text in, save it, and then exit. Sounds easy, right? If only that were so. Listing 13.X shows the code required before C# 4.

Example 13.8. Creating and saving a document in C# 3

```
object missing = Type.Missing;

Application app = new Application { Visible = true }; ❶

app.Documents.Add(ref missing, ref missing, ❷
                  ref missing, ref missing);
Document doc = app.ActiveDocument;
Paragraph para = doc.Paragraphs.Add(ref missing);
para.Range.Text = "Thank goodness for C# 4";

object filename = "demo.doc"; ❸
object format = WdSaveFormat.wdFormatDocument97;
doc.SaveAs(ref filename, ref format,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing);

doc.Close(ref missing, ref missing, ref missing); ❹
app.Quit(ref missing, ref missing, ref missing);
```

- ❶ Starts Word
- ❷ Creates a new document
- ❸ Saves the document
- ❹ Shuts down word

Each step in this code sounds simple: first we create an instance of the COM type ❶ and make it visible using an object initializer expression, then we create and fill in a new document ❷. The mechanism for inserting some text into a document isn't quite as straightforward as we might expect, but it's worth remembering that a Word document can have a fairly complex structure: this isn't as bad as it might be. A couple of the method calls here have optional by-reference parameters; we're not interested in them, so we pass a local variable by reference with a value of `Type.Missing`. If you've ever done any COM work before, you're probably very familiar with this pattern.

Next comes the really nasty bit: saving the document ❸. Yes, the `SaveAs` method really does have 16 parameters, of which we're only using two. Even those two need to be passed by reference, which means creating local variables for them. In terms of readability, this is a complete nightmare. Don't worry though - we'll soon sort it out.

Finally we close the document and the application ❹. Aside from the fact that both calls have three optional parameters which we don't care about, there's nothing interesting here.

Let's start off by using the features we've already seen in this chapter - they can cut the example down significantly on their own.

The revenge of default parameters and named arguments

First things first: let's get rid of all those arguments corresponding to optional parameters we're not interested in. That also means we don't need the `missing` variable. That still leaves us with two

parameters out of a possible 16 for the `SaveAs` method. At the moment it's obvious which is which based on the local variable names - but what if we've got them the wrong way round? All the parameters are weakly typed, so we're really going on a certain amount of guesswork. We can easily give the arguments names to clarify the call. If we wanted to use one of the later parameters we'd have to specify the name anyway, just to skip the ones we're not interested in.

Listing 13.X shows the code - it's looking a lot cleaner already.

Example 13.9. Automating Word using named arguments and without specifying unnecessary parameters

```
Application app = new Application { Visible = true };
app.Documents.Add();
Document doc = app.ActiveDocument;
Paragraph para = doc.Paragraphs.Add();
para.Range.Text = "Thank goodness for C# 4";

object filename = "demo.doc";
object format = WdSaveFormat.wdFormatDocument97;
doc.SaveAs(FileName: ref filename, FileFormat: ref format);

doc.Close();
app.Quit();
```

That's much better - although it's still ugly to have to create local variables for the `SaveAs` arguments we *are* specifying. Also, if you've been reading very carefully, you may be a little concerned about the optional parameters we've removed. They were `ref` parameters... but optional... which isn't a combination C# supports normally. What's going on?

When is a `ref` parameter not a `ref` parameter?

C# normally takes a pretty strict line on `ref` parameters. You have to mark the argument with `ref` as well, to show that you understand what's going on; that your variable may have its value changed by the method you're calling. That's all very well in normal code, but COM APIs often use `ref` parameters for pretty much *everything* for perceived performance reasons. They're usually not actually modifying the variable you pass in. Passing arguments by reference is slightly painful in C#. Not only do you have to specify the `ref` modifier, you've also got to have a variable; you can't just pass *values* by reference.

In C# 4 the compiler makes this a lot easier by letting you pass an argument by value into a COM method, even if it's for a `ref` parameter. Consider a call like this, where argument might happen to be a variable of type `string`, but the parameter is declared as `ref object`:

```
comObject.SomeMethod(argument);
```

The compiler emits code which is equivalent to this:

```
object tmp = argument;
comObject.SomeMethod(ref tmp);
```

Note that any changes made by `SomeMethod` are discarded, so the call really does behave as if you were passing argument by value. This same process is used for optional `ref` parameters: each involves a local variable initialized to `Type.Missing` and passed by reference into the COM method. If you decompile the slimlined C# code, you'll see that the IL emitted is actually pretty bulky with all of those extra variables.

We can now apply the finishing touches to our Word example, as shown in listing 13.X.

Example 13.10. Passing arguments by value in COM methods

```
Application app = new Application { Visible = true };
app.Documents.Add();
Document doc = app.ActiveDocument;
Paragraph para = doc.Paragraphs.Add();
para.Range.Text = "Thank goodness for C# 4";
doc.SaveAs(FileName: "test.doc", ❶
           FileFormat: WdSaveFormat.wdFormatDocument97);
doc.Close();
app.Quit();
```

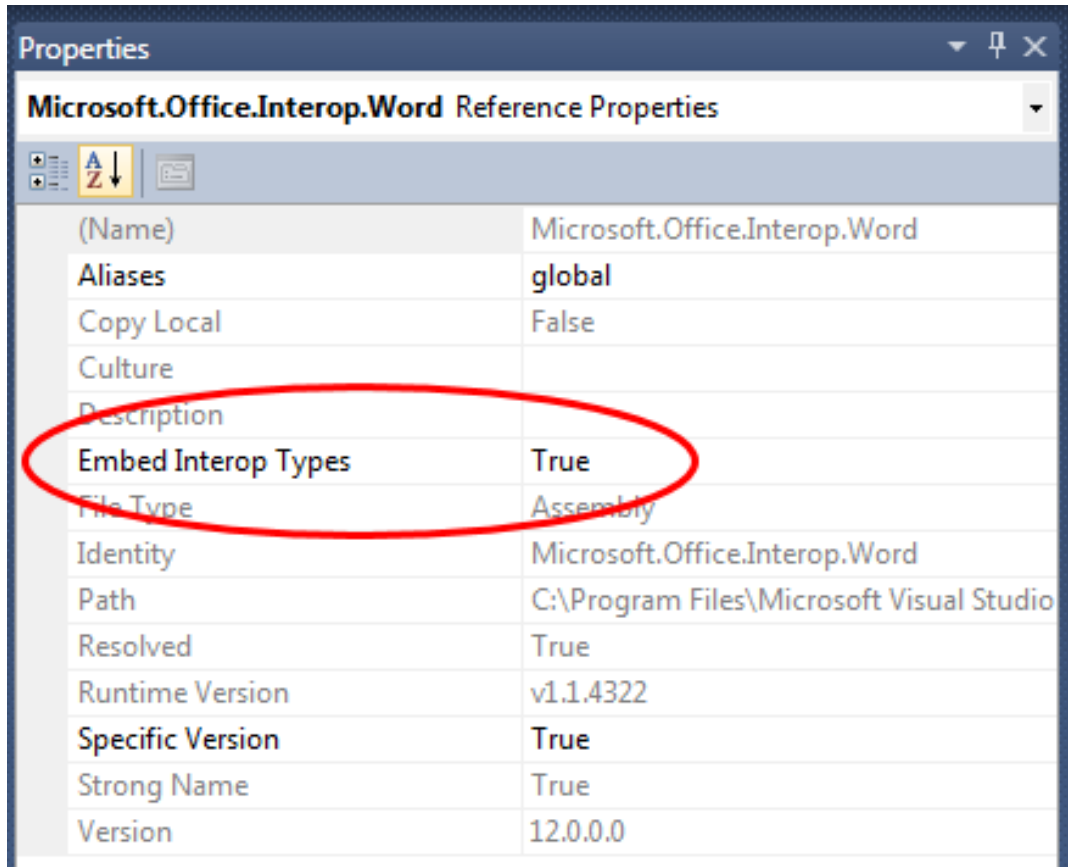
❶ Arguments passed by value

As you can see, the final result is a much cleaner bit of code than we started off with. With an API like Word you still need to work through a somewhat bewildering set of methods, properties and events in the core types such as `Application` and `Document`, but at least your code will be a lot easier to read. Of course, writing the code is only part of the battle: you usually need to be able to deploy it onto other machines as well. Again, C# 4 makes this task easier.

Linking Primary Interop Assemblies

When you build against a COM type, you use an assembly generated for the component library. Usually you use a *Primary Interop Assembly* or PIA, which is the canonical interop assembly for a COM library, signed by the publisher. You can generate these using the Type Library Importer tool (`tlbimp`) for your own COM libraries. PIAs make life easier in terms of having "one true way" of accessing the COM types, but they're a pain in other ways. For one thing, the right version of the PIA has to be present on the machine you're deploying your application to. It doesn't just have to be physically on the machine though - it also has to be registered (with the `RegAsm` tool). As an example of how this can be painful, depending on the environment your application will be deployed in, you may find that Office is installed but the relevant PIAs aren't, or that there's a different version of Office than the one you compiled against. You can redistribute the Office PIAs, but then you need to register them as part of your installation procedure - which means xcopy deployment isn't really an option.

C# 4 allows a very different approach. Instead of *referencing* a PIA like any other assembly, you can *link* it instead. In Visual Studio 2010 this is an option in the properties of the reference, as shown in figure 13.X.

Figure 13.4. Linking PIAs in Visual Studio 2010

For command line fans, you use the `/I` option instead of `/r` to link instead of reference:

```
csc /I:Path\To\PIA.dll MyCode.cs
```

When you link a PIA, the compiler embeds just the bits it needs from the PIA directly into your own assembly. It only takes the types it needs, and only the members within those types. For example, the compiler creates these types for the code we've written in this chapter:

```
namespace Microsoft.Office.Interop.Word
{
    [ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
    public interface _Application

    [ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
    public interface _Document

    [ComImport, CompilerGenerated, TypeIdentifier, Guid("...")]
    public interface Application : _Application

    [ComImport, Guid("..."), TypeIdentifier, CompilerGenerated]
    public interface Document : _Document

    [ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
    public interface Documents : IEnumerable
```

```
[TypeIdentifier("...", "WdSaveFormat"), CompilerGenerated]
public enum WdSaveFormat
{
}
```

And if you look in the `_Application` interface, it looks like this:

```
[ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
public interface _Application
{
    void _VtblGap1_4();
    Documents Documents { [...] get; }
    void _VtblGap2_1();
    Document ActiveDocument { [...] get; }
}
```

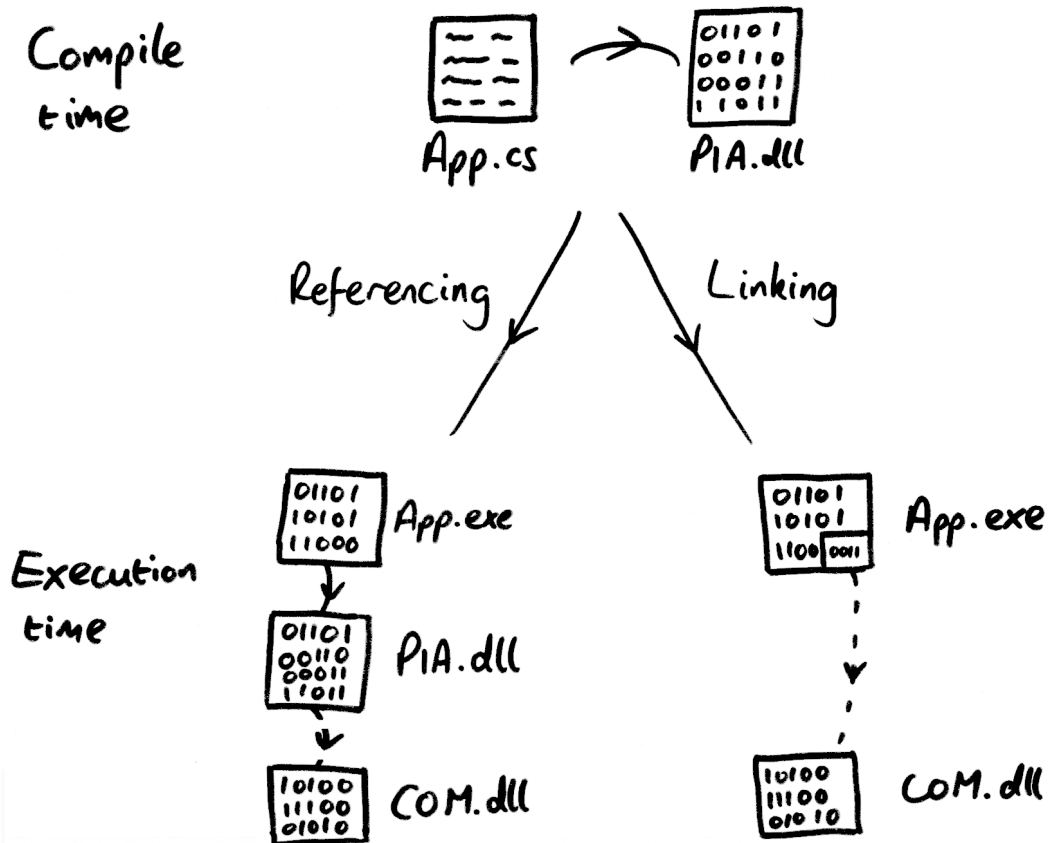
I've omitted the GUIDs and the property attributes here just for the sake of space, but you can always use Reflector to look at the embedded types. These are just interfaces and enums - there's no implementation. Whereas in a normal PIA there's a `CoClass` representing the actual implementation (but proxying everything to the real COM type of course) when the compiler needs to create an instance of a COM type via a linked PIA, it creates the instance using the GUID associated with the type. For example, the line in our Word demo which creates an instance of `Application` is translated into this code when linking is enabled⁴:

```
Application application = (Application) Activator.CreateInstance(
    Type.GetTypeFromCLSID(new Guid("...")));
```

Figure 13.X shows how this works at execution time.

⁴Well very nearly. The object initializer makes it slightly more complicated because the compiler uses an extra temporary variable.

Figure 13.5. Comparing referencing and linking



There are various benefits to embedding type libraries:

- Deployment is easier: the original PIA isn't needed, so there's nothing to install
- Versioning is simpler: so long as you only use members from the version of the COM library which is *actually* installed, it doesn't matter if you compile against an earlier or later PIA
- Memory usage may be reduced: if you only use a small fraction of the type library, there's no need to load a large PIA
- Variants are treated as dynamic types, reducing the amount of casting required

Don't worry about the last point for now - I need to explain dynamic typing before it'll make much sense. All will be revealed in the next chapter.

As you can see, Microsoft has really taken COM interoperability very seriously for C# 4, making the whole development process less painful. Of course the degree of pain has always been variable depending on the COM library you're developing against - some will benefit more than others from the new features.

Our next feature is entirely separate from COM and indeed named arguments and optional parameters, but again it just eases development a bit.

Generic variance for interfaces and delegates

You may remember that in chapter 3 I mentioned that the CLR had some support for variance in generic types, but that C# hadn't exposed that support yet. Well, that's changed with C# 4. C# has gained the syntax

required to declare that interfaces are variant, and the compiler now knows about the possible conversions for interfaces and delegates.

This isn't a life-changing feature - it's more a case of flattening some speed bumps you may have hit occasionally. It doesn't even remove all the bumps; there are various limitations, mostly in the name of keeping generics absolutely typesafe. However, it's still a nice feature to have up your sleeve.

Just in case you need a reminder of what variance is all about, let's start off with a recap of the two basic forms it comes in.

Types of variance: covariance and contravariance

In essence, variance is about being able to use an object of one type as if it were another, in a typesafe way.

Ultimately, it doesn't matter whether you remember the terminology I'm going to use in this section. It will be useful while you're reading the chapter, but you're unlikely to find yourself needing it in conversation. The concepts are far more important.

There are two types of variance: *covariance* and *contravariance*. They're essentially the same idea, but used in the context of values moving in different directions. We'll start with covariance, which is generally an easier concept to understand.

Covariance: values coming out of an API

Covariance is all about values being returned from an operation back to the caller. Let's imagine a very, very simple generic interface implementing the factory pattern. It has a single method, `CreateInstance`, which will return an instance of the appropriate type. Here's the code:

```
interface IFactory<T>
{
    T CreateInstance();
}
```

Now, `T` only occurs once in the interface (aside from in the name, of course). It's only used as the *return value* of a method. That means it makes sense to be able to treat a factory of a specific type as a factory of a more general type. To put it in real-world terms, you can think of a pizza factory as a food factory.

Some people find it easier to think in terms of "bigger" and "smaller" types. Covariance is about being able to use a bigger type instead of a smaller one, when that type is only ever being returned by the API.

Contravariance: values going into an API

Contravariance is the opposite way round. It's about values being passed *into* the API by the caller: the API is consuming the values instead of producing them. Let's imagine another simple interface - one which can pretty-print a particular document type to the console. Again, there's just one method, this time called `Print`:

```
interface IPrettyPrinter<T>
{
    void Print(T document);
}
```

This time `T` only occurs in the *input* positions in the interface, as a parameter. To put this into concrete terms again, if we had an implementation of `IPrettyPrinter<SourceCode>`, we should be able to use it as an `IPrettyPrinter<CSharpCode>`.

Going back to the "bigger" and "smaller" terminology, contravariance is about being able to use a smaller type instead of a bigger one when that type is ever being passed into the API.

Invariance: values going both ways

So if covariance applies when values only come *out* of an API, and contravariance applies when values only go *into* the API, what happens when a value goes both ways? In short: nothing. That type would be *invariant*. Here's an interface representing a type which can serialize and deserialize a data type.

```
interface IStorage<T>
{
    byte[] Serialize(T value);
    T Deserialize(byte[] data);
}
```

This time, if we have an instance for a particular type, we can't treat it as an implementation of the interface for either a bigger or a smaller type. If we tried to use it in a covariant way, we might pass in an object which it can't serialize - and if we tried to use it in a contravariant way, we might get an unexpected type out when we deserialized some bytes.

If it helps, you can think invariance as being like `ref` parameters: to pass a variable by reference, it has to be *exactly* the same type as the parameter itself, because the value goes into the method and effectively comes out again too.

Using variance in interfaces

C# 4 allows you to specify in the declaration of a generic interface or delegate that a type parameter can be used covariantly by using the `out` modifier, or contravariantly using the `in` modifier. Once the type has been declared, the relevant types of conversion are available implicitly. This works exactly the same way in both interfaces and delegates, but I'll show them separately just for clarity. Let's start with interfaces as they may be a little bit more familiar - and we've used them already to describe variance.

Variant conversions are reference conversions

Any conversion using variance or covariance is a *reference conversion*, which means that the same reference is returned after the conversion. It doesn't create a new object, it just treats the existing reference as if it matched the target type. This is the same as casting between reference types in a hierarchy: if you cast a `Stream` to `MemoryStream` (or use the implicit conversion the other way) there's still just one object.

The nature of these conversions introduces some limitations, as we'll see later, but it means they're very efficient, as well as making the behavior easier to understand in terms of object identity.

This time we'll use very familiar interfaces to demonstrate the ideas, with some simple user-defined types for the type arguments.

Expressing variance with "in" and "out"

There are two interfaces that demonstrate variance particularly effectively: `IEnumerable<T>` is covariant in `T`, and `IComparer<T>` is contravariant in `T`. Here are their new type declarations in .NET 4.0:

```
public interface IEnumerable<out T>
public interface IComparer<in T>
```

It's easy enough to remember - if a type parameter is only used for output, you can use *out*; if it's only used for input, you can use *in*. The compiler doesn't know whether or not you can remember which form is called covariance and which is called contravariance!

Unfortunately the framework doesn't contain very many inheritance hierarchies which would help us demonstrate variance particularly clearly, so I'll fall back to the standard object oriented example of shapes. The downloadable source code includes the definitions for `IShape`, `Circle` and `Square`, which are fairly obvious. The interface exposes properties for the bounding box of the shape and its area. I'm going to use two lists quite a lot in the following examples, so I'll show their construction code just for reference.

```
List<Circle> circles = new List<Circle> {  
    new Circle(new Point(0, 0), 15),  
    new Circle(new Point(10, 5), 20),  
};  
  
List<Square> squares = new List<Square> {  
    new Square(new Point(5, 10), 5),  
    new Square(new Point(-10, 0), 2)  
};
```

The only important point really concerns the types of the variables - they're declared as `List<Circle>` and `List<Square>` rather than `List<IShape>`. This can often be quite useful - if we were to access the list of circles elsewhere, we might want to get at circle-specific members without having to cast, for example. The actual values involved in the construction code are entirely irrelevant; I'll use the names `circles` and `squares` elsewhere to refer to the same lists, but without duplicating the code.⁵

Using interface covariance

To demonstrate covariance, we'll try to build up a list of shapes from a list of circles and a list of squares. Listing 13.X shows two different approaches, neither of which would have worked in C# 3.

Example 13.11. Building a list of general shapes from lists of circles and squares using variance

```
List<IShape> shapesByAdding = new List<IShape>(); ❶  
shapesByAdding.AddRange(circles);  
shapesByAdding.AddRange(squares);  
  
IEnumerable<IShape> shapeSequence = circles; ❷  
List<IShape> shapesByConcat = shapeSequence.Concat(squares).ToList();
```

❶ Adds lists directly

❷ Uses LINQ for concatenation

Effectively listing 13.X shows covariance in four places, each converting a sequence of circles or squares into a sequence of general shapes, as far as the type system is concerned. First we create a new `List<IShape>` and call `AddRange` to add the circle and square lists to it ❶. (We could have passed one of them into the constructor instead, then just called `AddRange` once.) The parameter for `List<T>.AddRange` is of type `IEnumerable<T>`, so in this case we're treating each list as an `IEnumerable<IShape>` - something which wouldn't have been possible before. `AddRange` *could* have been written as a generic method with its own type parameter, but it wasn't - and in fact doing this would have made some optimisations hard or impossible.

⁵ In the full source code solution these are exposed as properties on the static `Shapes` class, but in the snippets version I've included the construction code where it's needed, so you can tweak it easily if you want to.

The other way of creating a list which contains the data in two existing sequences is to use LINQ ❶. We can't directly call `circles.Concat(squares)` - we need to convert circles to an `IEnumerable<IShape>` first, so that the relevant `Concat(IEnumerable<IShape>)` overload is available. However, this covariant conversion from `List<Circle>` to `IEnumerable<IShape>` isn't actually changing the value - just how the compiler *treats* the value. It isn't building a new sequence, which is the important point. We then use covariance again in the call to `Concat`, this time treating the list of squares as an `IEnumerable<IShape>`. Covariance is particularly important in LINQ to Objects, as so much of the API is expressed in terms of `IEnumerable<T>`.

In C# 3 there would certainly have been other ways to approach the same problem. We could have built `List<IShape>` instances instead of `List<Circle>` and `List<Square>` for the original shapes; we could have used the LINQ `Cast` operator to convert the specific lists to more general ones; we could have written our own list class with a generic `AddRange` method. None of these would have been as convenient or as efficient as the alternatives offered here, however.

Using interface contravariance

We'll use the same types to demonstrate contravariance. This time we'll only use the list of circles, but a comparer which is able to compare *any* two shapes by just comparing the areas. We happen to want to sort a list of circles, but that poses no problems now, as shown in listing 13.X.

Example 13.12. Sorting circles using a general-purpose comparer and contravariance

```
class AreaComparer : IComparer<IShape> ❶
{
    public int Compare(IShape x, IShape y)
    {
        return x.Area.CompareTo(y.Area);
    }
}
...
IComparer<IShape> areaComparer = new AreaComparer();
circles.Sort(areaComparer); ❷
```

❶❶ Compares shapes by area

❷❶ Sorts using contravariance

There's nothing complicated here. Our `AreaComparer` class ❶ is about as simple as an implementation of `IComparer<T>` can be; it doesn't need any state, for example. In a production environment you would probably want to introduce a static property to access an instance, rather than making users call the constructor. You'd also normally implement some null handling in the `Compare` method, but that's not necessary for our example.

Once we have an `IComparer<IShape>`, we're using it to sort a list of circles ❶. The argument to `circles.Sort` needs to be an `IComparer<Circle>`, but covariance allows us to convert our comparer implicitly. It's as simple as that.

Surprise, surprise

If someone had presented you with this code as if it were C# 3, you might have looked at it and expected it to work. It seems obvious that it *should* be able to work, and this is a common feeling; the invariance in C# 2 and 3 often is an unwelcome surprise. The new abilities of C# 4 in this area aren't introducing new concepts you'd never have thought of before, they'll just allow you more flexibility.

These have both been very simple examples using single-method interfaces, but the same principles apply for more complex APIs. Of course, the more complex the interface is, the more likely it is that a type parameter will be used for both input and output, which would make it invariant. We'll come back to some tricky examples later, but first we'll look at delegates.

Using variance in delegates

Now we've seen how to use variance with interfaces, applying the same knowledge to delegates is easy. We'll use some very familiar types again:

```
delegate T Func<out T>()  
delegate void Action<in T>(T obj)
```

These are really equivalent to the `IFactory<T>` and `IPrettyPrinter<T>` interfaces we started off with. Using lambda expressions, we can demonstrate both of these very easily, and even chain the two together. Listing 13.X shows an example using our shape types.

Example 13.13. Using variance with simple `Func<T>` and `Action<T>` delegates

```
Func<Square> squareFactory = () => new Square(new Point(5, 5), 10);  
Func<IShape> shapeFactory = squareFactory; ❶
```

```
Action<IShape> shapePrinter = shape => Console.WriteLine(shape.Area);  
Action<Square> squarePrinter = shapePrinter; ❷
```

```
squarePrinter(squareFactory()); ❸  
shapePrinter(shapeFactory());
```

- ❶ Converts `Func<T>` using covariance
- ❷ Converts `Action<T>` using contravariance
- ❸ Sanity checking...

Hopefully by now the code will need little explanation. Our "square factory" always produces a square at the same position, with sides of length 10. Covariance allows us to treat a square factory as a general shape factory ❶ with no fuss. We then create a general-purpose action which just prints out the area of whatever shape is given to it. This time we use a contravariant conversion to treat the action as one which can be applied to any square ❷. Finally, we feed the square action with the result of calling the square factory, and the shape action with the result of calling the shape factory. Both print 100, as we'd expect.

Of course we've only used delegates with a single type parameter here. What happens if we use delegates or interfaces with multiple type parameters? What about type arguments which are themselves generic delegate types? Well, it can all get quite complicated...

Complex situations

Before I try to make your head spin, I should provide a little comfort. Although we'll be doing some weird and wonderful things, the compiler will stop you from making mistakes. You may still get confused by the error messages if you've got several type parameters used in funky ways, but once you've got it compiling you should be safe⁶. Complexity is possible in both the delegate and interface forms of variance, although the delegate version is usually more concise to work with. Let's start off with a relatively simple example.

⁶Assuming the bug around `Delegate.Combine` [<http://stackoverflow.com/questions/1120688>] is fixed, of course. This footnote is a warning to MEAP readers for 4.0 beta 1, as well as a reminder for me to check it out later on and revise the text appropriately.

Simultaneous covariance and contravariance with `Converter<TInput, TOutput>`

The `Converter<TInput, TOutput>` delegate has been around since .NET 2.0. It's effectively `Func<T, TResult>` but with a clearer expected purpose. Listing 13.X shows a few combinations of variance using a simple converter.

Example 13.14. Demonstrating covariance and contravariance with a single type

```
Converter<object, string> converter = x => x.ToString(); ❶
Converter<string, string> contravariance = converter;
Converter<object, object> covariance = converter;
Converter<string, object> both = converter; ❷
```

❶❶ Converts objects to strings

❷❶ Converts strings to objects

Listing 13.X shows the variance conversions available on a delegate of type `Converter<object, string>`: a delegate which takes any object and produces a string. First we implement the delegate using a simple lambda expression which calls `ToString` ❶. As it happens, we never actually *call* the delegate, so we could have just used a null reference, but I think it's easier to think about variance if you can pin down a concrete action which *would* happen if you called it.

The next two lines are relatively straightforward, so long as you only concentrate on one type parameter at a time. The `TInput` type parameter is only used in an input position, so it makes sense that you can use it contravariantly, using a `Converter<object, string>` as a `Converter<string, string>`. In other words, if you can pass *any* object reference into the converter, you can certainly hand it a string reference. Likewise the `TOutput` type parameter is only used in an output position (the return type) so it makes sense to use that covariantly: if the converter always returns a string reference, you can safely use it where you only need to guarantee that it will return an object reference.

The final line ❶ is just a logical extension of this idea. It uses both contravariance and covariance in the same conversion, to end up with a converter which only accepts strings and only declares that it will return an object reference. Note that you *can't* convert this back to the original conversion type without a cast - we've essentially relaxed the guarantees at every point, and you can't tighten them up again implicitly.

Let's up the ante a little, and see just how complex things can get if you try hard enough.

Higher order function insanity

The really weird stuff starts happening when you combine variant types together. I'm not going to go into a lot of detail here - I just want you to appreciate the potential for complexity. Let's look at four delegate declarations:

```
delegate Func<T> FuncFunc<out T>();
delegate void ActionAction<out T>(Action<T> action);
delegate void ActionFunc<in T>(Func<T> function);
delegate Action<T> FuncAction<in T>();
```

Each of these declarations is equivalent to "nesting" one of the standard delegates inside another. For example, `FuncAction<T>` is equivalent to `Func<Action<T>>`. Both represent a function which will return an `Action` which can be passed a `T`. But should this be covariant or contravariant? Well, the function is going to *return* something to do with `T`, so it sounds covariant - but that "something" then

takes a T so it sounds contravariant. The answer is that the delegate *is* contravariant in T, which is why it's declared with the `in` modifier.

As a quick rule of thumb, you can think of nested contravariance as reversing the previous variance, whereas covariance doesn't - so while `Action<Action<T>>` is covariant in T, `Action<Action<Action<T>>>` is contravariant. Compare that with `Func<T>` variance, where you can write `Func<Func<Func<...Func<T>...>>>` with as many levels of nesting as you like and still get covariance.

Just to give a similar example using interfaces, let's imagine we have something that can compare sequences. If it can compare two sequences of arbitrary objects, it can certainly compare two sequences of strings - but not vice versa. Converting this to code (without implementing the interface!) we can see this as:

```
IComparer<IEnumerable<object>> objectsComparer = ...;  
IComparer<IEnumerable<string>> stringsComparer = objectsComparer;
```

This conversion is legal: `IEnumerable<string>` is a "smaller" type than `IEnumerable<object>` due to the covariance of `IEnumerable<T>`; the contravariance of `IComparer<T>` then allows the conversion from a comparer of "bigger" type to a comparer of a "smaller" type.

Of course we've only used delegates and interfaces with a single type parameter in this section - it can all apply to multiple type parameters too. Don't worry though: you're unlikely to need this sort of brain-busting variance very often, and when you do you've got the compiler to help you. I really just wanted to make you aware of the possibilities.

On the flip side, there are some things you may expect to be able to do, but which aren't supported.

Limitations and notes

The variance support provided by C# 4 is mostly limited by what's provided by the CLR. It would be hard for the language to support conversions which were prohibited by the underlying platform. This can lead to a few surprises.

No variance for type parameters in classes

Only interfaces and delegates can have variant type parameters. Even if you have a class which only uses the type parameter for input (or only uses it for output) you cannot specify the `in` or `out` modifiers. For example `Comparer<T>`, the common implementation of `IComparer<T>`, is invariant - there's no conversion from `Comparer<IShape>` to `Comparer<Circle>`.

Aside from any implementation difficulties which this might have incurred, I'd say it makes a certain amount of sense conceptually. Interfaces represent a way of looking at an object from a particular perspective, whereas classes are more rooted in the object's *actual* type. This argument is weakened somewhat by inheritance letting you treat an object as an instance of any of the classes in its inheritance hierarchy, admittedly. Either way, the CLR doesn't allow it.

Variance only supports reference conversions

You can't use variance between two arbitrary type arguments just because there's a conversion between them. It has to be a *reference conversion*. Basically that limits it to conversions which operate on reference types and which don't affect the binary representation of the reference. This is so that the CLR can know that operations will be type safe without having to inject any actual conversion code anywhere. As I mentioned in section 13.3.2, variant conversions are themselves reference conversions, so there wouldn't be anywhere for the extra code to go anyway.

In particular, this restriction prohibits any conversions of value types and user-defined conversions. For example, the following conversions are all invalid:

- `IEnumerable<int>` to `IEnumerable<object>` - boxing conversion
- `IEnumerable<short>` to `IEnumerable<int>` - value type conversion
- `IEnumerable<XmlAttribute>` to `IEnumerable<string>` - user-defined conversion

User-defined conversions aren't likely to be a problem as they're relatively rare, but you may find the restriction around value types a pain.

"out" parameters aren't output positions

This one came as a surprise to me, although it makes sense in retrospect. Consider a delegate with the following definition:

```
delegate bool TryParser<T>(string input, out T value)
```

You might expect that you could make `T` covariant - after all, it's only used in an output position... or is it? The CLR doesn't really know about out parameters. As far as it's concerned, they're just `ref` parameters with an `[Out]` attribute applied to them. `C#` attaches special meaning to the attribute in terms of definite assignment, but the CLR doesn't. Now `ref` parameters mean data going both ways, so if you have a `ref` parameter of type `T`, that means `T` is invariant.

Delegates and interfaces using out parameters are quite rare, so this may well never affect you anyway, but it's worth knowing about just in case.

Variance has to be explicit

When I introduced the syntax for expressing variance - applying the `in` or `out` modifiers to type parameters - you may have wondered why we needed to bother at all. The compiler is able to *check* that whatever variance you try to apply is valid - so why doesn't it just apply it automatically?

It could do that, but I'm glad it doesn't. Normally you can add methods to an interface and only affect implementations rather than callers. However, if you've declared that a type parameter is variant and you then want to add a method which breaks that variance, all the *callers* are affected too. I can see this causing a lot of confusion. Variance requires some thought about what you might want to do in the future, and forcing developers to explicitly include the modifier encourages them to plan carefully before committing to variance.

There's less of an argument for this explicit nature when it comes to delegates: any change to the signature that would affect the variance would probably break existing uses anyway. However, there's a lot to be said for consistency - it would feel quite odd if you had to specify the variance in interfaces but not in delegate declarations.

Beware of breaking changes

Whenever new conversions become available there's the risk of your current code breaking. For instance, if you rely on the results of the `is` or `as` operators *not* allowing for variance, your code will behave differently when running under .NET 4.0. Likewise there are cases where overload resolution will choose a different method due to there being more applicable options now. This is another reason for variance to be explicitly specified: it reduces the risk of breaking your code.

These situations should be quite rare, however, and the benefit from variance is more significant than the potential drawbacks. You *do* have unit tests to catch subtle changes, right? In all seriousness, the `C#` team

takes code breakage very seriously, but sometimes there's no way of introducing a new feature without breaking code.⁷

No caller-specified or partial variance

This is really a matter of interest and comparison rather than anything else, but it's worth noting that C#'s variance is *very* different to Java's system. Java's generic variance manages to be extremely flexible by approaching it from the other side: instead of the type itself declaring the variance, code *using* the type can express the variance it needs.

Want to know more?

This book isn't about Java generics, but if this little teaser has piqued your interest, you may want to check out Angelika Langer's Java Generics FAQ [<http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>]. Be warned: it's a huge and complex topic!

For example, the `List<T>` interface in Java is roughly equivalent to `ICollection<T>` in C#. It contains methods to both add items and fetch them, so clearly in C# it's invariant - but in Java you decorate the type at the calling code to explain what variance you want. The compiler then stops you from using the members which go against that variance. For example, the following code would be perfectly valid:

```
List<Shape> shapes1 = new ArrayList<Shape>();  
List<? super Square> squares = shapes1; ❶  
squares.add(new Square(10, 10, 20, 20));
```

```
List<Circle> circles = new ArrayList<Circle>();  
circles.add(new Circle(10, 10, 20));  
List<? extends Shape> shapes2 = circles; ❷  
Shape shape = shapes2.get(0);
```

- ❶ Declaration using contravariance
- ❷ Declaration using covariance

For the most part, I prefer generics in C# to Java, and type erasure in particular can be a pain in many cases. However, I find this treatment of variance really interesting. I don't expect to see anything similar in future versions of C# - so think carefully about how you can split your interfaces to allow for flexibility, but without introducing more complexity than is really warranted.

Summary

This has been a bit of a "pick and mix" chapter, with three distinct areas. Having said that, COM greatly benefits from named arguments and optional parameters, so there's some overlap between them.

I suspect it will take a while for C# developers to get the hang of how best to use the new features for parameters and arguments. Overloading still provides extra portability for languages which don't support optional parameters, and named arguments may look strange in some situations until you get used to them. The benefits can be significant though, as I demonstrated with the example of building instances of immutable types. You'll need to take some care when assigning default values to optional parameters, but I hope that you'll find the suggestion of using null as a "default default value" to be a useful and flexible one which effectively side-steps some of the limitations and pitfalls you might otherwise encounter.

⁷In .NET 4.0b1 there's no warning given for behavioral changes, as there was when method group conversion variance was introduced in C# 2. I'm hoping this will change before VS2010 ships.

Working with COM has come on a *long* way for C# 4. I still prefer to use purely managed solutions where they're available, but at least the code calling into COM is a lot more readable now, as well as having a better deployment story. We're not quite finished with the COM story, as the dynamic typing features we'll see in the next chapter impact on COM too, but even without taking that into account we've seen a short sample become a lot more pleasant just by applying a few simple steps.

Finally we examined generic variance. Sometimes you may end up using variance without even knowing it, and I think most developers are more likely to use the variance declared in the framework interfaces and delegates rather than creating their own ones. I apologise if it occasionally became a bit tricky - but it's good to know just what's out there. If it's any consolation to you, C# team member Eric Lippert has publicly acknowledged [<http://blogs.msdn.com/ericlippert/archive/2007/10/24/covariance-and-contravariance-in-c-part-five-higher-order-functions-hurt-my-brain.aspx>] that higher order functions make even *his* head hurt, so we're in good company. Eric's post is one in a long series [<http://blogs.msdn.com/ericlippert/archive/tags/Covariance+and+Contravariance/default.aspx>] about variance, which is as much as anything a dialogue about the design decisions involved. If you haven't had enough of variance by now, it's an excellent read.

This chapter dealt with *relatively* small changes to C#. Chapter 14 deals with something far more fundamental: the ability to use C# in a dynamic manner.

Chapter 14. Dynamic binding in a static language

C# has always been a statically typed language, with no exceptions. There have been a few areas where the compiler has looked for particular names rather than interfaces, such as finding appropriate `Add` methods for collection initializers, but there's been nothing truly dynamic in the language beyond normal polymorphism. That changes with C# 4 - at least partially. The simplest way of explaining it is that there's a new static type called `dynamic`, which you can try to do pretty much anything with at compile time, and let the framework sort it out at execution time. Of course there's rather more to it than that, but that's the executive summary.

Given that C# is still a statically typed language everywhere that you're *not* using `dynamic`, I don't expect fans of dynamic programming to suddenly become C# advocates. That's not the point of the feature: it's all about interoperability. As dynamic languages such as IronRuby and IronPython join the .NET ecosystem, it would be crazy not to be able to call into C# code from IronPython and vice versa. Likewise developing against weakly-typed COM APIs has always been awkward in C#, with an abundance of casts cluttering the code. We've already seen some improvements in C# 4 when it comes to working with COM, and dynamic typing is the final new feature of C# 4.

One word of warning though - and I'll be repeating this throughout the chapter - it's worth being careful with dynamic typing. It's certainly fun to explore, and it's been very well implemented, but I still recommend that you stay away from it in production code unless there's a *clear* benefit to using it. Dynamic code will be slower than static code (even though the framework does a very good job of optimising it as far as it can) but more importantly, you lose a lot of compile-time safety. While unit testing will help you find a lot of the mistakes that can crop up when the compiler isn't able to help you much, I still prefer the immediate feedback of the compiler telling me if I'm trying to use a method which doesn't exist or can't be called with a given set of arguments.

Dynamic languages certainly have their place, but if you're really looking to write large chunks of your code dynamically, I suggest you use a language where that's the *normal* style instead of the exception. Now that you can easily call into dynamic languages from C#, you can fairly easily separate out the parts of your code which benefit from a largely dynamic style from those where static typing works better.

I don't want to put too much of a damper on things though: where dynamic typing *is* useful, it can be a lot simpler than the alternatives. In this chapter we'll take a look at the basic rules of dynamic typing in C# 4, and then dive into some examples: using COM dynamically, calling into some IronPython code, and making reflection a lot simpler. You can do all of this without knowing details, but after we've got the flavor of dynamic typing, we'll look at what's going on under the hood. In particular, we'll discuss the Dynamic Language Runtime and what the C# compiler does when it encounters dynamic code. Finally, we'll see how you can make your own types respond dynamically to methods calls, property accesses and the like. First though, let's take a step back.

What? When? Why? How?

Before we get to any code showing off this new feature of C# 4, it's worth getting a better handle on why it was introduced in the first place. I don't know any other languages which have gone from being purely static to partially dynamic; this is a significant step in C#'s evolution, whether you make use of it often or only occasionally.

We'll start off by taking a fresh look at what "dynamic" and "static" mean, consider some of the major use cases for dynamic typing in C#, and lead into how it's implemented in C# 4.

What is dynamic typing?

In chapter 2, I discussed the characteristics of a type system and described how C# was a statically typed language in versions 1-3. The compiler knows the type of expressions in the code, and knows the members available on any type. It applies a fairly complex set of rules to determine which exact member should be used. This includes overload resolution; the only choice which is left until later is to pick the implementation of virtual methods depending on the execution time type of the object. The process of working out which member to use is called *binding*, and in a statically typed language it occurs at compile time.

In a dynamically typed language, all of this binding occurs at execution time. A compiler is able to check that the code is *syntactically* correct, but it can't check that the methods you call and the properties you access are actually present. It's a bit like a word processor with no dictionary: it may be able to check your punctuation, but not your spelling. (If you're to have any sort of confidence in your code, you really need a good set of unit tests.) Some dynamic languages are interpreted to start with, with no compiler involved at all. Others provide an interpreter as well as a compiler, to allow rapid development with a *REPL*: a read-evaluate-print loop.¹

It's worth noting that the new dynamic features of C# 4 do *not* include interpreting C# source code at execution time: there's no direct equivalent of the JavaScript `eval` function, for example. To execute code based on data in strings, you need to use either the CodeDOM API (and `CSharpCodeProvider` in particular) or simple reflection to invoke individual members.

Of course, the same kind of work has to be done at *some* point in time no matter what approach you're taking. By asking the compiler to do more work before execution, static systems usually perform better than dynamic ones. Given the downsides we've mentioned so far, you might be wondering why anyone would want to bother with dynamic typing in the first place.

When is dynamic typing useful, and why?

Dynamic typing has two important points in its favor. First, if you know the name of a member you want to call, the arguments you want to call it with, and the object you want to call it on, that's all you need. That may sound like all the information you could have anyway, but there's more that the C# compiler would normally want to know. Crucially, in order to identify the member exactly (modulo overriding) it would need to know the type of the object you're calling it on, and the types of the arguments. Sometimes you just don't know those types at compile-time, even though you *do* know enough to be sure that the member will be present and correct when the code actually runs.

For example, if you know that the object you're using has a `Length` property you want to use, it doesn't matter whether it's a `String`, a `StringBuilder`, an `Array`, a `Stream`, or any of the other types with that property. You don't need that property to be defined by some common base class or interface - which can be useful if there isn't such a type. This is called *duck typing*, from the notion that "if it walks like a duck and quacks like a duck, I would call it a duck."² Even when there *is* a type which offers everything you need, it can sometimes be an irritation to tell the compiler exactly which type you're talking about. This is particularly relevant when using Microsoft Office APIs via COM. Many method and properties are declared to just return `VARIANT`, which means that C# code using these calls is often peppered with casts. Duck typing allows you to omit all of these casts, so long as you're confident about what you're doing.

The second important feature of dynamic typing is the ability of objects and types to respond to a call by analysing the name and arguments provided to it. It can behave as if the member had been declared by

¹Strictly speaking, REPL isn't solely associated with dynamic languages. Some statically typed languages have "interpreters" too which actually compile on the fly. Notably, F# comes with a tool called F# Interactive which does exactly this. However, interpreters are much more common for dynamic languages than static ones.

²The Wikipedia article on duck typing [http://en.wikipedia.org/wiki/Duck_typing] has more information about the history of the term.

the type in the normal way, even if the member names couldn't possibly be known until execution time. For example, consider the following call:

```
books.FindByAuthor("Joshua Bloch")
```

Normally this would require the `FindByAuthor` member to be declared by the designer of the type involved. In a dynamic data layer there can be a single smart piece of code which works out that when make a call like that and there's an `Author` property in the associated data (whether that's from a database, XML document, hard-coded data or anything else) then you probably want to do a query using the specified argument as the author to find. In some ways this is just a more complex way of writing something like:

```
books.Find("Author", "Joshua Bloch")
```

However, the first snippet feels more appropriate: the calling code knows the "Author" part statically, even if the receiving code doesn't. This approach can be used to mimic domain specific languages (DSLs) in some situations. It can also be used to create a natural API for exploring data structures such as XML trees.

Another feature of programming with dynamic languages *tends* to be an experimental style of programming using an appropriate interpreter, as I mentioned earlier. This isn't *directly* relevant to C# 4, but the fact that C# 4 can interoperate richly with dynamic languages running on the *DLR* (Dynamic Language Runtime) means that if you're dealing with a problem which would benefit from this style, you'll be able to use the results directly from C# instead of having to port it to C# afterwards.

We'll look at these scenarios in more depth when we've learned the basics of C# 4's dynamic abilities, so we can see more concrete examples. It's worth briefly point out that if these benefits *don't* apply to you, dynamic typing is more likely to be a hindrance than a help. Many developers won't need to use dynamic typing very much in their day-to-day coding, and even when it *is* required it may well only be for a small part of the code. Just like any feature, it can be overused; in my view it's usually worth thinking carefully about whether any alternative designs would allow static typing to solve the same problem elegantly. However, I'm biased due to having a background in statically typed languages - it's worth reading books on dynamically typed languages such as Python and Ruby to see a wider variety of benefits than the ones I present in this chapter.

You're probably getting anxious to see some real code by now, so we'll just take a moment to get a very brief overview of what's going on, and then dive into some examples.

How does C# 4 provide dynamic typing?

C# 4 introduces a new type called `dynamic`. The compiler treats this type differently to any normal CLR type³. Any expression that uses a dynamic value causes the compiler to change its behavior in a radical way. Instead of trying to work out *exactly* what the code means, binding each member access appropriately, performing overload resolution and so on, the compiler just parses the source code to work out what *kind* of operation you're trying to perform, its name, what arguments are involved and any other relevant information. Instead of emitting IL to execute the code directly, the compiler generates code which calls into the Dynamic Language Runtime with all the required information. The rest of the work is then performed at execution time.

In many ways this is similar to the differences between the code generated when converting a lambda expression to an expression tree instead a delegate type. We'll see later that expression trees are extremely important in the DLR, and in many cases the C# compiler will use expression trees to describe the code. (In the simplest cases where there's nothing but a member invocation, there's no need for an expression tree.)

³In fact, `dynamic` doesn't represent a specific CLR type. It's really just `System.Object` in conjunction with `System.Dynamic.DynamicAttribute`. We'll look at this in more detail in section 14.4, but for the moment you can probably pretend it's a real type.

When the DLR comes to bind the relevant call at execution time, it goes through a complicated process to determine what should happen. This not only has to take in the normal C# rules for method overloads and so on, but also the possibility that the object itself will want to be part of the decision, as we saw in our `FindByAuthor` example earlier.

Most of this happens under the hood though - the source code you write to use dynamic typing can be really simple.

The 5 minute guide to `dynamic`

Do you remember how many new bits of syntax were involved when you learned about LINQ? Well dynamic typing is just the opposite: there's a single contextual keyword, `dynamic`, which you can use in most places that you'd use a type name. That's all the new syntax that's required, and the main rules about `dynamic` are easily expressed, if you don't mind a little bit of hand-waving to start with:

- An implicit conversion exists from any CLR type to `dynamic`
- An implicit conversion exists from `dynamic` to any CLR type
- Any expression which uses a value of type `dynamic` is evaluated dynamically
- The static type of any dynamically-evaluated expression is deemed to be `dynamic` (with the exception of explicit conversions and constructor calls - in both those cases the compiler knows the type of the result, even if it doesn't know exactly how it's going to get there)

The detailed rules are more complicated, as we'll see in section 14.4, but for the moment let's stick with the simplified version above. Listing 14.1 provides a complete example demonstrating each point.

Example 14.1. Using `dynamic` to iterate through a list, concatenating strings

```
dynamic items = new List<string> { "First", "Second", "Third" };
dynamic valueToAdd = " (suffix)";
foreach (dynamic item in items)
{
    string result = item + valueToAdd;
    Console.WriteLine(result);
}
```

The result of listing 14.1 shouldn't come as much surprise: it writes out "First (suffix)" and so on. Of course we could easily have specified the types of the `items` and `valueToAdd` variables explicitly in this case, and it would all have worked in the normal way - but imagine that the variables are getting their values from other data sources instead of having them hard-coded. What would happen if we wanted to add an integer instead of a string? Listing 14.2 is just a slight variation, but note that we haven't changed the *declaration* of `valueToAdd`; just the assignment expression.

Example 14.2. Adding integers to strings dynamically

```
dynamic items = new List<string> { "First", "Second", "Third" };
dynamic valueToAdd = 2;
foreach (dynamic item in items)
{
    string result = item + valueToAdd; ❶
    Console.WriteLine(result);
}
```

- ❶ string + int concatenation

This time the first result is "First2" - which is hopefully what you'd expect. Using static typing, we'd have to have explicitly change the declaration of `valueToAdd` from `string` to `int`. The addition operator is still building a string though. What if we changed the items to be integers as well? Let's try that one simple change, as shown in listing 14.3.

Example 14.3. Adding integers to integers

```
dynamic items = new List<int> { 1, 2, 3 };
dynamic valueToAdd = 2;
foreach (dynamic item in items)
{
    string result = item + valueToAdd; ❶
    Console.WriteLine(result);
}
```

❶ `int + int` addition

Disaster! We're still trying to convert the result of the addition to a string. The only conversions which are allowed are the same ones which are present in C# normally, so there's no conversion from `int` to `string`. The result is an exception (at execution time, of course):

```
Unhandled Exception:
  Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
Cannot implicitly convert type 'int' to 'string'
   at CallSite.Target(Closure , CallSite , Object )
   at System.Dynamic.UpdateDelegates.UpdateAndExecute1[T0,TRet]
      (CallSite site, T0 arg0)
  ...
```

Unless you're perfect, you're likely to encounter `RuntimeBinderException` quite a lot when you start using dynamic typing. It's the new `NullPointerException`, in some ways: you're bound to come across it, but with any luck it'll be in the context of unit tests rather than customer bug reports. Anyway, we can fix it by changing the type of `result` to `dynamic`, so that the conversion isn't required anyway. Come to think of it, why bother with the result variable in the first place? Let's just call `Console.WriteLine` immediately. Listing 14.4 shows the changes.

Example 14.4. Adding integers to integer - but without the exception

```
dynamic items = new List<int> { 1, 2, 3 };
dynamic valueToAdd = 2;
foreach (dynamic item in items)
{
    Console.WriteLine(item + valueToAdd); ❶
}
```

❶ Calls overload with `int` argument

Now this prints 3, 4, 5 as we'd expect. Changing the input data would now not only change the operator which was chosen at execution time - it would also change which overload of `Console.WriteLine` was called. With the original data, it would call `Console.WriteLine(string)`; with the updated variables it would call `Console.WriteLine(int)`. The data could even contain a mixture of values, making the exact call change on every iteration!

You can use `dynamic` as the declared type for fields, parameters and return types as well. This is in stark contrast to the use of `var`, which is restricted to local variables.

Differences between var and dynamic

In many of the examples so far, when we've really known the types at compile-time, we could have used `var` to declare the variables. At first glance, the two features look very similar. In both cases it looks like we're declaring a variable without specifying its type - but using `dynamic` we're explicitly setting the type to be dynamic. You can only use `var` when the compiler is able to infer the type you mean *statically*, and the type system really does remain entirely static.

The compiler is very smart about the information it records, and the code which then *uses* that information at execution time is clever too: basically it's a "mini C# compiler" in its own right. It uses whatever static type information was known at compile time to make the code behave as intuitively as possible. Other than a few details of what you *can't* do with dynamic typing, that's all you really need to know in order to start using it in your own code. Later on we'll come back to those restrictions, as well as details of what the compiler is actually doing - but first let's see dynamic typing doing something genuinely *useful*.

Examples of dynamic typing

Dynamic typing is a little bit like unsafe code, or interoperability with native code using `P/Invoke`. Many developers will have no need for it, or use it once in a blue moon. For other developers - particularly those dealing with Microsoft Office - it will give a huge productivity boost, either by making their existing code simpler or by allowing radically different approaches to their problems.

This section is not meant to be exhaustive by any means, and I look forward to seeing innovative uses of dynamic typing from C# in the coming years. Will unit testing and mocking take a big step forward with new frameworks? Will we see dynamic web service clients, accessing RESTful services with simple member access? I'm not going to make any predictions, other than that it'll be an interesting area to keep an eye on.

We're going to look at three examples here: working with Excel, calling into Python, and using normal managed .NET types in a more flexible way.

COM in general, and Microsoft Office in particular

We've already seen most of the new features C# 4 brings to COM interop, but there was one that we couldn't cover in chapter 13 because we hadn't seen dynamic typing yet. If you choose to embed the interop types you're using into the assembly (by using the `/link` compiler switch, or setting the "Embed Interop Types" property to `True`) then anything in the API which would otherwise be declared as `object` is changed to `dynamic`. This makes it much easier to work with somewhat weakly typed APIs such as those exposed by Office. (Although the object model in Office is reasonably strong in itself, many properties are exposed as *variants* as they can deal with numbers, strings dates and so on.)

Again, I'll just show you just a short example here - one which does even less than the Word example in chapter 13. The dynamic aspect is very easy to understand from just this one example - although there's a quirk you might not expect. We're going to set the first ten cells of the top row of a new Excel worksheet to the numbers 1 to 20. Listing 14.X shows an initial, statically typed piece of code to achieve this.

Example 14.5. Setting a range of values with static typing

```

var app = Application { Visible = true }; ❶
app.Workbooks.Add();
Worksheet worksheet = (Worksheet) app.ActiveSheet;
Range start = (Range) worksheet.Cells[1, 1]; ❷
Range end = (Range) worksheet.Cells[1, 20];
worksheet.get_Range(start, end).Value2 = Enumerable.Range(1, 20) ❸
                                                    .ToArray();

```

- ❶ Open Excel with an active worksheet
- ❷ Determine start and end cells
- ❸ Fill the range with [1, 20]

This time we've imported the `Microsoft.Office.Interop.Excel` namespace - so the `Application` type refers to Excel, not Word. We're still using the new features of C# 4, by not specifying an argument for the optional parameter in the `Workbooks.Add()` call while we're setting things up ❶. When Excel is up and running, we work out the start and end cells of our overall range. In this case they're both on the same row, but we could have created a rectangular range instead by selecting two opposite corners. We *could* have created the range in a single call to `get_Range("A1:T1")` but I personally find it easier to work with numbers consistently. Cell names like B3 are great for humans, but harder to use in a program.

Once we've got the range, we set all the values in it by setting the `Value2` property with an array of integers. We can use a one-dimensional array as we're only setting a single row; to set a range spanning multiple rows we'd need to use a rectangular array. This all works, but we've had to use three casts in six lines of code. The indexer we call via `Cells` and the `ActiveSheet` property are both declared to return `object` normally. (Various parameters are *also* declared as type `object`, but that doesn't matter as much because there's an implicit conversion from any type to `object` - it's only coming the other way that requires the cast.)

With the Primary Interop Assembly set to embed the required types into our own binary, all of these examples become `dynamic`. With the implicit conversion from `dynamic` to any other type, we can just remove all the casts, as shown in listing 14.X.

Example 14.6. Using implicit conversions from `dynamic` in Excel

```

var app = new Application { Visible = true };
app.Workbooks.Add();
Worksheet worksheet = app.ActiveSheet;
Range start = worksheet.Cells[1, 1];
Range end = worksheet.Cells[1, 20];
worksheet.get_Range(start, end).Value2 = Enumerable.Range(1, 20)
                                                    .ToArray();

```

This really is exactly the same code as listing 14.X but without the casts. However, it's worth noting that the conversions are still checked at execution time. If we changed the declaration of `start` to be `Worksheet`, the conversion would fail and an exception would be thrown. Of course, you don't *have* to perform the conversion. You *could* just leave everything as `dynamic`:

```

var app = new Application { Visible = true };
app.Workbooks.Add();
dynamic worksheet = app.ActiveSheet;
dynamic start = worksheet.Cells[1, 1];
dynamic end = worksheet.Cells[1, 20];
worksheet.get_Range(start, end).Value2 = Enumerable.Range(1, 20)

```

```
.ToArray();
```

This approach has two problems. First, you don't get any IntelliSense on worksheet, start and end variables, because the compiler doesn't know the real types involved. More importantly, the code throws an exception on the last line: the COM dynamic binder fails on the call to `get_Range`. This is easy to fix by changing the method call to `Range` instead, as shown in listing 14.X - but the important point to take away is that code which works with static typing doesn't *always* work with dynamic typing.

Example 14.7. Using `dynamic` excessively, requiring a change in method name

```
var app = new Application { Visible = true };
app.Workbooks.Add();
dynamic worksheet = app.ActiveSheet;
dynamic start = worksheet.Cells[1, 1];
dynamic end = worksheet.Cells[1, 20];
worksheet.Range(start, end).Value2 = Enumerable.Range(1, 20)
                                                .ToArray();
```

For this reason, I'd encourage you to use static typing as far as possible even when using COM. The implicit conversion of `dynamic` is very useful in terms of removing casts, but taking it too far is dangerous - as well as inconvenient due to losing IntelliSense.

From the relatively old technology of COM, we're going to jump to interoperating with something much more recent: IronPython.

Dynamic languages such as IronPython

In this section I'm only going to use IronPython as an example, but of course that's not the only dynamic language available for the DLR. It's arguably the most mature, but there are already alternatives such as IronRuby and IronScheme. One of the stated aims of the DLR is to make it easier for budding language designers to create a working language which has good interoperability with other DLR languages and the traditional .NET languages such as C#, as well as access to the huge .NET framework libraries.

Why would I want to use IronPython from C#?

There are many reasons one might want to interoperate with a dynamic language, just as it's been beneficial to interoperate with other managed languages from .NET's infancy. It's clearly useful for a VB developer to be able to use a class library written in C# and vice versa - so why would the same not be true of dynamic languages? I asked Michael Foord, the author of *Iron Python in Action*, to come up with a few ideas for using IronPython within a C# application. Here's his list:

- User scripting
- Writing a layer of your application in IronPython
- Using Python as a configuration language
- Using Python as a rules engine with rules stored as text (even in a database)
- Using a Python library such as feedparser [<http://www.feedparser.org/>]
- Putting a live interpreter into your application for debugging

If you're still skeptical, you might want to consider that embedding a scripting language in a mainstream application is far from uncommon - indeed, Sid Meyer's Civilization IV computer game⁴ is scriptable with

⁴Or way of life, depending on how you view the world and your level of addiction to playing the game.

Python. This isn't just an afterthought for modifications, either - a lot of the core gameplay is written in Python: once they'd built the engine the developers found it to be a more powerful development environment than they'd originally imagined.

For this chapter, I'm going to pick the single example of using Python as a configuration language. Just as with the COM example, I'm going to keep it very simple, but hopefully it'll provide enough of a starting point for you to experiment more with it if you're interested.

Getting started: embedding "hello, world"

MEAP note: this is based on Visual Studio 4.0 beta 1 and IronPython 2.6-with-.NET 4.0 CTP 1. It may well change in terms of namespaces etc, which is why I haven't been precise yet. This will be fixed before publication!

There are various types available if you want to *host* or *embed* another language within a C# application, depending on the level of flexibility and control you want to achieve. We're only going to use `ScriptEngine` and `ScriptScope`, because our requirements are quite primitive. In our example we know we're always going to use Python, so we can ask the IronPython framework to create a `ScriptEngine` directly; in more general situations you can use a `ScriptRuntime` to pick language implementations dynamically by name. More demanding scenarios may require you to work with `ScriptHost` and `ScriptSource`, as well as using more of the features of the other types too.

Not content with merely printing "hello, world" once, our initial example will do so *twice*, first by using text passed directly into the engine as a string, and then by loading a file called `HelloWorld.py`.

Example 14.8. Printing "hello, world" twice using Python embedded in C#

```
ScriptEngine engine = Python.CreateEngine();
engine.Execute("print 'hello, world'");
engine.ExecuteFile("HelloWorld.py");
```

You may find this listing either quite dull or very exciting, both for the same reason. It's simple to understand, requiring very little explanation. It does very little, in terms of actual output... and yet the fact that it *is* so easy to embed Python code into C# is a cause for celebration. True, our level of interaction is somewhat minimal so far - but it really couldn't be much easier than this.

Note

The Python file contains a single line: `print "hello, world"` - note the double quotes in the file compared with the single quotes in the string literal we passed into `engine.Execute()`. Either would have been fine in either source. Python has various string literal representations, including triple single quotes or triple double quotes for multi-line literals. I only mention this because it's very useful not to have to escape double quotes any time you want to put Python code into a C# string literal.

The next type we need is `ScriptScope`, which will be crucial to our configuration script.

Storing and retrieving information from a `ScriptScope`

The execution methods we've used both have overloads with a second parameter - a scope. In its simplest terms, this can be regarded as a dictionary of names and values. Scripting languages often allow variables to be assigned without any explicit declaration, and when this is done in the top level of a program (instead of in a function or class) this usually affects a *global scope*. When a `ScriptScope` instance is passed into

an execution method, that is the global scope for the script you've asked the engine to execute. The script can retrieve existing values from the scope, and create new values, as shown in listing 14.X.

Example 14.9. Passing information between a host and the hosted script using `ScriptScope`

```
string python = @"
text = 'hello' ❶
output = input + 1
";
ScriptEngine engine = Python.CreateEngine();
ScriptScope scope = engine.CreateScope();
scope.SetVariable("input", 10); ❷
engine.Execute(python, scope);
Console.WriteLine(scope.GetVariable("text")); ❸
Console.WriteLine(scope.GetVariable("input"));
Console.WriteLine(scope.GetVariable("output"));
```

❶❶ Python code embedded as a C# string literal

❷❶ Sets variable for Python code to use

❸❶ Fetches variables back from scope

I've embedded the Python source code into the C# code as a verbatim string literal ❶ rather than putting it in a file, so that it's easier to see all the code in one place. I don't recommend that you do this in production code, partly because Python is sensitive to whitespace - reformatting the code in a seemingly-harmless way can make it fail completely at execution time.

The `SetVariable` and `GetVariable` methods simply put values into the scope ❶ and fetch them out again ❷ in the obvious way. They're declared in terms of `object` rather than `dynamic`, as you might have expected. However, `GetVariable` also allows you to specify a type argument, which acts as a conversion request. This is not quite the same as just casting the result of the nongeneric method, as the latter just unboxes the value - which means you need to cast it to exactly the right type. For example, we can put an integer into the scope, but retrieve it as a double:

```
scope.SetVariable("num", 20)
double x = scope.GetVariable<double>("num") ❶
double y = (double) scope.GetVariable("num"); ❷
```

❶❶ Converts successfully to double

❷❶ Unboxing throws exception

The scope can also hold functions which we can retrieve and then call dynamically, passing arguments and returning values. The easiest way of doing this is to use the `dynamic` type, as shown in listing 14.X.

Example 14.10. Calling a function declared in a `ScriptScope`

```
string python = @"
def sayHello(user):
    print 'Hello %(name)s' % {'name' : user}
";
ScriptEngine engine = Python.CreateEngine();
ScriptScope scope = engine.CreateScope();
engine.Execute(python, scope);
dynamic function = scope.GetVariable("sayHello");
function("Jon");
```

Configuration files may not often need this ability, but it can be useful in other situations. For example, you could easily use Python to script a graph-drawing program, by providing a function to be called on each input point. A simple example of this can be found on the book's web site.

Putting it all together

Now that we can get values into our scope, we're essentially done. We could potentially wrap the scope in another object providing access via an indexer - or even access the values dynamically using the techniques shown in section 14.5. The application code might look something like this:

```
static Configuration LoadConfiguration()  
{  
    ScriptEngine engine = Python.CreateEngine();  
    ScriptScope scope = engine.CreateScope();  
    engine.ExecuteFile("configuration.py", scope);  
    return Configuration.FromScriptScope(scope);  
}
```

The exact form of the `Configuration` type will depend on your application, but it's unlikely to be terribly exciting code. I've provided a sample dynamic implementation in the full source, which allows you to retrieve values as properties and call functions directly too. Of course we're not limited to just using primitive types in our configuration: the Python code could be arbitrarily complex, building collections, wiring up components and services and so forth. Indeed, it could perform a lot of the roles of a normal Dependency Injection or Inversion of Control container.

The important thing is that we now have a configuration file which is *active* instead of the traditional passive XML and .ini files. Of course, you could have embedded your own programming language into previous configuration files - but the result would probably have been less powerful, and would have taken a lot more effort to implement. As an example of where this could be useful in a simpler situation than full dependency injection, you might want to configure the number of threads to use for some background processing component in your application. You might normally use as many threads as you have processors in the system, but occasionally reduce it in order to help another application run smoothly on the same system. The configuration file would simply change from something like this:

```
agentThreads = System.Environment.ProcessorCount  
agentThreadName = 'Processing agent'
```

To this:

```
agentThreads = 1  
agentThreadName = 'Processing agent (single thread only)'
```

This change wouldn't require the application to be rebuilt or redeployed - just edit the file and restart the application.

Other than executing functions, we haven't really looked at using Python in a particularly dynamic way. The full power of Python is available, and using the dynamic type in your C# code you can take advantage of meta-programming and all the other dynamic features. The C# compiler is responsible for representing your code in an appropriate fashion, and the script engine is responsible for taking that code and working out what it means for the Python code. Just don't feel you *have* to be doing anything particularly clever for it to be worth embedding the script engine in your application. It's a simple step towards a more powerful application.

So far our examples have been interoperating with other systems. Dynamic typing can make sense even within a purely managed system, however. You may well have implemented it yourself in a very limited sense, with reflection. If you're lucky, a lot of that reflection code can be completely eliminated using dynamic typing.

Reflection

Reflection is effectively the manual way of doing dynamic typing. In my experience, it's very easy to make mistakes when writing code to use reflection, and even when it's working you often need to put extra effort in to optimise it. In this section we'll look at a few examples of using dynamic typing, and we'll go into a bit more detail than in the previous sections, as the examples will lead us into a wider discussion of what exactly is going on behind the scenes.

It's particularly tricky to use generic types and methods from reflection. For example, if you have an object which you know implements `ICollection<T>` for some type argument `T`, it can be very difficult to work out exactly what `T` is. If the only reason for discovering `T` is to then call another generic method, you really want to just ask the compiler to call whatever it *would* have called if you knew the actual type. Of course, that's exactly what dynamic typing does.

Execution-time type parameter inference

If you want to do more than just call a single method, it's often best to wrap all the additional work in a generic method. You can then call the generic method dynamically, but write all the rest of the code using static typing. Listing 14.X shows a simple example of this. We're going to pretend we've been given a list of some type and a new element by some other part of the system. We've been promised that they're compatible, but we don't know their types statically. There are various reasons this could happen; in particular there are some type relationships which C# just can't express. Anyway, our code is meant to add the new element to the end of the list, but only if there are fewer than ten elements in the list at the moment. The method returns whether or not the element was actually added. Obviously in real life the business logic would be more complicated, but the point is that we'd really like to be able to use the strong types for these operations.

Example 14.11. Using dynamic type inference

```
private static bool AddConditionallyImpl<T>(ICollection<T> list, T item)
{
    if (list.Count < 10) ❶
    {
        list.Add(item);
        return true;
    }
    return false;
}

public static bool AddConditionally(dynamic list, dynamic item)
{
    return AddConditionallyImpl(list, item); ❷
}

...
List<string> list = new List<string> { "x", "y" };
Console.WriteLine(AddConditionally(list, "z")); ❸
Console.WriteLine(list.Count); ❹
```

- ❶❶ Normal statically typed code
- ❷❷ Call helper method dynamically
- ❸ Prints "True" (item added)
- ❹ Prints "3"

The public method takes dynamic arguments: in previous versions of C# it would perhaps have taken `IEnumerable` and `Object`, relying on complicated checks with reflection to work out the type of the

list and then act appropriately. With dynamic typing, we can just call a strongly-typed implementation **❶** using the dynamic arguments **❷**, isolating the dynamic access to the single call in the wrapper method.

Of course, we could also expose the strongly-typed method publicly to avoid the dynamic typing for callers who knew their list types statically. It would be worth keeping the names different in that case, to avoid accidentally calling the dynamic version due to a slight mistake with the static types of the arguments. (It also makes it a lot easier to make the right call within the dynamic version when the names are different!)

As another example of I've already bemoaned the lack of generic operator support in C# - there's no concept of specifying a constraint saying "T has to have an operator which allows me to add two values of type T together." We used this in our initial demonstration of dynamic typing, so mentioning it here should come as no surprise. We'll take a slightly deeper look at this example, as it raises some interesting general points about dynamic typing.

Summing values dynamically

Have you ever looked at the list of overloads for `Enumerable.Sum`? It's pretty long. Admittedly half of the overloads are due to a projection, but even so there are 10 overloads, each of which just takes a sequence of elements and adds them together... and that doesn't even cover summing unsigned values, or bytes or shorts. How about we use dynamic typing to try to do it all in one method?

There are actually two approaches here, which have different pros and cons. We could write a new generic method which sums an `IEnumerable<T>` without restriction, or we could write one which sums an `IEnumerable<dynamic>` and then rely on interface variance (introduced in the next chapter) and an extra method to convert to an `IEnumerable<dynamic>` when we needed to. The mixture of dynamic and generics can get slightly hairy, with some surprising problems which we'll see later, so for the purposes of simplicity we'll sum `IEnumerable<T>`⁵. Listing 14.X shows an initial implementation which does pretty well, although it's not ideal. I've named the method `DynamicSum` rather than `Sum` to avoid clashing with the methods in `Enumerable`: the compiler will pick a non-generic overload over a generic one where both signatures have the same parameter types, and it's just simpler to avoid the collision in the first place.

Example 14.12. Summing an arbitrary sequence of elements dynamically

```
public static T DynamicSum<T>(this IEnumerable<T> source)
{
    dynamic total = default(T); ❶
    foreach (T element in source)
    {
        total += element; ❷
    }
    return total;
}
...
byte[] bytes = new byte[] { 1, 2, 3 };
Console.WriteLine(bytes.DynamicSum()); ❸
```

- ❶** Dynamically typed for later use
- ❷** Choose addition operator dynamically
- ❸** Prints "6"

The code is very straightforward: it looks almost exactly the same as any of the implementations of the normal `Sum` overloads would. I've omitted checking whether `source` is null just for brevity, but most of

⁵An article [<http://csharpindepth.com/Articles/Chapter13/FIXME.html>] discussing summing `IEnumerable<dynamic>` is available on the book's web site.

the rest is what you'd expect, other than possibly the use of `default(T)` to initialize `total`, which is declared as `dynamic` so that we get the desired dynamic behavior. Sure enough, running the code prints 6 as we'd expect it to.

We have to start off with an initial value somehow: we could try to use the first value in the sequence, but then we'd be stuck if the sequence were empty. For non-nullable value types, `default(T)` is almost always an appropriate value anyway: it's a natural zero. For reference types, we'll end up adding the first element of the sequence to `null`, which may or may not be appropriate. For nullable value types, we'll end up trying to add the first element to the null value for that type, which certainly *won't* be appropriate.

We could improve things somewhat by adding an overload which takes the "zero value" as another parameter, which would be okay for reference types but still wouldn't help much for nullable types...

Summing sequence of nullable values

The problem is that the way C# defines operators working with nullable types, if *any* value in the sequence is null, the result would end up being null. Assuming that we don't want that, and instead we want the same behavior as the existing `Enumerable.Sum` methods working over nullable types, we need to introduce a new method, as shown in listing 14.X:

Example 14.13. Summing nullable value types dynamically

```
public static T DynamicSum<T>(this IEnumerable<T?> source)
    where T : struct
{
    dynamic total = default(T);
    foreach (T? element in source)
    {
        if (element != null) ❶
        {
            total += element.Value;
        }
    }
    return total;
}
```

❶❶ Only sum non-null values

Again, the code is simple - once you've got your head round the fact that here `T` is the *non-nullable* type involved: if we were summing a `List<int?>` for example, `T` would be `int`. The result of the sum is always non-null, and we start off with the default value of the non-nullable type. This time when we iterate through the sequence, we only use non-null values ❶ (where "null" here means "the null value for the nullable type", not a null reference) and we add the underlying non-nullable value to the current total.

Even though we're overloading the `DynamicSum` method here, the new method will be called in preference to the old one when both are applicable: `T?` is always *more specific* than `T`, because there's an implicit conversion from `T` to `T?` but only an explicit conversion from `T?` to `T`. The overload resolution rules are tricky to work through, but this time they work in our favor.

It's worth noting the mixture of dynamic typing and static typing here. The method starts with the declaration of the `dynamic total` variable, but the iteration itself, the null check and the extraction of the underlying value are all compiled statically. This is easy to verify: if you change `element.Value` to `element.VALUE` you'll see a compilation error:

```
error CS1061: 'System.Nullable<T>' does not contain a definition
    for 'VALUE' and no extension method 'VALUE' accepting a first
```



```
argument of type 'System.Nullable<T>' could be found (are you
missing a using directive or an assembly reference?)
```

This is reassuring. We can restrict dynamic operations to just the ones that we *need* to be dynamic: we get the benefits of dynamic typing in terms of discovering the addition operator at execution time, but most of the code still benefits from the compile-time checking and performance benefits of static typing. This is similar to the dynamic `AddConditional` method in listing 14.X calling the statically typed version, but here the static and dynamic code appear within the same method.

Speaking of the addition operator, we're effectively duck-typing on it in this case. This is a good example of an area where we simply *can't* express the requirements statically - it's impossible to specify operators in an interface, so even if we had complete control of the framework, we couldn't really do any better. In many other situations, duck-typing is useful when dealing with disparate frameworks which happen to use the same member names but don't implement a common interface.

So, we now have two methods to sum a sequence of any type, so long as that type has an addition operator which results in a value of the same type. Hang on though... earlier on we summed a sequence of bytes. What's going on?

Attention to detail: binding operators

There's no addition operator defined for the `byte` type. If you try to add two bytes together, both are promoted to `int` values, and the return value is an `int`. There's one exception to this, however: the compound assignment `+=` operator is permitted, effectively converting both operands to integers and then casting back to `byte` at the end. We can see this in action if we try to sum values which overflow the range of `byte`:

```
byte[] values = new byte[] { 100, 100, 100 };
Console.WriteLine(values.DynamicSum());
```

By default, the result printed here is 44: when the operation overflowed, the result was truncated. That's *may* be what we want, but it goes against the behavior of the built-in `Enumerable.Sum` methods which throw an `OverflowException`. There are actually two alternatives to the current implementation. The first is to mimic `Enumerable.Sum`. We don't need to be clever here, because overflow-safe arithmetic is easy in C#: we just need to work in a *checked context*. We do this in dynamic code in the same way as we would in traditional C#. Listing 14.X shows a checked implementation of `DynamicSum`.

Example 14.14. Summing dynamically in a checked context

```
public static T DynamicSum<T>(this IEnumerable<T> source)
{
    dynamic total = default(T);
    checked
    {
        foreach (T element in source)
        {
            total += element;
        }
    }
    return total;
}
...
byte[] values = new byte[] { 100, 100, 100 };
Console.WriteLine(values.DynamicSum()); ❶
```

❶ Throws `OverflowException`

The other alternative is to avoid restricting the result to byte in the first place. The compiler knows how to promote values and add them, it's just that we'll get back an `int` instead of a byte. Of course, we don't know the types involved at compile-time, and we can't describe them generically... but we can change the method to return a dynamic value too. Then we just need to change how the addition is performed, and we're away. Listing 14.X shows this final change.

Example 14.15. Summing with a dynamic result and automatic byte to int promotion

```
public static dynamic DynamicSum<T>(this IEnumerable<T> source)
{
    dynamic total = default(T);
    foreach (T element in source)
    {
        total = total + element;
    }
    return total;
}
...
byte[] values = new byte[] { 100, 100, 100 };
Console.WriteLine(values.DynamicSum()); ❶
```

❶ Prints 300

Of course, as listing 14.X is no longer operating in a checked context, we could eventually overflow. However, the point of the example was to demonstrate that the normal rules for C# operators are obeyed, even though they're being bound at execution time. The statements `x+=y;` and `x=x+y;` may look very similar, but we've seen how they can behave very differently - and we'd only notice this difference at execution time when we're using dynamic code. Be careful!

Note that these are C# rules which are being applied here - not .NET rules. In situations where Visual Basic and C# would handle things differently, dynamic code compiled in one of the two languages will follow the rules of the compiler for that language.

I should warn you that things are about to get tricky. In fact, it's all extremely elegant, but it's complicated because programming languages provide a rich set of operations, and representing all the necessary information about those operations as data and then acting on it appropriately is a complex job. The good news is that you don't need to understand it all intimately. As ever, you'll get more out of dynamic typing the more familiar you are with the machinery behind it, but even if you just use the techniques we've seen so far there may well be situations where it makes you a lot more productive.

Looking behind the scenes

Despite the warning of the previous paragraph, I'm not going to go into *huge* amounts of detail about the inner workings of dynamic typing. There would be an awful lot of ground to cover, both in terms of the framework and language changes. It's not often that I shy away from the nitty-gritty of specifications, but in this case I truly believe there's not much to be gained from learning it all. I'll cover the most important (and interesting) points of course, and I can thoroughly recommend Sam Ng [<http://blogs.msdn.com/samng>]'s blog, the C# language specification and the DLR project page [<http://dlr.codeplex.com/Wiki/View.aspx?title=Docs%20and%20specs>] for more information if you need to dig into a particular scenario.

Our eventual goal is to understand what the C# compiler is doing - the code it emits to achieve dynamic binding at execution time. Unfortunately, none of the generated code will make any sense until we see the

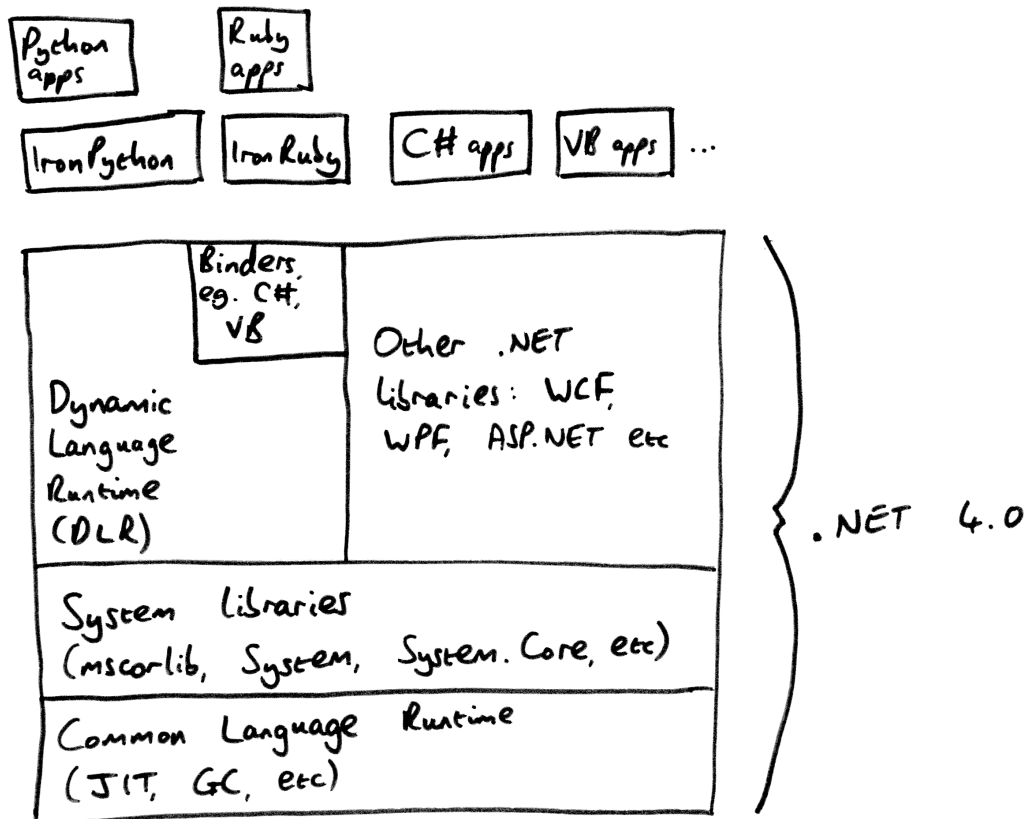
mechanism that underpins it all - namely the DLR. You might like to think of a statically typed program as a conventional stage play with a fixed script, and a dynamically typed program as more like an improvisation show. The DLR takes the place of the actors' brains frantically coming up with something to say in response to audience suggestions. Let's meet our quick-thinking star performer.

Introducing the Dynamic Language Runtime

I've been bandying the acronym "DLR" around for a while now, occasionally expanding it to "Dynamic Language Runtime" but never really explaining what it is. This has been entirely deliberate: I've been trying to get across the nature of dynamic typing and how it affects developers, rather than the details of the implementation. However, that excuse was never going to last until the end of the chapter - so here we are. In its barest terms, the Dynamic Language Runtime is a library which all dynamic languages and the C# compiler use to execute code dynamically.

Amazingly enough, it really is just a library. Despite its name, it isn't at the same level as the CLR (Common Language Runtime) - it doesn't deal in JIT compilation, native API marshalling, garbage collection and so forth. However, it builds on a lot of the work in .NET 2.0 and 3.5, particularly the `DynamicMethod` and `Expression` types. The expression tree API has been expanded in .NET 4.0 to allow the DLR to express more concepts, too. Figure 14.X shows how it all fits together.

Figure 14.1. How .NET 4.0 fits together



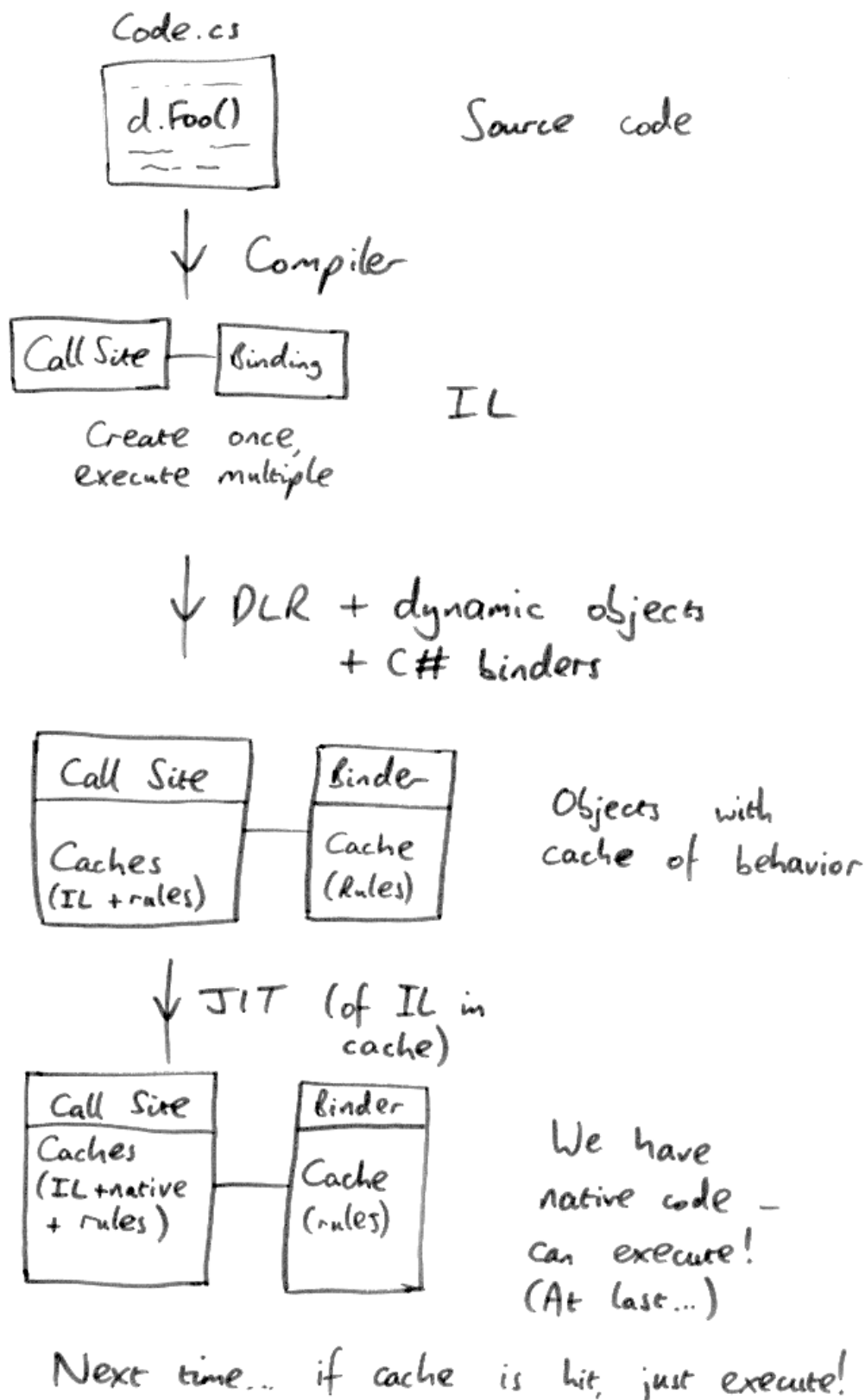
As well as the DLR, there's another library which may be new to you in figure 14.X. The `Microsoft.CSharp` assembly contains a number of types which are referenced by the C# compiler when you use `dynamic` in your code. Confusingly, this doesn't include the existing

`Microsoft.CSharp.Compiler` and `Microsoft.CSharp.CodeDomProvider`. (They're not even in the same assembly as each other!) We'll see exactly what the new types are used for in section 14.4.3, where we decompile some code written using `dynamic`.

There's one other important aspect which differentiates the DLR from the rest of the .NET framework: it's provided as Open Source. The complete code lives in a CodePlex project [<http://dlr.codeplex.com>], so you can download it and see the inner workings. One of the benefits of this approach is that the DLR hasn't had to be reimplemented for Mono [<http://mono-project.com>]: the same code runs on both .NET and its cross-platform cousin.

Although the DLR doesn't handle native code directly, you can think of it as doing a *similar* job to the CLR in one sense: just as the CLR converts IL (Intermediate Language) into native code, the DLR converts code represented using binders, call sites, meta-objects and various other concepts into expression trees which can then be compiled down into IL and eventually native code by the CLR. Figure 14.X shows a simplified view of the lifecycle of a single evaluation of a dynamic expression.

Figure 14.2. Lifecycle of a dynamic expression



As you can see, one of the important aspects of the DLR is a multi-level cache. This is crucial for performance reasons, but to understand that and the other concepts we've already mentioned, we'll need to dive one layer lower.

DLR core concepts

We can summarise the purpose of the DLR in *very* general terms as taking a high-level representation of code, and executing that code, based on various pieces of information which may only be known at execution time. In this section I'm going to introduce a lot of terminology to describe how the DLR works, but it's all contributing to that common aim.

Call sites

The first concept we need is a *call site*. This is the sort of atom of the DLR - the smallest piece of code which can be considered as a single unit. One expression may contain a lot of call sites, but the behavior is built up in the natural way, evaluating one call site at a time. For the rest of the discussion, we'll only consider a single call site at a time. It's going to be useful to have a small example of a call site to refer to, so here's a very simple one, where `d` is of course a variable of type `dynamic`.

```
d.Foo(10);
```

The call site is represented in code as a `System.Runtime.CompilerServices.CallSite<T>`. We'll see some details of how call sites are actually created in the next section, when we look at what the C# compiler does at compile-time.

Receivers and binders

Once we've got a call site, something has to decide what it means and how to execute it. In the DLR, there are two entities which can decide this: the *receiver* of a call, and the *binder*. The receiver of a call is simply the object that a member is called on. In our sample call site, the receiver is the object that `d` refers to at execution time. The binder will depend on the calling language - in this case, the C# compiler emits code to create a `CSharpInvokeMemberBinder`.

The DLR always gives precedence to the receiver: if it's a dynamic object which knows how to handle the call, then it will use whatever execution path the object provides. An object can advertise itself as being dynamic by implementing the new `IDynamicMetaObjectProvider` interface. The name is a mouthful, but it only contains a single element: `GetMetaObject`. You'll need to be a bit of an expression tree ninja to implement it correctly, as well as knowing the DLR quite well. However, in the right hands this can be a very powerful tool, giving you the lower level interaction with the DLR and its execution cache. If you need to implement dynamic behavior in a high-performance fashion, it's worth the investment of learning the details. There are two implementations of `IDynamicMetaObjectProvider` included in the framework, however, to make it easy to implement dynamic behavior in situations where performance isn't quite as critical. We'll look at all of this in more detail in section 14.5, but for now you just need to be aware of the interface itself, and that it represents the ability of an object to react dynamically.

If the receiver isn't dynamic, the binder gets to decide how the code should be executed. In our code, it would apply C#-specific rules to the code, and work out what to do. If you were creating your own dynamic language, you could implement your own binder to decide how it should behave in general (when the object doesn't override the behavior). This lies well beyond the scope of this book, but it's an interesting topic in and of itself: one of the aims of the DLR is to make it easier to implement your own languages.

Rules and caches

The decision for how to execute a call is represented as a *rule*. Fundamentally this consists of two elements of logic: the circumstances under which the call site should behave this way, and the behavior itself. The

The second part of a rule is the code to use when the rule matches, and it's represented as an expression tree. It *could* have been stored just as a compiled delegate to call - but keeping the expression tree representation means the cache can really optimise heavily. There are three levels of cache in the DLR: L0, L1 and L2. The caches store information in different ways, and with a different scope. Each call site has its own L0 and L1 caches, but an L2 cache may be shared between several similar call sites, as shown in figure 14.X.

Call Site

L \emptyset cache:
delegate

L1 cache:
rules (few)

Binder

L2 cache:
rules (many)

Call Site

Call Site ...

(Call sites with the same semantics)

The caches themselves are executable, which takes a little while to understand. The C# compiler generates code to simply execute the call site's L0 cache (which is a delegate accessed through the `Target` property). That's it! When L0 cache delegate is called, it looks through the rules it knows about, and if it finds a matching rule it executes the associated behavior. If it doesn't find any matches, it calls into the L1 cache, which in turn calls into the L2 cache. If the L2 cache can't find any matching rules, it asks the receiver or the binder to resolve the call. The results are then put into the cache for next time.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=569>

rebuilding the method from the new set of rules. This is where it becomes vital that the rules are available as expression trees: it's an awful lot easier to stitch together several expression trees than to analyze the IL generated for each rule and combine that together.

The result of all of this is that typical call sites which see similar context repeatedly are very, very fast; the dispatch mechanism is as about as lean as you could make it if you hand-coded the tests yourself. Of course this has to be weighed against the cost of all the dynamic code generation involved, but the multi-level cache is complicated precisely because it tries to achieve a balance across various different scenarios.

Now that we know a bit about the machinery in the DLR, we'll be able to understand what the C# compiler does for us in order to set it all in motion.

How the C# compiler handles `dynamic`

The main jobs of the C# compiler when it comes to dynamic code are to work out when dynamic behavior is required, and to capture all the necessary context so that the binder and receiver have enough information to resolve the call at execution time.

If it uses `dynamic`, it's `dynamic`!

There's one situation which is very obviously dynamic: when the target of a member call is dynamic. The compiler has no way of knowing how that will be resolved. It may be a truly dynamic object which will perform the resolution itself, or it may end up with the C# binder resolving it with reflection later. Either way, there's simply no opportunity for the call to be resolved statically.

However, when the dynamic value is being used as an *argument* for the call, there are some situations where you *might* expect the call to be resolved statically - particularly if there's a suitable overload which has a parameter type of `dynamic`. However, the rule is that if any part of a call is dynamic, the call becomes dynamic and will resolve the overload with the execution-time type of the dynamic value. Listing 14.X demonstrates this using a method with two overloads, and invoking it in a number of different ways.

Example 14.16. Experimenting with method overloading and dynamic values

```
static void Execute(string x)
{
    Console.WriteLine("String overload");
}

static void Execute(dynamic x)
{
    Console.WriteLine("Dynamic overload");
}

...
dynamic text = "text";
Execute(text); ❶
dynamic number = 10;
Execute(number); ❷
```

❶ Prints "String overload"

❷ Prints "Dynamic overload"

Both calls to `Execute` are bound dynamically. At execution time they are resolved using the types of the actual values, namely `string` and `int`. The parameter of type `dynamic` is treated as if it were declared

with type `object` for the purposes of overload resolution - indeed, if you look at the compiled code you'll see it *is* a parameter of type `object`, just with an attribute applied. This also means you can't have two methods who signatures differ just by `dynamic/object`. Speaking of looking at compiled code, let's dig into the IL generated for dynamic calls.

Creating call sites and binders

You don't need to know the details of what the compiler does with dynamic expressions in order to use them, but it can be instructive to see what the compiled code looks like. In particular, if you need to decompile your code for any other reason, it means you won't be surprised by what the dynamic parts look like. My tool of choice for this kind of work is Reflector [<http://www.red-gate.com/products/reflector/>], but you could use **ildasm** if you wanted to read the IL directly.

We're only going to look at a single example - I'm sure I could fill a whole chapter by looking at implementation details, but the idea is only to give you the gist of what the compiler is up to. If you find this example interesting, you may well want to experiment more on your own. Just remember that the exact details are implementation-specific; they may change in future compiler versions, so long as the behavior is equivalent. Here's the sample snippet, which exists in a `Main` method in the normal manner for Snippet:

```
string text = "text to cut";  
dynamic startIndex = 2;  
string substring = text.Substring(startIndex);
```

Pretty simple, right? Well, it actually contains two dynamic operations - one to call `Substring`, and one to dynamically convert the result (which is just `dynamic` at compile-time) to a string. Listing 14.X shows the decompiled code for the `Snippet` class. I've omitted the class declaration itself and the implicit parameterless constructor, just to save space - and I've reformatted the code with significantly reduced whitespace for the same reason.

Example 14.17. The results of compiling dynamic code

```
[CompilerGenerated]
private static class <Main>o__SiteContainer0 { ❶
    public static CallSite<Func<CallSite, object, string>> <>p__Site1;
    public static CallSite<Func<CallSite, string, object, object>>
        <>p__Site2;
}

private static void Main() {
    string text = "text to cut";
    object startIndex = 2;
    if (<Main>o__SiteContainer0.<>p__Site1 == null) { ❷
        <Main>o__SiteContainer0.<>p__Site1 =
            CallSite<Func<CallSite, object, string>>.Create(
                new CSharpConvertBinder(typeof(string),
                    CSharpConversionKind.ImplicitConversion, false));
    }
    if (<Main>o__SiteContainer0.<>p__Site2 == null) { ❸
        <Main>o__SiteContainer0.<>p__Site2 =
            CallSite<Func<CallSite, string, object, object>>.Create(
                new CSharpInvokeMemberBinder(CSharpCallFlags.None,
                    "Substring", typeof(Snippet), null,
                    new CSharpArgumentInfo[] {
                        new CSharpArgumentInfo(
                            CSharpArgumentInfoFlags.UseCompileTimeType, null), ❹
                        new CSharpArgumentInfo(
                            CSharpArgumentInfoFlags.None, null) }));
    }
    string substring = ❺
        <Main>o__SiteContainer0.<>p__Site1.Target.Invoke(
            <Main>o__SiteContainer0.<>p__Site1,
            <Main>o__SiteContainer0.<>p__Site2.Target.Invoke(
                <Main>o__SiteContainer0.<>p__Site2, text, startIndex));
}
```

- ❶ Storage for call sites
- ❷ Creation of conversion call site
- ❸ Creation of substring call site
- ❹ Preserve static type of text variable
- ❺ Invocation of both calls

I don't know about you, but I'm jolly glad that I never have to write or encounter code quite like that, other than for the purpose of learning about exactly what's going on. There's nothing new about that though - the generated code for iterator blocks, expression trees and anonymous functions can be pretty gruesome too.

There's a nested static type which is used to store all the call sites ❶, as they only need to be created once. (And indeed if they were created each time the cache would be useless!) It's possible that the call sites *could* be created more than once due to multi-threading, but if that happens it's just slightly inefficient - and it means the lazy creation is achieved with no locking at all. It doesn't really matter if one call site instance is replaced with another.

After the call sites are created (❶ and ❷) they are simply invoked. The substring call is invoked first (read the code from the innermost part of the statement outwards) and then the conversion is invoked on the result ❸. At this point we have a statically typed value again, so we can assign it to the `substring` variable.

There's one more aspect to this code which I'd like to highlight, and that's the way that some static type information is preserved in the call site. The type information itself is present in the delegate signature used for the type argument of the call site (`Func<CallSite, string, object, object>`) and a flag in the corresponding `CSharpArgumentInfo` indicates that this type information should be used in the binder ❶. (Even though this is the target of the method, it's represented as an argument; instance methods are treated as static methods with an implicit first parameter of "this".) This is a crucial part of making the binder behave as if it were just recompiling your code at execution time. Let's take a look at why this is so important.

The C# compiler gets even smarter

C# 4 lets you straddle the static/dynamic boundary not just by having some of your code bound statically and some bound dynamically, but also by combining the two ideas within a single binding. It remembers everything it needs to know within the call site, then cleverly works merges this information with the types of the dynamic values at execution time.

Preserving compiler behavior at execution time

The ideal model for working out how the binder should behave is to imagine that instead of having a dynamic value in your source code, you have a value of exactly the right type: the type of the actual value at execution time. However, that only applies for dynamic values; any value with a type known at compile time is treated as being of that statically-determined type; the actual value isn't used for lookups such as member resolution. I'll give two examples of where this makes a difference. Listing 14.X shows a simple overloaded method in a single type.

Example 14.18. Dynamic overload resolution within a single type

```
static void Execute(dynamic x, string y)
{
    Console.WriteLine("dynamic, string");
}

static void Execute(dynamic x, object y)
{
    Console.WriteLine("dynamic, object");
}
...
object text = "text";
dynamic d = 10;
Execute(d, text); ❶
```

❶ Prints "dynamic, object"

The important variable here is `text`. Its *compile-time* type is `object`, but at *execution time* it's a string. The call to `Execute` is dynamic because we're using the dynamic variable `d` as one of the arguments, but the overload resolution uses the static type of `text`, so the result is `dynamic, object`. If the `text` variable had been declared as `dynamic` as well, it would have used the other overload.

Listing 14.X is similar, but this time it's the receiver of the call which matters.

Example 14.19. Dynamic overload resolution within a class hierarchy

```
class Base
{
    public void Execute(object x)
    {
        Console.WriteLine("object");
    }
}

class Derived : Base
{
    public void Execute(string x)
    {
        Console.WriteLine("string");
    }
}

...
Base receiver = new Derived();
dynamic d = "text";
receiver.Execute(d); ❶
```

❶ Prints "object"

In listing 14.X, the type of `receiver` is `Derived` at execution time, so you might have expected the overload introduced in `Derived` to be called. However, the compile-time type of `receiver` is `Base`, and so the binder restricts the set of methods it considers to just the ones which *would* have been available if we'd been binding the method statically.

I take my hat off to the C# team for their attention to detail here - they've clearly worked very hard to make the behavior of execution-time binding match compile-time binding as closely as possible. That's assuming you *want* C# behavior of course... what about COM calls?

Delegating to the COM binder

In section 14.3.1 I sneakily mentioned "the COM binder" without explaining what I meant, because we hadn't covered "binders" in general. Now that we know what they do, how can code sometimes use the COM binder and sometimes use the C# binder? Doesn't the call site specify one or the other?

The answer is a nice use of composition. The C# compiler always generates a call site using the C# binder... but that asks the COM binder whether it's able to bind a call first. The COM binder will refuse to bind any non-COM calls, at which point the C# binder applies its own logic instead. That way we appear to get two binders for the price of one - and any other language can use the same form of piggy-backing to achieve consistent COM execution very easily.

Despite all of these decisions which have to be taken later, there are still some compile-time checks available, even for code which will be fully bound at execution time.

Compile-time errors for dynamic code

As I said near the start of this chapter, one of the disadvantages of dynamic typing is that some errors which would normally be detected by the compiler are delayed until execution time, at which point an exception is thrown. There are many situations where the compiler has to just hope you know what you're doing, but where it *can* help you, it will. The simplest example of this is when you try to call a method

with a statically typed receiver (or indeed a static method) and none of the overloads can possibly be valid, whatever type the dynamic value has at execution time. Listing 14.X shows three examples of invalid calls, two of which are caught by the compiler.

Example 14.20. Catching errors in dynamic calls at compile-time

```
string text = "cut me up";
dynamic guid = Guid.NewGuid();
text.Substring(guid);
text.Substring("x", guid);
text.Substring(guid, guid, guid);
```

Here we have three calls to `string.Substring`. The compiler knows the exact set of possible overloads, because it knows the type of `text` statically. It doesn't complain at the first call, because it can't tell what type `guid` will be - if it turns out to be an integer, all will be well. However, the final two lines throw up errors: there are no overloads which take a string as the first argument, and there are no overloads with three parameters. The compiler can *guarantee* that these would fail at execution time, so it's reasonable for it to fail at compile time instead.

A slightly trickier example is with type inference. If a dynamic value is used to infer a type argument in a call to a generic method, then the actual type argument won't be known until execution time and no validation can occur beforehand. However, any type argument which would be inferred without using *any* dynamic values can cause type inference to fail at compile-time. Listing 14.X shows an example of this

Example 14.21. Generic type inference with mixed static and dynamic values

```
void Execute<T>(T first, T second, string other) where T : struct
{
}
...
dynamic guid = Guid.NewGuid();
Execute(10, 0, guid);
Execute(10, false, guid);
Execute("hello", "hello", guid);
```

Again, the first call compiles, but would fail at execution time. The second call won't compile because `T` can't be both `int` and `bool`, and there are no conversions between the two of them. The third call won't compile because `T` is inferred to be `string`, which violates the constraint that it must be a value type.

That covers the most important points in terms of what the compiler *can* do for you. However, you can't use `dynamic` absolutely everywhere. There are limitations, some of which are painful, but most of which are quite obscure.

Restrictions on dynamic code

You can *mostly* use `dynamic` wherever you'd normally use a type name, and then write normal C#. However, there are a few exceptions. This isn't an exhaustive list, but it covers the cases you're most likely to run into.

Extension methods aren't resolved dynamically

The compiler emits *some* of the context of the call into the call site, as we've already seen: in particular, the site knows the static types that the compiler was aware of. However, it *doesn't* currently know which

using directives occurred in the source file containing the call. That means it doesn't know which extension methods are available at execution time.

This doesn't just mean you can't call extension methods *on* dynamic values - it means you can't pass them into extension methods as arguments either. There are two workarounds, however, both of which are helpfully suggested by the compiler. If you actually know which overload you want, you can cast the dynamic value to the right type within the method call. Otherwise, assuming you know which static class contains the extension method, you can just call it as a normal static method. Listing 14.X shows an example of a failing call and both workarounds.

Example 14.22. Calling extension methods with dynamic arguments

```
dynamic size = 5;
var numbers = Enumerable.Range(10, 10);
var error = numbers.Take(size);
var workaround1 = numbers.Take((int) size);
var workaround2 = Enumerable.Take(numbers, size);
```

Both of these approaches will work if you want to call the extension method with the dynamic value as the implicit `this` value, too - although the cast becomes pretty ugly in that case.

Delegate conversion restrictions with `dynamic`

The compiler has to know the exact delegate (or expression) type involved when converting a lambda expression, an anonymous method or a method group. You can't assign any of these to a plain `Delegate` or `object` variable without casting, and the same is true for `dynamic` too. However, a cast is enough to keep the compiler happy. This could be useful in some situations if you want to execute the delegate dynamically later. You can also use a delegate with a dynamic type as one of its parameters if that's useful. Listing 14.X shows some examples which will compile, and some which won't.

Example 14.23. Dynamic types and lambda expressions

```
dynamic badMethodGroup = Console.WriteLine;
dynamic goodMethodGroup = (Action<string>) Console.WriteLine;

dynamic badLambda = y => y + 1;
dynamic goodLambda = (Func<int, int>) (y => y + 1);

dynamic veryDynamic = (Func<dynamic, dynamic>) (d => d.SomeMethod());
```

Note that because of the way overload resolution works, this means you can't use lambda expressions in dynamically bound calls at all without casting - even if the only method which could possibly be invoked has a known delegate type at compile time. For example, this code will not compile:

```
void Method(Action<string> action, string value)
{
    action(value);
}
...
Method(x => Console.WriteLine(x), "error"); ❶
```

❶ Compile-time error

It's worth pointing out that all is not lost in terms of LINQ and `dynamic` interacting. You can have a strongly typed collection with an element type of `dynamic`, at which point you can still use extension

methods, lambda expressions and even query expressions. The collection can contain objects of different types, and they'll behave appropriately at execution time, as shown in listing 14.X.

Example 14.24. Querying a collection of dynamic elements

```
var list = new List<dynamic> { 50, 5m, 5d, 3 };
var query = from number in list
            where number > 4
            select (number / 20) * 10;

foreach (var item in query)
{
    Console.WriteLine(item);
}
```

This prints 20, 2.50, and 2.5. I deliberately divided by 20 and then multiplied by 10 to show the difference between decimal and double: the decimal type keeps track of precision without normalising, which is why 2.50 is displayed instead 2.5. The first value is an integer, so integer division is used, hence the value of 20 instead of 25.

Constructors and static methods

You can call constructors and methods dynamically in the sense that you can specify dynamic arguments, but you can't resolve a constructor or static method against a dynamic type. There's just no way of specifying which type you mean.

If you run into a situation where you *want* to be able to do this dynamically in some way, try to think of ways to use instance methods instead - for instance, by creating a factory type. You may well find that you can get the "dynamic" behavior you want using simple polymorphism or interfaces, but within static typing.

Type declarations and generic type parameters

You can't declare that a type has a base class of `dynamic`. You also can't use `dynamic` in a type parameter constraint, or as part of the set of interfaces that your type implements. You *can* use it as a type argument for a base class, or when you're specifying an interface for a variable declaration. So, for example, these declarations are invalid:

- `class BaseTypeOfDynamic : dynamic`
- `class DynamicTypeConstraint<T> where T : dynamic`
- `class DynamicTypeConstraint<T> where T : List<dynamic>`
- `class DynamicInterface : IEnumerable<dynamic>`

These are valid, however:

- `class GenericDynamicBaseClass : List<dynamic>`
- `IEnumerable<dynamic> variable;`

Most of these restrictions around generics are the result of the `dynamic` type not really existing as a .NET type. The CLR doesn't know about it - any uses in your code are translated into object with the `DynamicAttribute` applied appropriately. (For dynamic base types such as `List<dynamic>`, an

alternative constructor for `DynamicAttribute` is used to indicate which parts of the type declaration are dynamic.) All the dynamic behavior is achieved through compiler cleverness in deciding how the source code should be translated, and *library* cleverness at execution time. This equivalence between `dynamic` and `object` is evident in various places, but it's perhaps most obvious if you look at `typeof(dynamic)` and `typeof(object)`, which return the same reference. In general, if you find you can't do what you want to with the `dynamic` type, remember what it looks like to the CLR and see if that explains the problem. It may not suggest a solution, but at least you'll get better at predicting what will work ahead of time.

That's all the detail I'm going to give about how C# 4 treats `dynamic`, but there's another aspect of the dynamic typing picture which we really need to look at to get a well-rounded view of the topic: reacting dynamically. It's one thing to be able to *call* code dynamically, but it's another to be able to *respond* dynamically to those calls.

Implementing dynamic behavior

The C# language doesn't offer any specific help in implementing dynamic behavior, but the framework does. A type has to implement `IDynamicMetaObjectProvider` in order to react dynamically, but there are two built-in implementations which can take a lot of the work away in many cases. We'll look at both of these, as well as a *very* simple implementation of `IDynamicMetaObjectProvider`, just to show you what's involved. These three approaches are really very different, and we'll start with the simplest of them: `ExpandoObject`.

Using ExpandoObject

`System.Dynamic.ExpandoObject` looks like a funny beast at first glance. Its single public constructor has no parameters. It has no public methods, unless you count the explicit implementation of various interfaces - crucially `IDynamicMetaObjectProvider` and `IDictionary<string, object>`. (The other interfaces it implements are all due to `IDictionary<,>` extending other interfaces.) Oh, and it's sealed - so it's not a matter of deriving from it to implement useful behavior. No, `ExpandoObject` is *only* useful if you refer to it via `dynamic` or one of the interfaces it implements.

Setting and retrieving individual properties

The dictionary interface gives a hint as to its purpose - it's basically a way of storing objects via names. However, those names can also be used as properties via dynamic typing. Listing 14.X shows this working both ways⁶.

Example 14.25. Storing and retrieving values with ExpandoObject

```
dynamic expando = new ExpandoObject();
IDictionary<string, object> dictionary = expando;
expando.First = "value set dynamically";
Console.WriteLine(dictionary["First"]);

dictionary.Add("Second", "value set with dictionary");
Console.WriteLine(expando.Second);
```

Listing 14.X just uses strings as the values for convenience - you can use any object, as you'd expect with an `IDictionary<string, object>`. If you specify a delegate as the value, you can then call the delegate as if it were a method on the `expando`, as shown in listing 14.X.

⁶We should be able to set values with an indexer here - it's a bug in 4.0b1. I'll change the code when a version of .NET 4.0 is released that fixes it.

Example 14.26. Faking methods on an ExpandoObject with delegates

```
dynamic expando = new ExpandoObject();
expando.AddOne = (Func<int, int>) (x => x + 1);
Console.WriteLine(expando.AddOne(10));
```

Although this looks like a method access, you can also think of it as a property access which returns a delegate, and then an invocation of the delegate. If you created a statically typed class with an `AddOne` property of type `Func<int, int>` you could use exactly the same syntax. The C# generated to call `AddOne` does in fact use "invoke member" rather than trying to access it as a property and then invoke it, but `ExpandoObject` knows what to do. You can still access the property to retrieve the delegate if you want to though.

Let's move on to a slightly larger example - although we're still not going to do anything particularly tricky.

Creating a DOM tree

We're going to create a tree of expandos which mirrors an XML DOM tree. This is a pretty crude implementation, designed for simplicity of demonstration rather than real world use. In particular, it's going to assume we don't have any XML namespaces to worry about. Each node in the tree has two name/value pairs which will always be present: `XElement`, which stores the original LINQ to XML element used to create the node, and `ToXml`, which stores a delegate which just returns the node as an XML string. You could just call `node.XElement.ToString()`, but this way gives another example of how delegates work with `ExpandoObject`. One point to mention is that I used `ToXml` instead of `ToString`, as setting the `ToString` property on an expando *doesn't* override the normal `ToString` method. This could lead to very confusing bugs, so I opted for the different name instead.

The interesting part isn't the fixed names though, it's the ones which depend on the real XML. I'm going to ignore attributes completely, but any *elements* in the original XML which are children of the original element are accessible via properties of the same name. For instance, consider the following XML:

```
<root>
  <branch>
    <leaf />
  </branch>
</root>
```

Assuming a dynamic variable called `root` representing the `Root` element, we could access the leaf node with two simple property accesses, which can occur in a single statement:

```
dynamic leaf = root.branch.leaf;
```

If an element occurs more than once within a parent, the property just refers to the first element with that name. To make the other elements accessible, each element will also be exposed via a property using the element name with a suffix of "List" which returns a `List<dynamic>` containing each of the elements with that name in document order. In other words, the above access could also be represented as `root.branchList[0].leaf`, or perhaps `root.branchList[0].leafList[0]`. Note that the indexer here is being applied to the list - you can't define your own indexer behavior for expandos. The implementation of all of this is actually remarkably simple, with a single recursive method doing all the work, as shown in listing 14.X.

Example 14.27. Implementing a simplistic XML DOM conversion with ExpandoObject

```

public static dynamic CreateDynamicXml(XElement element)
{
    dynamic expando = new ExpandoObject();
    expando.XElement = element;❶
    expando.ToXml = (Func<string>)element.ToString;❷

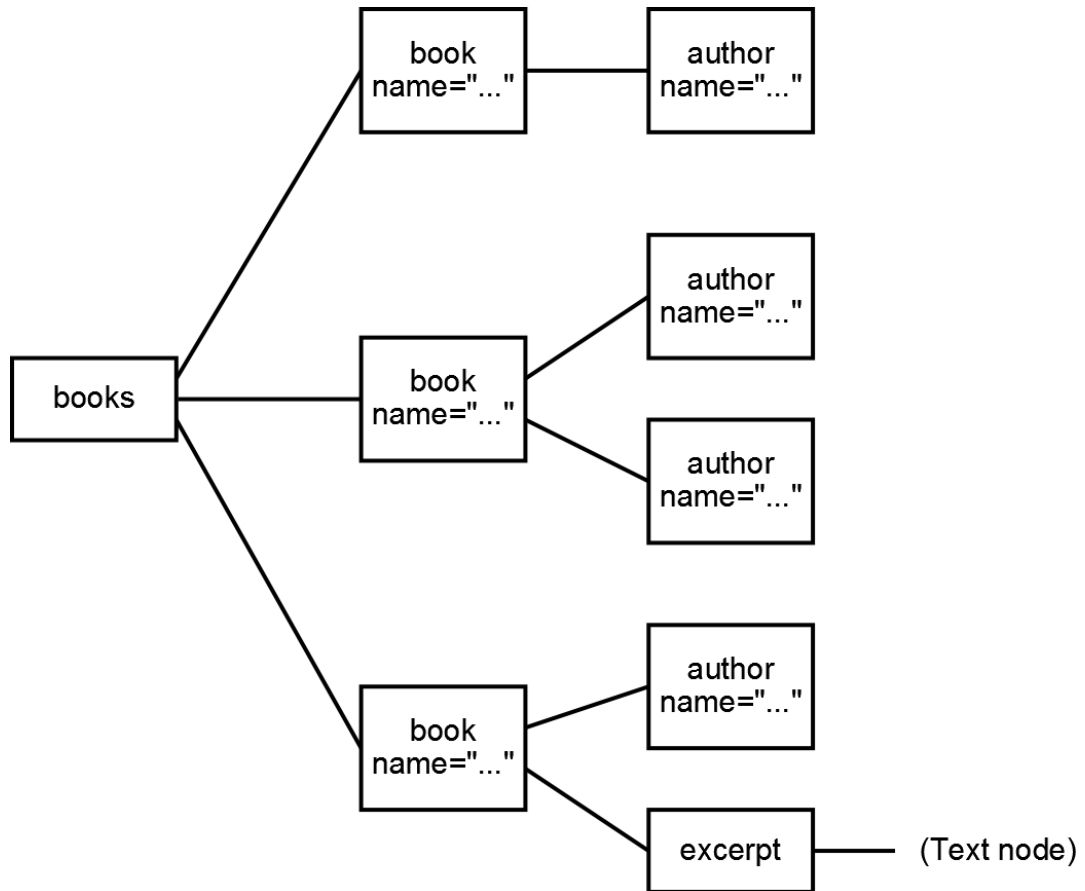
    IDictionary<string, object> dictionary = expando;
    foreach (XElement subElement in element.Elements())
    {
        dynamic subNode = CreateDynamicXml(subElement);❸
        string name = subElement.Name.LocalName;
        string listName = name + "List";
        if (dictionary.ContainsKey(name))
        {
            ((List<dynamic>) dictionary[listName]).Add(subNode);❹
        }
        else
        {
            dictionary.Add(name, (object) subNode);❺
            dictionary.Add(listName, new List<dynamic> { subNode });
        }
    }
    return expando;
}

```

- ❶ Assigns a simple property
- ❷ Converts a method group to delegate to use as property
- ❸ Recursively processes sub-element
- ❹ Adds repeated element to list
- ❺ Creates new list and sets properties

Without the list handling, listing 14.X would have been even simpler. We set the `XElement` and `ToXml` properties dynamically (❶ and ❷), but we can't do that for the elements or their lists, because we don't know the names at compile time. We use the dictionary representation instead (❸ and ❹), which also allows us to check for repeated elements easily. You can't tell whether or not an expando contains a value for a particular key just by accessing it as a property: any attempt to access a property which hasn't already been defined results in an exception. The recursive handling of sub-elements is as straightforward in dynamic code as it would be in statically typed code: we just call the method recursively ❺ with each sub-element, using its result to populate the appropriate properties.

We're going to need some XML to use as an example, but it's helpful to picture it graphically as well as in its raw format. We'll use a very simple structure representing books. Each book has a single name represented as an attribute, and may have multiple authors, each with their own element. Figure 14.X shows the whole file as a tree, and the text appears below.

Figure 14.4. Tree structure of sample XML file

```

<books>
  <book name="Mortal Engines">
    <author name="Philip Reeve" />
  </book>
  <book name="The Talisman">
    <author name="Stephen King" />
    <author name="Peter Straub" />
  </book>
  <book name="Rose">
    <author name="Holly Webb" />
    <excerpt>
      Rose was remembering the illustrations from
      Morally Instructive Tales for the Nursery.
    </excerpt>
  </book>
</books>

```

Listing 14.X shows a brief example of how the expando code can be used with this XML document, including the `ToXml` and `XElement` properties. The `books.xml` file contains the XML tree shown in the figure.

Example 14.28. Using a dynamic DOM created from expandos

```
XDocument doc = XDocument.Load("books.xml");
dynamic root = CreateDynamicXml(doc.Root);
Console.WriteLine(root.book.author.ToXml());
Console.WriteLine(root.bookList[2].excerpt.XElement.Value);
```

Listing 14.X should hold no surprises, unless you're unfamiliar with the `XElement.Value` property which simply returns the text within an element. The output of the listing is as we'd expect:

```
<author name="Philip Reeve" />
Rose was remembering the illustrations from
Morally Instructive Tales for the Nursery.
```

This is all very well, but there are a few issues with our DOM. In particular:

- It doesn't handle attributes at all
- We need two properties for each element name, due to the need to represent lists
- It would be nice to override `ToString()` instead of adding an extra property
- The result is mutable - there's nothing to stop code from adding its own properties afterwards
- Although the expando is mutable, it won't reflect any changes to the underlying `XElement` (which is also mutable)

Fixing these issues requires more control than just being able to set properties. Enter `DynamicObject`...

Using DynamicObject

`DynamicObject` is a more powerful way of interacting with the DLR than using `ExpandoObject`, but it's a lot simpler than implementing `IDynamicMetaObjectProvider`. Although it's not *actually* an abstract class, you really need to derive from it to do anything useful - and the only constructor is protected, so it might as well be abstract for all practical purposes. There are four kinds of method which you might wish to override:

- `TryXXX()` invocation methods, representing dynamic calls to the object
- `GetDynamicMemberNames()`, which can return an list of the available members
- The normal `Equals()/GetHashCode()/ToString()` methods which can be overridden as usual
- `GetMetaObject()` which returns the meta-object used by the DLR

We'll look at all but the last of these to improve our XML DOM representation, and we'll discuss meta-objects in the next section when we implement `IDynamicMetaObjectProvider`. In addition, it can be very useful to create new members in your derived type, even if callers are likely to use any instances as dynamic values anyway. Before we take any of these steps, we'll need a class to add all the code to.

Getting started

As we're deriving from `DynamicObject` instead of just calling methods on it, we need to start with a class declaration. Listing 14.X shows the basic skeleton that we'll be fleshing out.

Example 14.29. Skeleton of DynamicXElement

```
public sealed class DynamicXElement : DynamicObject
{
    private readonly XElement element; ❶

    private DynamicXElement(XElement element) ❷
    {
        this.element = element;
    }

    public static dynamic CreateInstance(XElement element) ❸
    {
        return new DynamicXElement(element);
    }
}
```

- ❶❶ XElement this instance wraps
- ❷❶ Private constructor prevents direct instantiation
- ❸❶ Public method to create instances

The `DynamicXElement` class just wraps an `XElement` ❶. This will be all the state we have, which is a significant design decision in itself. When we created an `ExpandoObject` earlier, we recursed into its structure and populated a whole mirrored tree. We really had to do that, because we couldn't intercept property accesses with custom code later on. Obviously this is more expensive than the `DynamicXElement` approach, where we will only ever wrap the elements of the tree we actually have to. Additionally, it means that any changes to the `XElement` after we've created the `expando` are effectively lost: if you add more sub-elements, for example, they won't appear as properties because they weren't present when we took the snapshot. The lightweight wrapping approach is always "live" - any changes you make in the tree will be visible through the wrapper.

The *disadvantage* of this is that we no longer provide the same idea of identity that we had before. With the `expando`, the expression `root.book.author` would evaluate to the same reference if we used it twice. Using `DynamicXElement`, each time the expression is evaluated it will create new instances to wrap the sub-elements. We *could* implement some sort of smart caching to get around this, but it could end up getting very complicated very quickly.

I've chosen to make the constructor of `DynamicXElement` private ❶ and instead provide a public static method to create instances ❷. The method has a return type of `dynamic`, because that's how we expect developers to use the class. A slight alternative would have been to create a separate public static class with an extension method to `XElement`, and keep `DynamicXElement` itself internal. The class itself is an implementation detail: there's not a lot of point in using it unless you're working dynamically.

With our skeleton in place, we can start adding features. We'll start with really simple stuff: adding methods and indexers as if this were just a normal class.

DynamicObject support for simple members

When we created our `expando`, there were two members we *always* added: the `ToXml` "method" and the `XElement` property. This time we don't need a new method to convert the object to a string representation: we can override the normal `ToString()` method. We can also provide the `XElement` property as if we were writing any other class. One of the nice things about `DynamicObject` is that when you don't need truly dynamic behavior, you don't have to implement it. The meta-object used to resolve calls uses any of the `TryXXX` methods, it checks to see whether the member already exists as a straightforward CLR member. If it does, that member will be called. This makes life significantly simpler.

We're going to have two indexers in `DynamicXElement` as well, to provide access to attributes and replace our element lists. Listing 14.X shows the new code to be added to the class.

Example 14.30. Adding non-dynamic members to `DynamicXElement`

```
public override string ToString() ❶
{
    return element.ToString();
}

public XElement XElement ❷
{
    get { return element; }
}

public XAttribute this[XName name] ❸
{
    get { return element.Attribute(name); }
}

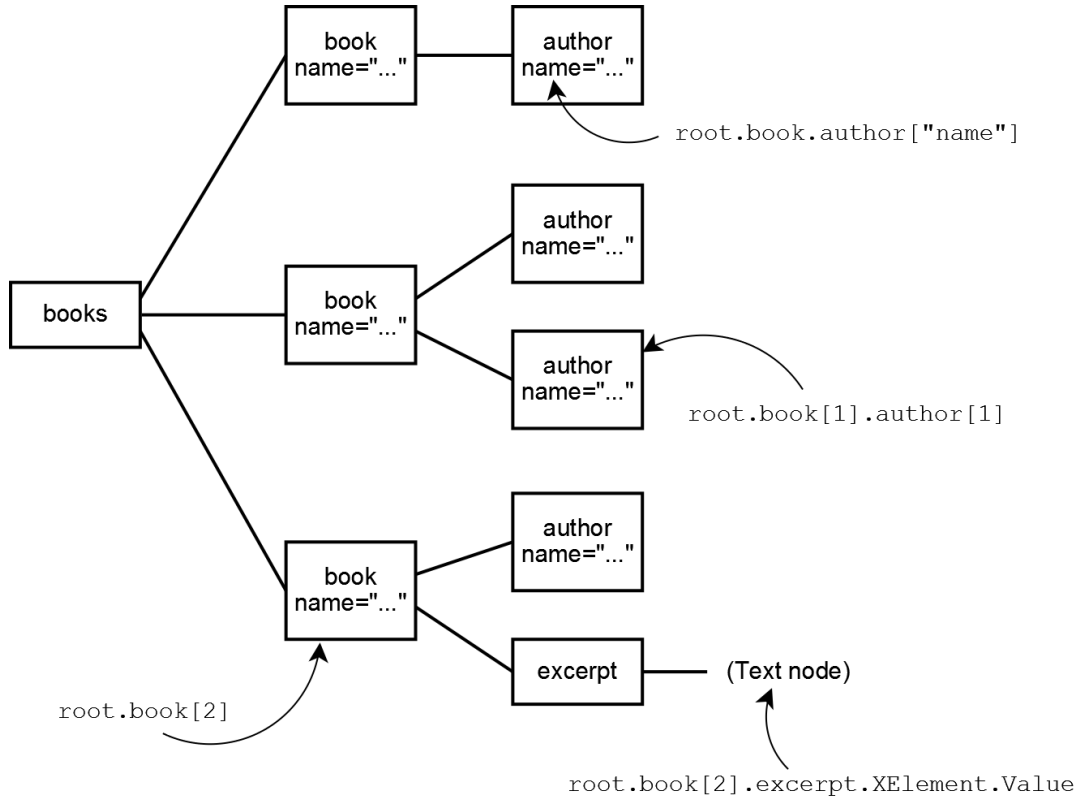
public dynamic this[int index] ❹
{
    get
    {
        XElement parent = element.Parent;
        if (parent == null) ❺
        {
            if (index != 0)
            {
                throw new ArgumentOutOfRangeException();
            }
            return this;
        }
        XElement sibling = parent.Elements(element.Name) ❻
                                .ElementAt(index);
        return element == sibling ? this
                                : new DynamicXElement(sibling);
    }
}
```

- ❶❶ Overrides `ToString()` as normal
- ❷❷ Returns wrapped element
- ❸❸ Indexer retrieving an attribute
- ❹❹ Indexer retrieving a sibling element
- ❺❶ Is this a root element?
- ❻❷ Find appropriate sibling

There's a fair amount of code in listing 14.X, but most of it is very straightforward. We override `ToString()` ❶ by just proxying the call to the `XElement`, and if we wanted to implement value equality we could do something similar for `Equals()` and `GetHashCode()`. The property returning the underlying element ❷ and the indexer for attributes ❸ are also very simple, although it's worth noting that we only need to use an `XName` for the parameter to the attribute indexer: if you provide a string at execution time, `DynamicObject` will take care of calling the implicit conversion to `XName` for you.

The trickiest part of the code is understanding what the indexer with the `int` parameter ❶ is meant to be doing. It's probably easiest to explain this in terms of expected usage. The idea is to avoid having the extra "list" property by making an element act as both a single element and a list of elements. Figure 14.X shows our sample XML with a few expressions to reach different nodes within it.

Figure 14.5. Selecting data using `DynamicXElement`



Once you understand what the indexer is meant to do, the implementation is fairly simple, complicated only by the possibility that we could already be at the top of the tree ❶. Otherwise we just have to ask the element for all its siblings, then pick the one we've been asked for ❷.

So far we haven't done anything dynamic except in terms of the return type of `CreateInstance()` - none of our examples will work, because we haven't written the code to fetch sub-elements. Let's fix that now.

Overriding `TryXXX` methods

In `DynamicObject`, you respond to calls dynamically by overriding one of the `TryXXX` methods. There are 12 of them, representing different types of operation, as shown in table 14.X.

Table 14.1. Virtual `TryXXX` methods in `DynamicObject`

Name	Type of call represented (where <code>x</code> is the dynamic object)
<code>TryBinaryOperation</code>	Binary operation such as <code>x + y</code>
<code>TryConvert</code>	Conversions such as <code>(Target) x</code>

Name	Type of call represented (where <i>x</i> is the dynamic object)
TryCreateInstance	Object creation expressions: no equivalent in C#
TryDeleteIndex	Indexer removal operation: no equivalent in C#
TryDeleteMember	Property removal operation: no equivalent in C#
TryGetIndex	Indexer "getter" such as <i>x</i> [10]
TryGetMember	Property "getter" such as <i>x</i> .Property
TryInvoke	Direct invocation effectively treating <i>x</i> like a delegate, such as <i>x</i> (10)
TryInvokeMember	Invocation of a member, such as <i>x</i> .Method()
TrySetIndex	Indexer "setter" such as <i>x</i> [10] = 20
TrySetMember	Property setter, such as <i>x</i> .Property = 10
TryUnaryOperation	Unary operation such as ! <i>x</i> or - <i>x</i>

Each of these methods has a Boolean return type to indicate whether or not the binding was successful. Each takes an appropriate binder as the first parameter, and if the operation logically has arguments (for instance the arguments to a method, or the indexes for an indexer) these are represented as an `object[]`. Finally, if the operation might have a return value (which includes everything except the set and delete operations) then there's an out parameter of type `object` to capture that value. The exact type of the binder depends on the operation: there's a different binder type for each of the operations. For example, the full signature of `TryInvokeMember` is:

```
public virtual bool TryInvokeMember(InvokeMemberBinder binder,
    object[] args, out object result)
```

You only need to override the methods representing operations you support dynamically. In our case, we have dynamic read-only properties (for the elements) so we need to override `TryGetMember()`, as shown in listing 14.X.

Example 14.31. Implementing a dynamic property with `TryGetMember()`

```
public override bool TryGetMember(GetMemberBinder binder,
    out object result)
{
    XElement subElement = element.Element(binder.Name); ❶
    if (subElement != null)
    {
        result = new DynamicXElement((XElement)subElement); ❷
        return true;
    }
    return base.TryGetMember(binder, out result); ❸
}
```

- ❶ Find the first matching sub-element
- ❷ If found, build a new dynamic element
- ❸ Otherwise use the base implementation

The implementation in listing 14.X is quite simple. The binder contains the name of the property which was requested, so we look for the appropriate sub-element in the tree ❶. If there is one, we create a new `DynamicXElement` with it, assign that to the output parameter `result`, and return `true` to indicate

that the call was bound successfully ❷. If there was no sub-element with the right name, we just call the base implementation of `TryGetMember()` ❸. The base implementation of each of the `TryXXX` methods just returns `false` and sets the output parameter to `null` if there is one. We could easily have done this explicitly, but we'd have had two separate statements: one to set the output parameter and one to return `false`. If you prefer the slightly longer code, there's absolutely no reason not to write it - the base implementations are just slightly convenient in terms of doing everything required to indicate that the binding failed.

There's one bit of complexity I've side-stepped: the binder has another property (`IgnoreCase`) which indicates whether or not the property should be bound in a case-insensitive way. For example, Visual Basic is case-insensitive, so its binder implementation would return `true` for this property, whereas C#'s would return `false`. In our situation, it's slightly awkward. Not only would it be more work for `TryGetMember` to find the element in a case-insensitive manner ("more work" is always unpleasant, but it's not a good reason not to implement it) - there's the more philosophical problem of what happens when you then use the indexer to select siblings. Should the object remember whether it's case-sensitive or not, and select siblings in the same way later on? If so, you'd see different results for `element.Name[2]` depending on the language. If, on the other hand, the indexer is always case-sensitive, then `element.name[0]` might not even find itself! This sort of impedance mismatch is likely to happen in similar situations. If you aim for perfection, you're likely to tie yourself up in knots. Instead, aim for a practical solution that you're confident you can implement and maintain, and then document the restrictions.

With all this in place, we can test `DynamicXElement` as shown in listing 14.X.

Example 14.32. Testing `DynamicXElement`

```
XDocument doc = XDocument.Load("books.xml");
dynamic root = CreateDynamicXml(element.Root);
Console.WriteLine(root.book["name"]);
Console.WriteLine(root.book[2].author[1]);
```

We could add more complexity to our class, of course. We could add a `Parent` property to go back up the tree, or we might want to change to access sub-elements using method calls and make property access represent attributes. The principle would be exactly the same: where you know the name in advance, implement it as a normal class member. If you need it to be dynamic, override the appropriate `DynamicObject` method.

There's one more piece of polish to apply to `DynamicXElement` before we leave it though. It's time to advertise what we've got to offer.

Overriding `GetDynamicMemberNames`

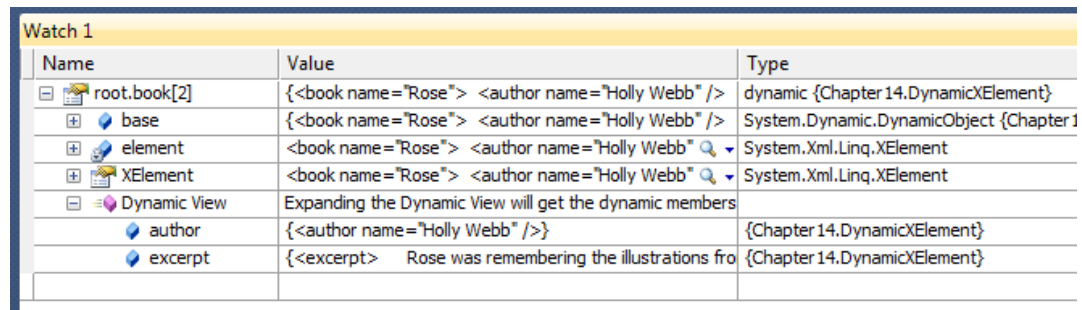
Some languages, such as Python, allow an object to publish what names it knows about; it's the `dir` function in Python, if you're interested. This information is useful in a REPL environment, and it can also be handy when you're debugging in an IDE. The DLR makes this information available through the `GetDynamicMemberNames()` method of both `DynamicObject` and `DynamicMetaObject` (we'll meet the latter in a minute). All we have to do is override this method, provide a sequence of the dynamic member names, and we make our object's properties more discoverable. Listing 14.X shows the implementation for `DynamicXElement`.

Example 14.33. Implementing `GetDynamicMemberNames` in `DynamicXElement`

```
public override IEnumerable<string> GetDynamicMemberNames()
{
    return element.Elements()
        .Select(x => x.Name.LocalName)
        .Distinct()
        .OrderBy(x => x);
}
```

As you can see, all we need is a simple LINQ query. Of course that won't *always* be the case, but I suspect many dynamic implementations will be able to use LINQ in this way. In this case we need to make sure that we don't return the same value more than once if there's more than one element with any particular name, and I've sorted the results just for consistency. In the Visual Studio 2010 debugger, you can expand the "Dynamic View" of a dynamic object and see the property names and values, as shown in figure 14.X.

Figure 14.6. Visual Studio 2010 displaying dynamic properties of a `DynamicXElement`



Name	Value	Type
root.book[2]	{<book name="Rose"> <author name="Holly Webb" />}	dynamic {Chapter14.DynamicXElement}
base	{<book name="Rose"> <author name="Holly Webb" />}	System.Dynamic.DynamicObject {Chapter14.DynamicXElement}
element	<book name="Rose"> <author name="Holly Webb" />	System.Xml.Linq.XElement
XElement	<book name="Rose"> <author name="Holly Webb" />	System.Xml.Linq.XElement
Dynamic View	Expanding the Dynamic View will get the dynamic members	
author	{<author name="Holly Webb" />}	{Chapter14.DynamicXElement}
excerpt	{<excerpt> Rose was remembering the illustrations fro	{Chapter14.DynamicXElement}

Unfortunately the dynamic view just calls `ToString()` on each of the values; there's no way of drilling down further. *FIXME: Check this against later betas!*

We've now finished our `DynamicXElement` class, as far as we're going to take it in this book. I believe that `DynamicObject` hits a sweet spot between control and simplicity: it's fairly easy to get it right, but it has far fewer restrictions than `ExpandoObject`. However, if you really need total control over binding, you'll need to implement `IDynamicMetaObjectProvider` directly.

Implementing `IDynamicMetaObjectProvider`

FIXME: MEAP readers, I need your help! `IDynamicMetaObjectProvider` is all very well, but I can't currently think of a good example which uses it in anything other than a very contrived way. I will keep thinking, but if you have any ideas of what you'd like to see in this section, please post them in the forum.

Summary

It feels like we've come a very long way from mainstream, statically typed C#. We've looked at some situations where dynamic typing can be useful, how C# 4 makes it possible (both in terms of the code you write and how it works under the surface) and how to respond dynamically to calls. Along the way, we've seen a bit of COM, a bit of Python, some reflection, and learned a little about the Dynamic Language Runtime.

This has *not* been a complete guide to how the DLR works, or even how C# operates with it. The truth is, this is a deep topic with many dark corners. In reality, you're unlikely to bump into the problems - and most developers won't even use the simple scenarios very often. I'm sure whole books will be written about the DLR, but I hope I've given enough detail here to let 99% of C# developers get on with their jobs without needing any more information. If you want to know more, the documentation on the DLR web site [<http://dlr.codeplex.com/Wiki/View.aspx?title=Docs%20and%20specs>] is a good starting point.

If you never use the `dynamic` type, you can pretty much ignore dynamic typing entirely. I recommend that that's exactly what you do for the majority of your code - in particular, I wouldn't use it as a crutch to avoid creating appropriate interfaces, base classes and so on. Where you *do* need dynamic typing, I'd use it as sparingly as possible: don't take the attitude of "I'm using `dynamic` in this method, so I might as well just make *everything* dynamic."

I don't want to sound too negative, however. If you find yourself in a situation where dynamic typing is helpful, I'm sure you'll be very thankful that it's present in C# 4. Even if you never need it for production code, I'd encourage you to give it a try for the fun of it - I've found it fascinating to delve into. You may also find the DLR useful without really using dynamic typing: most of our Python example didn't use any dynamic typing, but it used the DLR to execute the Python script containing the configuration data.

Between this chapter and the previous one, we've now covered all the new features of C# as a language. However, part of the aim of this book is to help developers evolve their ideas of idiomatic C#. Two of the new technologies introduced into .NET 4.0 have the potential to change the way we write code in terms of robustness and concurrency, just as LINQ has changed our perspective on working with collections. In the next chapter, we'll look at the Code Contracts and Parallel Extensions libraries.