SAMPLE CHAPTER

Metaprogramming in .NET

Kevin Hazzard
Jason Bock

FOREWORD BY Rockford Lhotka

MANNING

Metaprogramming in .NET

by Kevin Hazzard
Jason Bock

Chapter 1

brief contents

1 Metaprogramming concepts

In this chapter

- Defining metaprogramming
- Exploring examples of metaprogramming

The basic principles of object-oriented programming (OOP) are understood by most software developers these days. For example, you probably understand how encapsulation and implementation-hiding can increase the cohesion of classes. Languages like C# and Visual Basic are excellent for creating so-called coarsely grained types because they expose simple features for grouping and hiding both code and data. You can use cohesive types to raise the abstraction level across a system, which allows for loose coupling to occur. Systems that enjoy loose coupling at the top level are much easier to maintain because each subsystem isn't as dependent on the others as they could be in a poor design. Those benefits are realized at the lower levels, too, typically through lowered complexity and greater reusability of classes. In figure 1.1, which of the two systems depicted would likely be easier to modify?

Without knowing what the gray circles represent, most developers would pick the diagram on the right as the better one. This isn't even a developer skill. Show the diagrams to an accountant and she'll also choose the one on the right as the less

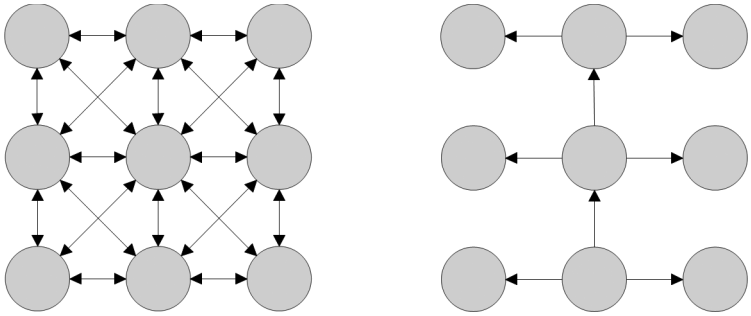

Figure 1.1 Which system is easier to change?

complex. We recognize simplicity when we see it. Our challenge as programmers is in seeing the opportunities for simplicity in the systems we develop. Language features like encapsulation, abstraction, inheritance, data-hiding, and polymorphism are great, but they only take you part of the way there.

The I in SOLID

Along the way, we'll refer to some of the five SOLID (single responsibility, open-closed, Liskov substitution, interface segregation, and dependency inversion) principles of object-oriented design (OOD). While we're thinking about coupling and cohesion, it's a good time to discuss the "I" in SOLID—the interface segregation principle (ISP). The ISP says that many client-specific interfaces are better than one general-purpose interface.

This seems to contradict the notion that high cohesion is always a good thing. If you study the ISP along with the other four SOLID principles, though, you'll discover that it speaks to the correctness of the middle ground in software development. The diagram on the left in figure 1.1 may represent absurdly tight coupling and low cohesion. The one on the right may embody the other extreme. The ISP tells us that there may be an unseen middle design that's best of all.

The metaprogramming style of software development shares many of the goals of traditional OOP. Metaprogramming is all about making software simpler and reusable. But rather than depending strictly on language features to reduce code complexity or increase reusability, metaprogramming achieves those goals through a variety of libraries and coding techniques. There are language-specific features that make metaprogramming easier in some circumstances. For the most part, however, metaprogramming is a set of language-independent skills. We use C# for most of the examples in this book, but don't be surprised when we toss in a bit of JavaScript or F# here and there when it helps to teach an idea at hand.

If you know a little bit about metaprogramming, you may scoff at the idea that metaprogramming reduces complexity. It's true that some types of metaprogramming require a deeper understanding of tools that may be out-of-sight from your per-

spective today. You may have been told in the past that to do metaprogramming, you must understand how compilers work. Many years ago that was largely true, but today you can learn and use highly effective metaprogramming techniques without having to know much at all about compilers. After all, complexity is in the eye of the beholder, as the saying goes. As perceived complexity from the end user's standpoint goes down, internal complexity of the design often goes up. Complexity reduction when metaprogramming follows the same rules. To achieve simplicity on the outside, the code on the inside of a metaprogramming-enabled component typically takes on added responsibilities.

For example, so-called Domain-Specific Languages (DSLs) are often built with metaprogramming tools and techniques. DSLs are important because they can fundamentally change the way that a company produces intellectual property (IP). When a DSL enables a company to shift some of its IP development from traditional programmers to analysts, time to market can be dramatically reduced. Well-designed DSLs can also increase the comprehension of business rules across the enterprise, allowing people into the game from other roles that have been traditionally unable to participate in the process. A flowcharting tool that produces executable code is a good example of such a DSL because it enables business stakeholders to describe their intent in their own vocabulary.

The trade-off is that DSLs are notoriously difficult to design, write, test, and support. Some argue that DSLs are much too complex and not worth the trouble. But from the consumer's vantage point, DSLs are precious to the businesses they serve precisely because they lower perceived complexity. In the end, isn't that what we do for a living? We make difficult business problems seem simple. As you study metaprogramming throughout this book, keep that thought in mind.

> **NOTE** *DSLs in Action* by Debasish Ghosh (www.manning.com/ghosh) and *DSLs in Boo* by Oren Eini writing as Ayende Rahien (www.manning.com/rahien) are both excellent choices if your goal is to learn how to create full-featured DSLs.

At times, you may struggle as you try to learn so many new things at once. There will be enough promise in each new thing you learn to prove that the struggle is worthwhile. In the end, you'll have many new tools for fighting software complexity and for writing reusable code. As you begin to put metaprogramming to work in your projects, others will study what you've done. They'll marvel at the kung fu of your metaprogramming skills. Soon they'll begin to emulate you, and, as they say, imitation is the sincerest form of flattery.

Let's begin by defining what metaprogramming is. Then we'll dive into a few interesting examples to show how it's used.

1.1 Definitions of metaprogramming

The classic definition for a metaprogram is "a computer program that writes new computer programs." This sounds a lot like the definition of a compiler. A compiler for a programming language like C# could be thought of as the ultimate metaprogram, because its only job is to produce other programs from source code. But to call the C# compiler a metaprogram is a stretch. Unstated in the definition of a traditional compiler is the idea that the execution step is fixed in time, and the existence of the compiled outputs are somewhat unseen by end users. Also, metaprogramming techniques are clearly different because they're almost always used to deal with some sort of ever-changing stimulus.

There may be semistructured documents that need parsing on the fly. You may need a way to express trading restrictions from your partners that change daily. Database schemas change from time to time, and you may need a way to make your programs adapt gracefully. All of these problems are perfect for metaprogram-based solutions. They don't require compilers in the traditional sense. They do require the flexibility that a compiler affords to adapt to situations at hand.

The C# compiler in its current form is almost always invoked by programmers during a build process to produce a new program. In the near future, that will be changing with the release of Microsoft's Roslyn (code name) tools. Roslyn opens the black box of the C# and VB compilers to make them available before, during, and after the deployment of your applications. When that happens, we expect to see Microsoft's compilers used in many metaprogramming scenarios.

> **DEFINITION** *Metaprogramming* may be among the most misunderstood terms in computer jargon. It's certainly one of the more difficult to define. To make learning about it easier, each time you see the word *metaprogramming* in this book, try to think of it as *after*-programming or *beside*-programming. The Greek prefix *meta* allows for both of those definitions to be correct. Most of the examples in this book demonstrate programming after traditional compilation has occurred, or by using dynamic code that runs alongside other processes. For each example, ask yourself which kind of metaprogramming you're observing. Some of the more in-depth examples demonstrate both kinds simultaneously.

Also inherent in the classic definition of metaprogramming is the notion that the code-generation process is embedded within an application to perform some type of dynamic processing logic. The word *dynamic* gets tossed around a lot in discussions about metaprogramming because it's often used to add adaptive interfaces to a program at runtime. For example, a dynamic XML application might read XML Schema Definitions (XSD) at runtime to construct and compile high-performance XML parsers that can be used right away or saved for future use. Such an application would perform well and be highly adaptable to new types of XML without the need for recompilation.

Another common definition for metaprogramming is "a computer program that manipulates other programs at runtime." Scripting languages often fit this mold, providing the simple but powerful tools for doing metaprogramming. A program that manipulates another program doesn't have to be a scripting language. The `dynamic` keyword in C# can be used to emit a kind of manipulating code into a compiled application, like this:

```
dynamic document = DocumentFactory.Create();
document.Open();
```

Using the `dynamic` keyword, the call to the `Open()` method shown here is embedded into a bit of C# code known as a `CallSite`. We dive into `CallSites` in great detail later in chapter 8. For now, all you need to understand is that what appears to be a type-safe call to the `Open()` method in the document object is implemented through C#'s runtime binder using the literal string `"Open."` When you dig around in the Intermediate Language (IL) emitted by the compiler for the preceding snippet, you may be surprised to see the literal string `"Open"` passed to the binder to invoke the method. The C# code certainly didn't look like a scripting language, but what was emitted certainly has that flavor. Through the various runtime binders for interfacing with plain old CLR objects (POCO), Python scripts, Ruby scripts, and COM objects, C# `CallSites` exhibit the second definition of metaprogramming rather well. In chapter 8, we show you how to interface with all those languages and object types using C# dynamic typing.

Writing new programs at runtime and manipulating programs at runtime aren't mutually exclusive concepts. Many of the metaprogramming examples you'll encounter in this book do both. First, they may use some sort of code-generation technique to create and compile code on the fly to adapt to some emerging set of circumstances. Next, they may control, monitor, or invoke those same programs to achieve the desired outcome.

More metaprogramming jargon

There are a few more terms that you may encounter when you start reading articles and other books on metaprogramming. You may run across the term *metalanguage* to refer to the language used in the original program (the one that's writing the others). We prefer the term *metaprogram* because it's more generic. Remember that metaprogramming is largely a language-independent craft.

Other terms you're likely to hear are *target language* or *object language*, referring to the code produced by the metaprogram. Both those terms imply that there's an intermediate language that the metaprogrammer cares about in the process. As you'll soon discover, the output of a .NET metaprogram could be Common Intermediate Language (CIL) which, for all intents and purposes, you can regard as native code. In those cases, there's no target language in the classical sense.

1.2 Examples of metaprogramming

For most people, the best way to learn is by example. Let's examine a few examples of metaprogramming in action. We begin with the simplest of metaprogramming concepts: invoking bits of dynamically supplied JavaScript at runtime. This prototype will give you an appreciation for the flexibility that metaprogramming can add to a web application, even though the example is contrived for simplicity.

Next, we look at how to use introspective interfaces to drive application behavior at runtime. Through it you'll learn how to do simple reflection to peer into objects at runtime. But the real purpose of that example is to help you understand the performance considerations when deciding to metaprogramming-enable an interface to make it friendlier and more adaptive at runtime.

The third example in this section concerns code generation, arguably the classic definition of metaprogramming. We show you two runtime types of code generation: creating source code from a so-called object graph assembled by hand and creating executable IL from a lambda expression. For the second type, we let the C# compiler do the heavy lifting first. Then we build the lambda expressions by hand before turning them into runnable code.

The last example in this section demonstrates how you can use the dynamic features of the C# 4 compiler to do some fairly interesting metaprogramming with little effort. You'll learn a little bit about how the `CallSite` and `CSharpRuntimeBinder` types work. The real goal of that example, though, is to highlight some of the best practices around using dynamic types in C#.

The examples here are designed to provide basic prototypes that you'll need to learn faster when reading future chapters. Also, by examining several simple approaches to metaprogramming in rapid succession, we hope to give you a more holistic view of this important programming paradigm.

1.2.1 Metaprogramming via scripting

There are many dynamic programming languages. Some of them are also considered to be scripting languages. Languages like Python or Ruby would work well for our first example because they have clean, easy-to-understand syntaxes and they're loaded with great metaprogramming capabilities. But rather than starting with one of those languages, which could steepen the learning curve if you don't know them, let's begin, in the following listing, with the two most popular languages in the world.

Listing 1.1 Dynamic Number Conversion (HTML and JavaScript)

```
<!DOCTYPE html>
<html>
     <head>
       <script type="text/javascript">
       function convert() {
         var fromValue = eval(fromVal.value);
         toVal.innerHTML = eval(formula.value).toString();
       }
```

```
      </script>
    </head>
    <body>
      <span>fromValue:</span> 
      <input id="fromVal" type="text"/><br/>

      <span>formula:</span> 
      <input id="formula" type="text"/><br/>

      <input type="button" onclick="javascript:convert();"
        value="Convert" /><br/>

      <span>toValue:</span> <span id="toVal"></span>
    </body>
</html>
```

The admittedly unattractive web page created by this markup demonstrates a core metaprogramming concept. After locating the DynamicConversion.htm file in the book's sample source code, load it up and enter some values into the `fromValue` and formula fields, as shown in figure 1.2. Be sure to use the token `fromValue` somewhere in the formula to refer to the numeric value that you type into the `fromValue` field.

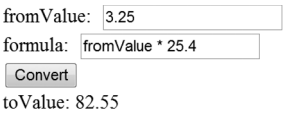

Figure 1.2 DynamicConversion.htm—converting inches to millimeters

Figure 1.2 shows a calculation that multiplies the user-supplied `fromValue` by 25.4, which is the simple formula for converting inches to millimeters. Typing in a `fromValue` such as `3.25` and clicking Convert shows that 3.25 inches is equivalent to 82.55 millimeters. There are two bits of JavaScript code in this web page that make it work: a function called `convert()` and the `onclick` event handler for the Convert button, which invokes the `convert()` function when the button is clicked. In the `convert()` function, the HTML Document Object Model (DOM) is used to fetch the value from the first text box on the page, the one named `fromVal`. The string is evaluated by the JavaScript DOM by passing it to the aptly-named `eval()` function:

```
var fromValue = eval(fromVal.value);
```

This is a neat trick, but how does it work? When we typed the string `"3.25"` into the `fromVal` element, we weren't thinking of writing JavaScript per se. We were trying to express a numeric value. But the `eval()` function did interpret our input as JavaScript because that's all it can do. The `eval()` function gives you direct access to JavaScript's compiler at runtime, so the string `"3.25"` compiled as JavaScript code is treated as the literal value for the floating point number we know as 3.25. That makes sense. The parsed literal number is then assigned to a local variable defined in the script named `fromValue`. The next line of code in the `convert()` function uses `eval()` once again:

```
toVal.innerHTML = eval(formula.value).toString();
```

The string `"fromValue*25.4"` looks a bit more like a script than the first input because it contains a mathematical expression. The result of executing that script is a number that's converted into a string and written back to the web page for the user to

see. Once again, in that single line of code, you can see the HTML DOM and the JavaScript DOM working together to accomplish what's required.

The bit of metaprogramming lurking in this example is the way that the predefined JavaScript variable called `fromValue` is referenced within the formula provided by the user. The token `fromValue` in the user-supplied formula is somehow *bound* by the second `eval()` statement to the value of the predefined variable in the DOM's local execution scope. This kind of *late binding* is fairly common in metaprogramming. With JavaScript, writing a script that can refer to objects defined in the larger execution context, otherwise called the script scope, is simple to do. When you use libraries like jQuery or the Reactive Extensions for JavaScript (RxJS) for the first time, how they can do so much in so few lines of code seems utterly magical. The magic lies in the metaprogramming foundation upon which JavaScript was conceived, which we examine at the end of this chapter. If JavaScript didn't expose its compiler in this ingeniously simple way, neither jQuery nor RxJS would exist.

Defining the local variable `fromValue` is a convention in the design of this particular web page. Rather than using a variable with a specific name, you could inject your own variable into the local scope and use it instead, as shown in figure 1.3.

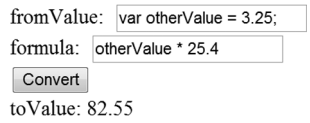

Figure 1.3 DynamicConversion.htm—injecting variables into JavaScript

As you can see in figure 1.3, the value in the predefined `fromValue` variable is no longer being used in the user-supplied formula. This example takes advantage of the fact that when the first `eval()` statement runs in the `convert()` function, any JavaScript code can be provided to the compiler. A new variable named `otherValue` is injected into scope which the formula references instead. This *side effect* functions properly because the inches to millimeters calculation produces the correct output.

If you can create whole new objects using the JavaScript DOM, who knows what else you might be able to reference from a user-supplied script at runtime? You might have access to some of JavaScript's built-in libraries, for example. Let's give that a try. The example shown in figure 1.4 uses JavaScript's built-in `Math` class to calculate the tangent value at 45 degrees. In case you don't remember your college trigonometry, the tangent line on a circle at 45 degrees should have a slope of 1.

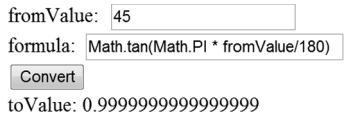

Figure 1.4 DynamicConversion.htm—using JavaScript's `Math` class dynamically

The `tan()` function needs radians, not degrees. The formula first converts the degrees supplied by the user to radians using the constant for pi from JavaScript's `Math` class. In JavaScript, getting the constant for pi is as *easy as pie*, as the saying goes. Then the `Math` class is used again to compute the tangent value using the trigonometric `tan()` function. The result shows a slight rounding error, but it's pretty close and neatly illustrates the idea of using JavaScript's libraries from a dynamic script.

As you can see, the name chosen for the `convert()` function is wearing a bit thin as you begin to realize that this number converter can become pretty much whatever the user wants. For example, pass a single-quoted string for the `fromValue` and invoke one or more JavaScript strings in the formula to manipulate it. As you'll observe, the user-supplied input doesn't have to be a number at all. So it goes with metaprogramming in general. You'll often find that the metaprogramming-enabled interfaces you encounter seem simple from the outside. Beneath the surface, however, a lot of interesting and useful functionality is often waiting to be discovered.

Having studied the important metaprogramming concepts of late binding and runtime compilation, let's turn our attention to another popular technique that's used throughout the .NET Framework Class Library (FCL) to make code easier to write and comprehend.

1.2.2 Metaprogramming via reflection

The surface simplicity that many metaprogramming-enabled interfaces expose is often quite deliberate. As you'll see throughout this book, metaprogramming is commonly used to hide complexity by providing natural interfaces to complicated processes. Let's take a look at one of the simplest uses of this idea. Imagine that a `ListBox` control exists named `listProducts`. Your goal is to load the control with a list of (you guessed it) `Product` objects from a data context. Each `Product` contains a string property named `ProductName` and an integer property named `ProductID`. You want `ProductName` to be visible to the user, and when they click an item in the `ListBox`, you want the associated `ProductID` to be the selected value. Since .NET 1.0, the code to do that has been this simple:

```
listProducts.DisplayMember = "ProductName";
listProducts.ValueMember = "ProductID";
listProducts.DataSource = DataContext.Products;
```

In English, this code might be read as, "Bind these `Product` objects to this `ListBox`, displaying each `ProductName` to the user and setting the `ProductID` for each item as the selectable backing value." Notice how the declarative quality of the code makes it easy to understand what's going on. In fact, the C# code and the English rendering of it are quite similar.

You may have written code that does data binding as we've described dozens of times, but have you ever stopped to think about what's going on behind the scenes? How can strings be used in a statically typed language like C# to locate and bind property values by name at runtime? After all, the strings assigned to the `DisplayMember` and `ValueMember` properties could have been variables instead of string literals. The treatment of them by Microsoft's data-binding code must be performed completely at runtime.

The answer is based on something known as the reflection application programming interface (API), which can *illustrate* the inner workings of a class at runtime,

Declarative programming

In 1957, the FORTRAN programming language appeared—the great-grandparent of all the so-called imperative programming languages. In English, the word *imperative* is used to mean *command* or *duty*. FORTRAN and its descendants are called imperative languages because they give the computer commands to fulfill in a specific order. Imperative languages are good for instructing computers *how* to do work using specific sequences of instructions. The data binding example at hand hints at the power of a programming style called *declarative* that aims to move you from demanding *how* the computer should work to declaring *what* you want done instead. You can express what you want, and Microsoft's data-binding code figures out how to do it for you.

hence the name. Microsoft's `ListBox` data-binding code uses reflection to use bits of metadata left behind by the compiler, as shown in the following listing.

Listing 1.2 `DataSource` reflection logic (C#)

```
public System.Collections.IEnumerable DataSource
{
  set
  {
    foreach (object current in value)
    {
      System.Reflection.PropertyInfo displayMetadata =
        current.GetType().GetProperty(DisplayMember);
      string displayString =
        displayMetadata.GetValue(current, null).ToString();
      // ...

      System.Reflection.PropertyInfo valueMetadata =
        current.GetType().GetProperty(ValueMember);
      object valueObject =
        valueMetadata.GetValue(current, null);
      // ...
    }
  }
}
```

Keep in mind that Microsoft's real data binding code is quite a bit more optimized than this. As each element in the `DataSource` collection is iterated over, its type is obtained using the `GetType()` method, which is inherited from `System.Object`.

NOTE If you have any doubts about how fundamental reflection is in the .NET ecosystem, think for a moment about the significance that the `GetType()` method is included in `System.Object`. The base class for all .NET types is quite sparsely populated yet the `GetType()` method, which is critically important for metadata discovery and metaprogramming, was deemed important enough to be exposed from every single .NET object.

The `System.Type` object returned from `GetType()` has a method called `GetProperty()` that returns a `PropertyInfo` object. In turn, `PropertyInfo` has a method defined within it called `GetValue()` that's used to obtain the runtime value of a property on an object that implements the metadata described by the `PropertyInfo`.

In the `System.Reflection` namespace, you may be interested in several of these `Info` classes for expressing the various types of metadata, such as `FieldInfo`, `MethodInfo`, `ConstructorInfo`, `PropertyInfo`, and so on. As seen in listing 1.2, these classes are categorical in nature. Once you have an `Info` class in hand, you must supply an instance of the type you're interested in to do anything useful. In listing 1.2, the current `Product` reference in the loop is passed to the `GetValue()` method to fetch the instance values for each targeted property. Now that you know the `Info` classes in reflection are categorical, you may be thinking about reusing them to optimize the data binding code. Now that's thinking like a metaprogrammer! The following listing shows an optimized version of the code.

Listing 1.3 Optimized `DataSource` binding logic (C#)

```
public IEnumerable DataSource {
  set {
    IEnumerator iterator = value.GetEnumerator();
    object currentItem;
    do {
      if (!iterator.MoveNext())
        return;
      currentItem = iterator.Current;
    } while (currentItem == null);

    PropertyInfo displayMetadata =
      currentItem.GetType().GetProperty(DisplayMember);
    PropertyInfo valueMetadata =
      currentItem.GetType().GetProperty(ValueMember);

    do {
      currentItem = iterator.Current;
      string displayString =
        displayMetadata.GetValue(currentItem, null).ToString();
      // ...

      object valueObject =
        valueMetadata.GetValue(currentItem, null);
      // ...
    } while (iterator.MoveNext());
  }
}
```

The first portion of the optimized `DataSource` data binding code shown in listing 1.3 iterates until it finds a non-null current item. This is necessary because you can't assume that the collection supplied as the `DataSource` has all non-null elements. The first elements could be empty. Once an element is located, some of its type metadata is cached for later use. Then the iteration over the elements uses the cached `PropertyInfo` objects to fetch the values from each element. As you can imagine, this is a more

efficient approach because you don't have to perform the costly metadata resolution for every single object in the collection. Using caching and other optimizations to improve runtime performance is a common metaprogramming practice.

> **The magic string problem**
>
> One of the drawbacks of any metaprogramming approach that uses literal strings to drive application behavior at runtime is the fact that compile-time verification by compilers can't be performed. What would happen if you misspelled the `DisplayMember` value as `"ProductNane"`? You would discover that error during testing quickly. But what if you allowed the user to specify that string through an application setting, or worse, via a query parameter? Malicious users could begin probing for so-called *magic strings* that could be used to exploit your code by injecting new behaviors. An entire class of related exploits known as SQL injection attacks still plagues poorly designed websites, despite the fact that fixing the problem takes only a few minutes.

For brevity, Microsoft's `DataSource` binding implementation isn't shown here. It includes many interesting optimizations you can learn from. When you're ready, use the skills you pick up in chapter 2 to *introspect* into Microsoft's real data-binding code. You'll learn a lot from that exercise.

Next, we turn our attention to the idea of code generation, which is how most developers define metaprogramming.

1.2.3 Metaprogramming via code generation

So far we've looked at scripting and reflection as tools for metaprogramming. Now let's focus on generating new code at runtime. To ease into the subject, we focus on two of the simpler approaches to code generation using the Microsoft .NET Framework:

- Generating source code with the CodeDOM
- Generating IL with expression trees

To be as illustrative as possible, the approaches are quite different, but the outcomes only vary by the fact that one approach produces source code text, and the other emits new functions that are immediately executable.

CREATING SOURCE CODE AT RUNTIME WITH THE CODEDOM

Document-oriented programming models are common in software design because the document is such a powerfully simple metaphor for organizing information. You may have used the HTML DOM and the JavaScript DOM to do web development, for example. Microsoft has included something known as the CodeDOM in the .NET Framework. As its name implies, the CodeDOM allows you to take a document-oriented approach to code generation.

The CodeDOM comes from the early days of .NET and reflects some of the most primitive thinking about creating a standardized code-generation system for Microsoft's platform. The term *primitive* isn't pejorative in this case because the CodeDOM,

despite the fact that Microsoft hasn't focused its attention there in recent years, is still an elegant code-generation system that many metaprogrammers still enjoy using. The CodeDOM uses a so-called *code graph*-based approach to creating code on the fly.

For all of the CodeDOM snippets shown in this section, the following namespace imports are required:

```
using System;
using System.IO;
using System.Text;
using System.CodeDom;
using System.Diagnostics;
using System.CodeDom.Compiler;
```

To understand how the CodeDOM functions as a source code generator, let's begin by exploring which .NET programming languages the CodeDOM supports. The `CodeDomProvider` class is one of the central classes in the `System.CodeDom.Compiler` namespace and it includes a handy, static method called `GetAllCompilerInfo()`, which returns an array of `CompilerInfo` objects. Each `CompilerInfo` object has a method called `GetLanguages()` you can use to obtain the list of tokens that can be used to instantiate the language provider, like this:

```
foreach (System.CodeDom.Compiler.CompilerInfo ci in
  System.CodeDom.Compiler.CodeDomProvider.GetAllCompilerInfo())
{
  foreach (string language in ci.GetLanguages())
    System.Console.Write("{0}     ", language);
  System.Console.WriteLine();
}
```

Running this snippet in a console application or in LINQPad generates the list of synonyms for each of the installed language providers in the system. Figure 1.5 shows LINQPad acting as a sort of C# scratchpad to execute this bit of code.

As you can see in the LINQPad output, five language providers are installed on our system: C#, Visual Basic, JavaScript, Visual J#, and Managed C++. Each provider allows for the use of three or four synonyms for instantiating them. We come back to provider instantiation near the end of this example.

Notice that F# isn't among the supported languages. Microsoft hasn't been putting much effort into the CodeDOM in the last several years. There have been small enhancements and corrections in recent releases of the .NET Framework, but don't expect to see whole new language providers appear, for example. Microsoft still uses the CodeDOM heavily in its own major subsystems. The Text Templating Transformation Toolkit (T4) engine and the ASP.NET page generator still depend on the CodeDOM for code generation. Going forward, however, Microsoft will almost certainly continue to focus its research and development dollars for code generation in tools like the Roslyn API.

Next, let's take a look at dynamically generating a class. The CodeDOM uses the concept of a *code graph* to assemble .NET objects programmatically. As a C# source

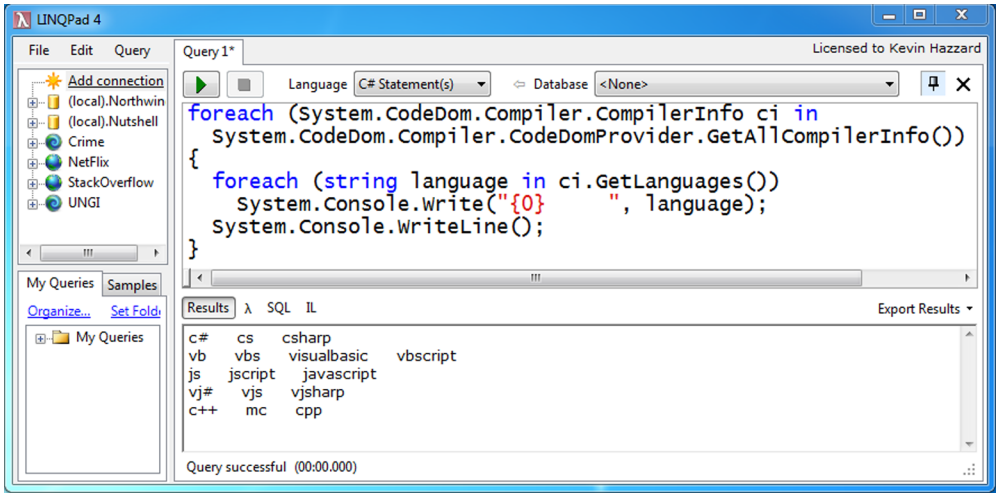

Figure 1.5 Enumerating the synonyms for the CodeDOM language providers using LINQPad

LINQPad: A tool that every .NET developer needs

It's not often that we speak categorically about development tools. As *polyglot* programmers, we admire most development tools in a somewhat egalitarian fashion. Once in a while though, a tool comes along that's so valuable we feel we must recommend it to every developer we meet. LINQPad, written by Joe Albahari, is such a tool. It can be used as a scratchpad for your .NET code. As its name implies, it's also good at helping to write and debug LINQ queries. As of this writing, you can freely download LINQPad from http://LINQPad.net. If you don't already have it, we encourage you to download it and begin exploring right away.

file might start with the declaration of a namespace, a CodeDOM graph typically begins with the creation of a `System.CodeDom.CodeNamespace` object. The `CodeNamespace` serves as the root of the graph. Going back to the source code analogy, the curly braces following a `namespace` declaration in C# are used to contain the types that will be defined within it. The `CodeNamespace` type in the CodeDOM behaves the same way. It's a container in which various types and code can be defined. Before jumping into the code sample, let us take a moment to describe how the code works. Here are the steps:

1. Create a `CodeNamespace` that is the CodeDOM class that represents a CLR (Common Language Runtime) namespace. We'll call our example namespace `MetaWorld` to make it memorable.
2. Create a `CodeNamespaceImport` to import the `System` namespace in the generated source code. These are like `using` declarations in C# or `Import` declarations in Visual Basic.

3. Create a `CodeTypeDeclaration` named `"Program"` for the class that will be generated. This is like using the `class` keyword in your code to declare a new type.
4. Create a `CodeMemberMethod` named `"Main"` that will serve as the entry point function in the `Program` class. The method object will be inserted into the `Program` class. This follows how source code is written. The `Program` class is defined in the namespace, and the `Main` function is defined in the `Program` class.
5. Create a `CodeMethodInvokeExpression` to call `"Console.WriteLine"` with a `CodePrimitiveExpression` parameter of "Hello, world!". This is the hardest part to understand because of the nested way in which the code is structured.

You can probably see where this is going. We'll be dynamically generating the time-honored "Hello, world!" program with the code shown in the following listing.

Listing 1.4 Assembling the "Hello, world!" program with the CodeDOM (C#)

```
partial class HelloWorldCodeDOM
{
  static CodeNamespace BuildProgram()
  {
    var ns = new CodeNamespace("MetaWorld");
    var systemImport = new CodeNamespaceImport("System");
    ns.Imports.Add(systemImport);
    var programClass = new CodeTypeDeclaration("Program");
    ns.Types.Add(programClass);
    var methodMain = new CodeMemberMethod
    {
      Attributes = MemberAttributes.Static
      , Name = "Main"
    };
    methodMain.Statements.Add(
      new CodeMethodInvokeExpression(
        new CodeSnippetExpression("Console")
        , "WriteLine"
        , new CodePrimitiveExpression("Hello, world!")
      )
    );
    programClass.Members.Add(methodMain);
    return ns;
  }
}
```

NOTE In chapter 4, we show in depth how to generate code dynamically using the CodeDOM. In the small example shown here, we used the `CodeSnippetExpression` for simplicity. Using that CodeDOM object can lock you into producing code for one specific language, which often defeats one purpose for using the CodeDOM to begin with.

The `BuildProgram()` method shown in listing 1.4 encapsulates the script outlined earlier, returning a `CodeNamespace` object to the caller. You haven't yet rendered the source code. That comes next. The `CodeNamespace` object can be used by a `CodeDomProvider` to generate source code. Now you have to use one of the five language providers installed on our computer to do the work. The example in listing 1.5 performs the following steps to do that:

1. Create a `CodeGeneratorOptions` object to instruct the chosen compiler how to behave. You can control indentation, line spacing, bracing, and more with this class.
2. Create a `StringWriter` that the language provider will stream the generated source code into. An attached `StringBuilder` holds the generated source code.
3. Create a C# language provider and invoke the `GenerateCodeFromNamespace` method, passing the `CodeNamespace` constructed by the `BuildProgram()` method shown in listing 1.4.

Once completed, the `StringBuilder` will contain the source code you're after. The example program dumps the emitted source code to the console. But it could as easily be written to disk.

Listing 1.5 Generating source code from a `CodeNamespace` (C#)

```
partial class HelloWorldCodeDOM
{
  static void Main()
  {
    CodeNamespace prgNamespace = BuildProgram();
    var compilerOptions = new CodeGeneratorOptions()
    {
      IndentString = "  ",
      BracingStyle = "C",
      BlankLinesBetweenMembers = false
    };
    var codeText = new StringBuilder();
    using (var codeWriter = new StringWriter(codeText))
    {
      CodeDomProvider.CreateProvider("c#")
        .GenerateCodeFromNamespace(
          prgNamespace, codeWriter, compilerOptions);
    }
    var script = codeText.ToString();
    Console.WriteLine(script);
  }
}
```

Compile and run this little code-generator program to see the nicely formatted C# program it produces in the following listing.

Listing 1.6 CodeDOM-generated C# source code for "Hello, world!"

```
namespace MetaWorld
{
  using System;

  public class Program
  {
    static void Main()
    {
      Console.WriteLine("Hello, world!");
    }
  }
}
```

Generating C# source code is easy, isn't it? But what if you wanted to generate managed C++ source code for the same program? You might be surprised at how simple that change is. Modify the string that reads `"c#"` in the call to `CodeDomProver.CreateProvider()` in listing 1.5 to read `"c++"`, and the metaprogram will generate C++ code instead. The following listing shows the C++ version of the dynamically generated source code after making that small change.

Listing 1.7 CodeDOM-generated C++ source code for "Hello, world!"

```
namespace MetaWorld {
  using namespace System;
  using namespace System;
  ref class Program;

  public ref class Program
  {
    static System::Void Main();
  };
}
namespace MetaWorld {
  inline System::Void Program::Main()
  {
    Console->WriteLine(L"Hello, world!");
  }
}
```

The output of the slightly modified program is nicely formatted source code written in Managed C++, which you could save to disk for compilation in a future build step, for example. According to the output from the LINQPad run shown in figure 1.5, you could also have used the synonyms `"mc"` and `"cpp"` to instantiate the C++ language provider. The remaining providers for Visual Basic, JavaScript, and Visual J# are available to create well-formatted code in those languages, too. Give them a try to see that switching the output language when generating source code from a CodeDOM code graph is almost effortless.

We hope this example reveals how straightforward it is to generate source code at runtime. Yet we haven't answered the central question about why you would want to do such a thing. Why would you ever want to generate source code? Here are some ideas taken from real-world projects that have used code-generation techniques successfully:

- Creating entity classes from database metadata for an object-relational mapping (ORM) tool during a build process.
- Automating the generation of a SOAP client to embed features in the proxy classes that aren't exposed through Microsoft's command-line tools.
- Automating the generation of boundary test cases for code based on simple method parameter and return type analysis.

The list goes on and on. Whatever your reasons for wanting to generate source code, the CodeDOM makes it fairly easy. The CodeDOM isn't the only way to generate source code in the .NET Framework, but once you become comfortable with the classes in the

`System.CodeDom` namespace, it's not a bad choice. The preceding example is deliberately simple. When you're ready to dive deeper into the CodeDOM, see chapter 4, which is dedicated to the CodeDOM and which shows many advanced metaprogramming techniques with rich, reusable examples.

Now that we've delved into expressing code as data, let's turn our attention to a more recently introduced way to do that in the .NET Framework.

CREATING IL AT RUNTIME USING EXPRESSION TREES

One of the most common metaprogramming techniques is expressing code as data. That may sound a bit odd at first. The CodeDOM example in the last section described code as a set of data structures to emit source code. An arguably more interesting metaprogramming practice involves compiling the data that represents a body of code into an assembly that can be saved to disk or executed immediately by the running application. This cuts out the step of having to compile intermediate source code files. Better still, if the code graph were somehow independent of the machine architecture, it could be serialized to a remote computer to be compiled and executed there. The remote computer need not be using the same operating system or even the same processor architecture, as long as it has the means for compiling the serialized data structure. Those types of in-memory compilation scenarios are quite a bit more common in metaprogramming than the ones for generating source in an intermediate step.

To demonstrate this idea of in-memory compilation, the code graph must somehow be assembled at runtime into IL. As it turns out, the CodeDOM classes you examined can compile code graphs and blocks of raw source code written in one of the supported languages into .NET assemblies. Those dynamically generated assemblies can be written to disk for future use or exposed as in-memory types for immediate use by the currently executing application. There are also classes in the `Reflection.Emit` namespace that are well-suited for IL generation. But both the CodeDOM and `Reflection.Emit` approaches are a bit too complex for an introductory chapter designed to bring developers up to speed who may be learning about metaprogramming for the first time. Both the CodeDOM and `Reflection.Emit` approaches are important, which is why we dedicate chapters 4 and 5, respectively, to them. To get comfortable with dynamic IL generation in .NET right now, expression trees are the best vehicle for learning the fundamentals.

To understand expression trees, you need to understand a bit of history concerning delegates in the .NET Framework and languages. Delegates were introduced in the first version of the Framework. They were pretty slow in the early days, so a lot of performance-conscious developers avoided using them for computationally intensive work. Early delegates, as expressed in C# and Visual Basic, also had to be named at compile time, which made them a bit awkward feeling. When the 2.0 Framework shipped, anonymous methods were added to the C# language. Under the covers, the runtime implementation of delegates also got a big performance boost at that time. These were steps in the right direction. Higher-order functions could now be declared inline without having to assign names to them. They performed well at

From C++ function pointers to .NET expression trees

The C++ language uses so-called *function pointers* to pass functions around as parameters to other functions. Using this technique, a function caller can provide a variety of implementations at runtime, passing the one that best suits the current needs of the application. Does that sound familiar? Indeed, these so-called higher-order functions in C++ enable a rudimentary kind of application composition that can be used for metaprogramming.

The problem with this approach is that the compiler can't check that the parameters or the return type of the referenced functions correctly match the expectations of the caller. The .NET Framework 1.0 introduced delegates to deal with this problem. Delegates can be passed around like function references, but they fully enforce the call contract in a type-safe way. Through several revisions of the .NET Framework, the delegate concept has greatly evolved. Today we have .NET expression trees, which masterfully blend the concepts of higher-order functions with code as data and a runtime compiler into a rich instrument for everyday metaprogramming.

runtime, too. Anonymous methods made the C# delegate syntax much more coherent, but the language still lacked the overall expressive power of truly functional programming languages.

What makes a programming language functional?

According to computer scientist Dr. John Hughes, in his research paper "Why Functional Programming Matters," programming languages can be considered functional if they have first-class support for both higher-order functions and lazy evaluation. Higher-order functions are those that accept other functions as parameters or return new functions to their callers. The C# language has had that capability since the beginning, courtesy of delegates in the Common Language Runtime (CLR). *Lazy evaluation* means waiting until a calculation is needed to perform it. The .NET class library includes the `Lazy<T>` type for deferring execution, but it's not a language construct. The C# and Visual Basic languages both support the `yield return` syntax in their iterator blocks which, when chained together as the LINQ standard query operators do, exhibits a useful kind of lazy evaluation for list comprehension. But this isn't the language-supported kind of lazy evaluation that Dr. Hughes was talking about. If you want true lazy evaluation capability in a .NET language today, you should take a look at F#, which is the only .NET language from Microsoft that supports it.

In 2006, Microsoft added expression trees to the Base Class Library (BCL) and lambda expression support to the C# and Visual Basic languages. These features were added to support LINQ. With LINQ, the .NET languages could seriously compete with more functional languages for constructing what are known as *list comprehensions*. That term goes way back into computer science history. For now, the best way to think about list comprehension is that it enables lists of objects to be created from other lists. That sounds as absurdly simple as it is. When you think about it, doesn't most of our work

in software development involve list creation and manipulation? Indeed, list handling is one of those core concepts that can make or break a programming language.

With the new LINQ-oriented features added to C#, a function that generically accepts two parameters and returns a result could be expressed as follows:

```
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
```

Functions that compare one integer to another and return a Boolean result would certainly fit this pattern. An instance of a function that tests whether a `Left` parameter is greater than a `Right` parameter might be expressed this way:

```
public bool GreaterThan(int Left, int Right)
{
  return Left > Right;
}
```

It may seem odd to think about "instances of functions," but that metaprogramming concept will become clearer in the next few minutes. The `GreaterThan` function as it's defined above is okay, but using it as a predicate for filtering query results, for example, is a bit cumbersome. The fact that it's an independently defined and named function is part of the problem. To use it, you'd have to wrap it in a specific delegate type or the closed generic type `Func<int,int,bool>`. C# now offers a much more succinct way to do this using a lambda expression:

```
(Left, Right) => Left > Right
```

The `=>` operator is read as "goes to," so for this expression in English, you might read it as "`Left` and `Right` parameters go to the result of testing if `Left` is greater than `Right`." Notice first of all that as a pure expression, there's no requirement that the `Left` and `Right` parameters be of any specific types. You could be comparing floating point numbers, integers, strings—who knows? But for compilers like C#, which isn't as good at doing deep type inference as F# is, you need to get more specific, like this:

```
Func<int, int, bool> GreaterThan = (Left, Right) => Left > Right;
```

Now the `Left` and `Right` parameters are both known by the compiler to be integer types.

We added the name `GreaterThan` back to the definition to show how this newfangled functional delegate described as a lambda expression links back to the old-fashioned function by the same name shown earlier. On the inside, both functions are identical. You could invoke either of them with code like this:

```
int Left = 7;
int Right = 11;
System.Console.WriteLine("{0} > {1} = {2}",
  Left, Right, GreaterThan(Left, Right));
```

This would print to the console `7>11=False` as you'd expect. Being able to define functional delegates using lambda expressions sure makes the code more succinct. The compiler support for lambda expressions is nice, but using lambda expressions in

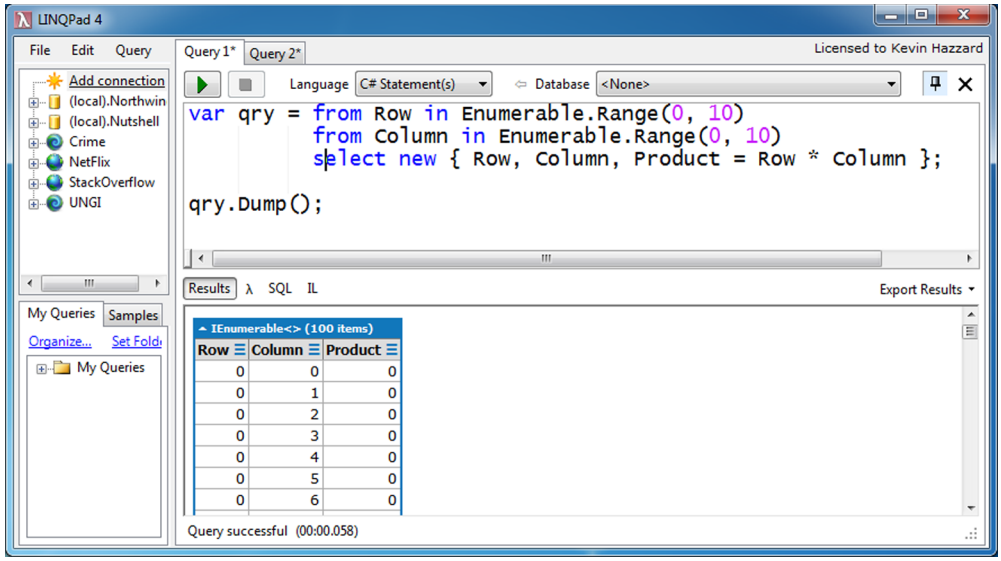

Figure 1.6 Cross-joining two ranges in LINQ

this way isn't how they add real value. Using them inline in LINQ expressions is more typical. Figure 1.6 shows LINQPad being used again. This time, we're performing a cross-join in LINQ to multiply one range of numbers against another. The `Dump()` function is a LINQPad feature that makes it easy to dump the results of an expression. If you were running the example code in a console application, you'd need to write out the results with custom code.

With two ranges of 10 values each, the result contains 100 items as shown in LINQPad's Results tab. Forty-five of those 100 results are redundant because the cross-joined ranges overlap and multiplication is commutative. You can eliminate the duplicates by comparing the row and column values in a predicate, by adding a filter like this:

```
qry.Where(a => a.Row >= a.Column).Dump();
```

Notice how the lambda expression passed to the `Where()` standard query operator looks a lot like the `GreaterThan Func<int,int,bool>` shown earlier. The difference is that rather than taking two parameters, the two compared values are accessed from properties of a single parameter. By using that expression in the `Where()` standard query operator, the results are filtered by it. We call this type of filtering function a *predicate*.

You could read the predicate expression `a=>a.Row>=a.Column` in English as "Return true for items where the `Row` number is greater than or equal to the `Column` number." If you're unaccustomed to LINQ, the way this expression is used in context may be a bit confusing. What's the `a` parameter? Where did it come from? What type is it? One of the clues can be found in the Results tab in LINQPad. Notice in figure 1.6 that the result of the `Dump()` is of type `IEnumerable<>`. The query must produce a list

of something. Still, you don't know what type those items in the list have because the `selectnew{…}` syntax was used to produce an anonymous type which has no name from the programmer's perspective. You can tell from the output that each unnamed thing has a `Row` property, a `Column` property, and a `Product` property. Behind the scenes, the items in the list do have a named type, but you wouldn't want to read it. For anonymous types, the compiler generates a long, strange-looking name that only has value internally. If LINQPad were to show you the name in the Results, it would only get in the way.

Now that you understand that the query produces a list of anonymously typed objects, the parameter named `a` in the lambda expression might make a bit more sense. In this case, the name `a` was chosen because each of the objects passed to the function will be one of those anonymous types. You could have used any name for the parameter. But when lambda functions are embedded like this, we often use parameter names that are a bit more compact. This increases comprehension because the use of the function is so close to its definition, long descriptive names aren't often needed. When functions are declared the old-fashioned way, totally separated from their points of use, more descriptive parameter names tend to increase comprehension.

In figure 1.7, reading the numbers from top to bottom and left to right, you can visualize what the improved results of the filtered query look like.

The 45 duplicate values in the cross-join operation that would have appeared on the bottom and left sides have been filtered out by the predicate. Try the filtered query in LINQPad to see that it produces the result shown in figure 1.7. Then try using some other predicates to manipulate the results in interesting ways. After all, the best way to learn is to play.

Now that you understand how to filter queries using a lambda expression in LINQ, you're ready to understand an interesting metaprogramming connection. Asking the C# compiler to turn the lambda expression into a function is purely a compile-time process. It may be newfangled looking, but it's still sort of *old school*, as they say. What if you need to be able to pass in a variety of filter predicates based on the circumstances

0	0	0	0	0	0	0	0	0	0
	1	2	3	4	5	6	7	8	9
		4	6	8	10	12	14	16	18
			9	12	15	18	21	24	27
				16	20	24	28	32	36
					25	30	35	40	45
						36	42	48	54
							49	56	63
								64	72
									81

Figure 1.7 The results of filtering a cross-join of two ranges with a lambda predicate

The importance of playfulness

For adults, some types of play are pure distraction. Blasting away at aliens in a first-person shooting game for hours at a time can certainly wash away the worries of the day, but it probably doesn't help to establish and refine the prototypes that your mind needs to absorb new ideas. Playfulness can be useful as a learning tool. Small children lack analytical skills, so they use play to build experience and knowledge about how the world works. As you get older, your play becomes more and more structured until eventually you may even forget how to do it. Throughout this book, we encourage you to be playful with the topics we're teaching you. When we show you one way to do something, try a few variations to cement what you've learned deep into your mind.

at the moment? Moreover, what if some of those *filtering algorithms* can't be known at compile time? Or perhaps they come from a remote process that can create new filters on the fly, sending them across the wire to your application to compile and use. Metaprogramming to the rescue!

Moving from the idea of concrete, compile-time functions to a more generic abstraction of those functions is fairly straightforward in .NET, thanks to so-called expression trees and the latest compilers from Microsoft. You've already seen how the C# compiler can turn a lambda expression into a real function that you can call like any other. Internally, compilers operate by parsing the text of source code into abstract syntax trees (AST). These trees follow certain basic rules for expressing code as data. When lambda expressions are compiled, for example, they fit into the resulting AST as any other .NET constructs would.

What happened to Compiler-as-a-Service?

At the Microsoft Professional Developer Conference in 2008, Anders Hejlsberg hinted at things to come in future versions of the C# programming language. One of them was called Compiler-as-a-Service. The basic idea was to expose some of the C# compiler's black box functionality for developers outside of Microsoft to use. Microsoft has since dropped that name, but the ideas behind what was known as Compiler-as-a-Service are alive and well. Jump ahead to chapter 10 to find out more right away.

What if you could preserve the AST created by the compiler so that it could be modified at runtime to suit your needs? If that were possible, all sorts of interesting things would be possible. Unfortunately, as of this writing the parser and AST generator for C# is still exposed in such a way that the average developer can't use it to implement the dynamic execution scenarios just described. But the aptly named expression trees in the .NET Framework are an interesting step in that direction.

Expression trees were introduced into .NET 3.0 to support LINQ. Then they got great enhancements to support the Dynamic Language Runtime (DLR) in version 4.0 of the Framework. In addition to the expression tree enhancements in version 4.0, the

.NET compilers got the ability to parse C# and Visual Basic code directly into expressions. Think back to the `GreaterThan()` function defined as a lambda expression earlier. Remember the `Func<int,int,bool>` that created a real, callable function at compile time? Now evaluate the following line of code and look for the differences:

```
Expression<Func<int, int, bool>> GreaterThanExpr =
  (Left, Right) => Left > Right;
```

Syntactically, the `GreaterThan Func<>` has been *enclosed* in an `Expression<>` type and renamed to `GreaterThanExpr`. The new name will make it clearly different in the discussion that follows. But the lambda expression looks exactly the same. What's the effect of the change? First, if you try to compile an invocation of this new `GreaterThanExpr` expression, it will fail:

```
bool result = GreaterThanExpr(7, 11); // won't compile!
```

The `GreaterThanExpr` expression can't be invoked directly as the `GreaterThan` function could. That's because after compilation, `GreaterThanExpr` is data, not code. Rather than compiling the lambda expression into an immediately runnable function, the C# compiler built an `Expression` object instead. To invoke the expression, you need to take one more step at runtime to convert this bit of data into a runnable function.

```
Func<int, int, bool> GreaterThan =
  GreaterThanExpr.Compile();

bool result = GreaterThan(7, 11); // compiles!
```

The `Expression` class exposes a `Compile()` method that can be called to emit runnable code. This dynamically generated function is identical to the one produced by both the old-fashioned, separately defined method named `GreaterThan` and the precompiled `Func<>` delegate by the same name. Calling a method to compile expressions at runtime may feel a bit odd at first. But once you experience the power of dynamically assembled expressions, you'll begin to feel right at home.

How LINQ uses Expressions

LINQ queries benefit directly from the way that `Expression`s can be compiled. But the various LINQ providers don't typically call the `Compile()` method as shown here. Each provider has its own method for converting expression trees into code. When a predicate like `a => a.Row > a.Count` is compiled, it might produce IL that can be invoked in a .NET application. But the same expression could be used to produce a WHERE clause in a SQL statement or an XPath query or an OData $filter expression. In LINQ, expression trees act as a sort of neutral form for conveying the intent of code. The LINQ providers interpret that intent at runtime to turn it into something that can be executed.

As you've seen in the previous example, the C# compiler can build an `Expression` for you at compile time. That's certainly convenient, but you can also assemble lambda

expressions by hand which may be useful in some applications. The code in the following listing shows how to construct and compile an `Expression` class programmatically that implements the `GreaterThan` function seen previously.

Listing 1.8 Assembling a lambda `Expression` manually

```
using System;
using System.Linq.Expressions;

class ManuallyAssembledLambda
{
  static Func<int, int, bool> CompileLambda()
  {
    ParameterExpression Left =
      Expression.Parameter(typeof(int), "Left");
    ParameterExpression Right =
      Expression.Parameter(typeof(int), "Right");

    Expression<Func<int, int, bool>> GreaterThanExpr =
      Expression.Lambda<Func<int, int, bool>>
      (
        Expression.GreaterThan(Left, Right),
        Left, Right
      );

    return GreaterThanExpr.Compile();
  }

  static void Main()
  {
    int L = 7, R = 11;
    Console.WriteLine("{0} > {1} is {2}", L, R,
      CompileLambda()(L, R));
  }
}
```

The `CompileLambda()` method starts by creating two `ParameterExpression` objects: one for an integer named `Left`, and another for an integer named `Right`. Then the static `Lambda<TDelegate>` method in the `Expression` class is used to generate a strongly typed `Expression` for the delegate type you need. The `TDelegate` for the lambda expression is of type `Func<int,int,bool>` because you want the resulting expression to take two integer parameters and return a Boolean value based on the comparison of them. Notice that the root of the lambda expression is obtained from the `GreaterThan`property on the `Expression` class. The returned value is an `Expression` subclass known as a `BinaryExpression`, meaning it takes two parameters. The `Expression` type serves as a factory class for many `Expression`-derived types and other helper members. Here are a few of the other `Expression` subtypes you're likely to use when building expression trees programmatically:

- `BinaryExpression`—`Add`, `Multiply`, `Modulo`, `GreaterThan`, `LessThan`, and so on
- `BlockExpression`—Acts as a container for a sequence of other `Expressions`
- `ConditionalExpression`—`IfThen`, `IfThenElse`, and so on

- `GotoExpression`—For branching and returning to `LabelExpressions`
- `IndexExpression`—For array and property access
- `MethodCallExpression`—For invoking methods
- `NewExpression`—For calling constructors
- `SwitchExpression`—For testing object equivalence against a set of values
- `TryExpression`—For implementing exception handling
- `UnaryExpression`—`Convert`, `Not`, `Negate`, `Increment`, `Decrement`, and so on

The list goes on and on. In fact, there are more than 500 methods and properties returning dozens of expression types in that class. They cover about any coding construct you can imagine (and probably many more that you can't). Complex expression trees can be constructed entirely from `Expression`–derived objects instantiated directly from static properties and methods in this base class.

To round out this introductory section on the topic, let's look at one more interesting example. The manually assembled lambda expression shown earlier is nice, but it only provides predicates for integers. Moreover, it emits code only for greater-than operations. The following listing shows a more dynamic version of that code that can be used for any data type and a variety of ordering comparisons.

Listing 1.9 A `DynamicPredicate` class using expression trees

```
using System;
using System.Linq.Expressions;

class DynamicPredicate
{
  public static Expression<Func<T, T, bool>>
    Generate<T>(string op)
  {
    ParameterExpression x =
      Expression.Parameter(typeof(T), "x");
    ParameterExpression y =
      Expression.Parameter(typeof(T), "y");
    return Expression.Lambda<Func<T, T, bool>>
      (
        (op.Equals(">")) ? Expression.GreaterThan(x, y) :
          (op.Equals("<")) ? Expression.LessThan(x, y) :
          (op.Equals(">=")) ? Expression.GreaterThanOrEqual(x, y) :
          (op.Equals("<=")) ? Expression.LessThanOrEqual(x, y) :
          (op.Equals("!=")) ? Expression.NotEqual(x, y) :
          Expression.Equal(x, y),
        x, y
      );
  }
}
```

A generic function has been built to generate a type-safe expression based on the compared type and a comparison operation using a type parameter and a standard string parameter, respectively. The generator function is aptly named `Generate`. In the

following listing, notice how predicates for different data types can now be defined and compiled dynamically.

Listing 1.10 Invoking the `DynamicPredicate`

```
static void Main()
{
  string op = ">=";
  var integerPredicate =
    DynamicPredicate.Generate<int>(op).Compile();
  var floatPredicate =
    DynamicPredicate.Generate<float>(op).Compile();

  int iA = 12, iB = 4;
  Console.WriteLine("{0} {1} {2} : {3}", iA, op, iB,
      integerPredicate(iA, iB));

  float fA = 867.0f, fB = 867.0f;
  Console.WriteLine("{0} {1} {2} : {3}", fA, op, fB,
      floatPredicate(fA, fB));

  Console.WriteLine("{0} {1} {2} : {3}", fA, ">", fB,
      DynamicPredicate.Generate<float>(">").Compile()(fA, fB));
}
```

The first predicate generated in this example uses the greater-than-or-equal-to operator on integer types. The next one is for the same operator comparing floating-point types. The predicates are then used to perform simple comparisons. In the last statement, a dynamic predicate is built for the greater-than operator on floating-point types, which is used to compare the same floating-point values from the last invocation. Figure 1.8 shows the result of running the code.

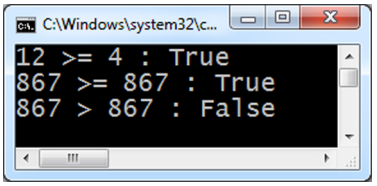

Figure 1.8 Exercising the `DynamicPredicate` class

We've only scratched the surface of what expression trees can do in .NET. The power of LINQ and the DLR wouldn't be possible without them. For example, LINQ's `IQueryable` interface can be used to consume dynamically assembled expressions, giving you a truly elegant way to make the search and query interfaces in your applications simple to extend over time. For that example and more, turn to chapter 6. In the meantime, let's take a look at one more way to do metaprogramming in .NET: dynamic typing.

1.2.4 Metaprogramming via dynamic objects

Statically typed languages rule the roost, as they say, in the .NET world. Even though Microsoft's IronPython implementation of the venerable Python programming language is impressive both in terms of performance and compatibility, programmers accustomed to working on the Microsoft stack don't seem to be as attracted to it as some of us had hoped. It's not just that old habits die hard. New skills are quite difficult to form, in

particular when one believes that the tools they use are excellent for problem-solving. Asking C++ and Visual Basic 6 developers to upgrade their skills to learn C# and Visual Basic .NET was difficult enough. Asking those developers to invest time and energy to learn Python and Ruby has proven to be tougher still.

Dynamic languages like JavaScript, Python, and Ruby have a lot to offer. The languages themselves are all wonderfully expressive. Well-developed platforms and libraries like jQuery, Django, and Rails make it easy to get going. After working with these languages for a while, one discovers what seems to be endless depth in the included libraries. Almost anything you could ever want for building rich applications has been created and captured in the standard libraries. Pythonistas say about their language that it comes *batteries included*.

Alas, dynamic languages may never be as popular on the .NET Framework as our trustworthy, statically typed companions of old. But that doesn't mean that the dynamic programming language developers should have all the fun.

A C# DYNAMIC TYPING BACKGROUNDER

On Jan. 25, 2008, Charlie Calvert of Microsoft Corporation posted a blog article titled "Future Focus I: Dynamic Lookup." In that post, he wrote, "The next version of Visual Studio will provide a common infrastructure that will enable all .NET languages, including C#, to optionally resolve names in a program at runtime instead of compile time. We call this technology dynamic lookup."

True to his word, version 4.0 of the C# programming language included great support for creating and handling dynamically typed objects. Charlie went on in the post to list the key scenarios for using this new capability. Years later, his list is still compelling. The list includes:

- Office automation and COM interop
- Consuming types written in dynamic languages
- Enhanced support for reflection

We look at the first two scenarios in detail in Part 3 (chapters 8-10) of this book. Because you've already gotten a taste of reflection in this chapter, let's build on that learning by examining Charlie's third scenario. We begin by taking a quick tour of so-called *duck typing*. This odd-sounding term traces its origins to James Whitcomb Riley, a 19th century poet: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." The current phrase, specific to ensuring type-appropriateness in programming languages, is only a couple of decades old.

With respect to computer programming, you might translate Riley's "walk like a duck…" line into "If an object supports the methods and properties I expect, I can use them."

We use the word *expect* deliberately because the duck-typing concept is all about compile-time versus runtime expectations. If you expect an object to have a method named `CompareTo` in it, taking one `Object` parameter and returning an integer result, do you care how it got there? The answer depends somewhat on your worldview. More

The L in SOLID

The programming acronym SOLID packs a lot of meaning into five little letters. The L stands for the Liskov substitution principle (LSP), which has a genuinely formidable sounding ring to it. Although it may sound a bit unnerving, the LSP isn't all that challenging to understand. It means that subtypes behave like the types from which they're derived. Inherent in the LSP is that the compiler gives the programmer support for enforcing type correctness at compile time.

Why is this significant to understand in a discussion about duck typing? Well, in statically typed languages like C#, classes behave like contracts. They make promises about their members, the count, order, and type of parameters that must be provided, and the return types. Truly dynamic languages don't enforce contracts this way, which makes them feel different to the programmer who's accustomed to getting LSP support from the compiler. This will be important to keep in mind as you learn about C#'s dynamic typing capabilities.

importantly, it depends on your tools. Examine the code in figure 1.9, which throws an exception at runtime while trying to perform a simple sort.

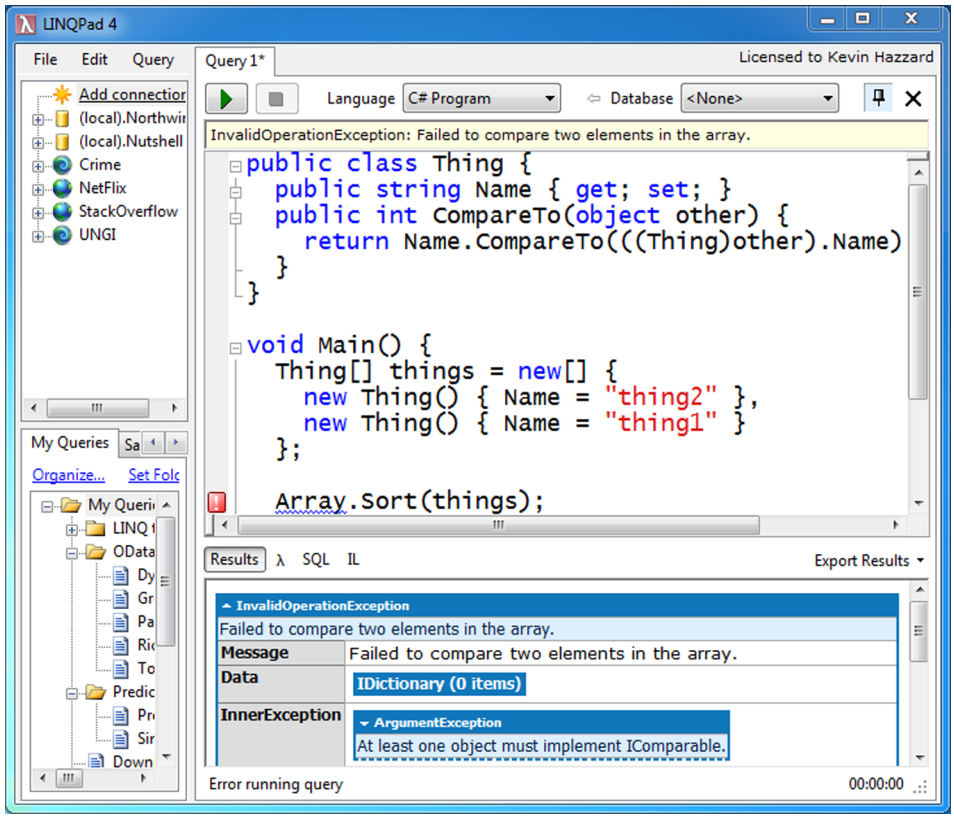

Figure 1.9 Simple sorting that throws an exception

The code looks okay, and indeed, it compiles perfectly. But an `ArgumentException` is thrown at runtime from the `Sort` function, indicating that "At least one object (in the comparison) must implement `IComparable`." If you were new to C#, having come from the Python or Ruby worlds, this error might cause real confusion. Having looked to other C# programs as examples, a dynamic-language programmer might have concluded that implementing a `CompareTo` function in a class with the expected method signature was all that's required to do sorting of arrays containing that type. The `Sort` function implementation is a bit more demanding than that, however. Not only must you include a `CompareTo` method in your class, it must specifically be of the type `IComparable.CompareTo`. Adding that one simple declaration to the `Thing` class solves the problem:

```
public class Thing : IComparable
{
  public string Name { get; set; }
  public int CompareTo(object other)
  {
    return Name.CompareTo(((Thing)other).Name);
  }
}
```

This kind of demand is foreign to programmers accustomed to working in dynamic languages because it doesn't seem substantive to them. In fact, to a dynamic-language programmer, this sort of demand feels downright offensive. Why should the `Array.Sort` function care how the `CompareTo` method got into the `Thing` class? To them, the existence of the function at runtime should be enough.

No value in religious debate

This is the point in the discussion where we could spiral downward into a somewhat religious argument about the worth of various programming models, but we're going to resist the urge. The dynamic way of programming is no better or worse than the static way. It's different. Some problems are well-suited to one type of solution or another. The fact of the matter is that software developers get great work done in statically typed environments and dynamically typed ones, too. That makes both approaches worthy of study and respect.

The question that Charlie Calvert and the C# compiler team posed in 2008 is: "Can one modern programming language support both the static and dynamic typing models well?" With all due respect, the answer to that question is resoundingly no. C# is still a statically typed language. The dynamic capability in version 4.0 has been bolted on to the side of the language, as you'll discover in a moment. Declaring and using a dynamic object in C# could hardly be easier:

```
dynamic name = "Kevin";
System.Console.WriteLine("{0} ({1})",
  name, name.Length);
```

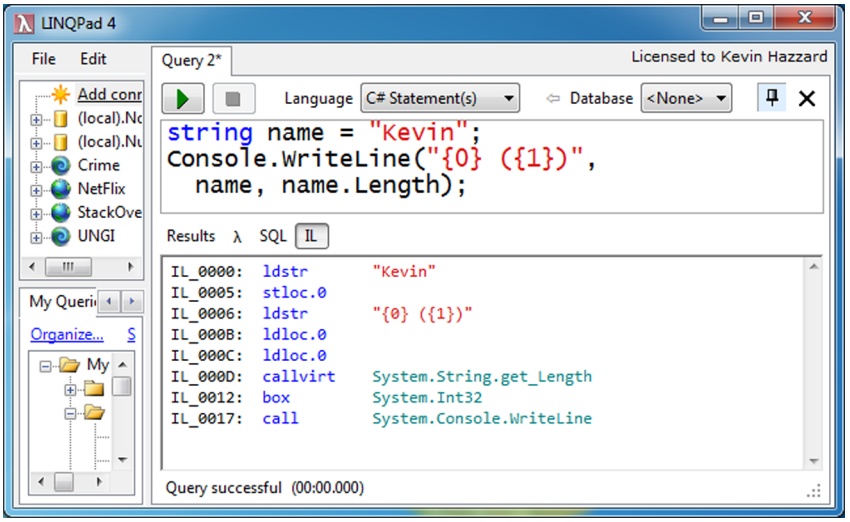

Figure 1.10 The IL from writing a string to the console

Run that code in LINQPad after selecting C# Statements from the Language drop-down list to see that it dutifully formats and prints the name and the length of the string as `'Kevin(5)'` to the Results tab. Now change the declared type for the `name` variable to a `string` and run it again. You'll notice that the output in the Results tab is identical. What's the difference? While you have the `name` variable defined as a `string`, switch to LINQPad's IL tab. This will show you the compiled IL for the code, which will look something like figure 1.10.

Your introduction to IL is coming in chapter 2, but this code is so simple, you should be able to make out what's going on. The two literal strings you see in the C# code are pushed onto the stack before the `get_Length` method is called on the `System.String` class using the `callvirt` opcode. If you've never looked at IL, you may be surprised to find that the `Length` property accessor for a string is implemented with the name `get_Length`. The result of calling `get_Length` is of type `System.Int32`, but the `Console.WriteLine` method expects parameters of type of `System.Object`. The `box` opcode is used to box that integer value type as an object before calling `System.WriteLine`. All this was done with eight IL opcodes.

Now change the type of the `name` variable back to `dynamic`, rerun the code, and observe the IL it produces. We won't show that IL listing here because it would take up a couple of pages and require several more pages to fully describe. In chapter 8, we do a deeper investigation of the `CallSite` class and other metaprogramming-relevant classes from the `System.Runtime.CompilerServices` and `Microsoft.CSharp` namespaces. As you scroll through the IL on your own, take time to notice two literal strings that are pushed on the stack that weren't pushed in the previous example:

```
IL_0012:  ldstr       "WriteLine"
//... some code omitted ...
IL_0089:  ldstr       "Length"
```

You may also notice that the reference to the `get_Length` method is also missing from the IL. How could the `get_Length` method be invoked if it's not in the IL? The fact that it's missing is an interesting clue, as it turns out. Also, do you see literal strings for `"WriteLine"` and `"Length"` in the original C# code? No, so why do these literal strings appear in the IL now? When the type of the `name` variable was changed from `string` to `dynamic`, many changes happened under the covers, as you can see.

The outermost change involves the emission by the C# compiler of `CallSite` objects into the IL. A `CallSite` is literally the *site* in the code where something dynamic happens. It may surprise you that in this code, there are two `CallSite` objects. One is used to invoke the `Length` property accessor on the `name`, which happens through a call to the runtime binder's `GetMember` method:

```
IL_0089:  ldstr       "Length"
//... some code omitted ...
IL_00AA:  call        Microsoft.CSharp.RuntimeBinder.Binder.GetMember
```

That makes sense given that the `name` variable was marked as `dynamic`. Using the dot operator after the `name` variable to invoke the `Length` property ends up passing the literal string `"Length"` to the C# runtime binder to reflect against the object to get the value. Do you remember that Charlie said in his blog post how *dynamic lookup* would simplify reflection? The reflection that the C# runtime binder is doing for you also explains why the call to the `get_Length` function is conspicuously absent in this version of the code. There's no need to bind up a call to `get_Length` at compile time because the invocation is going to happen at the `CallSite` at runtime via reflection.

The remaining mysteries at this point are (a) that literal string `"WriteLine"` that you found in the IL, and (b) that second `CallSite` that was emitted into the site container. Could they be related? Indeed, they are related. The second `CallSite` is used to call `InvokeMember` on the C# runtime binder to dispatch a dynamic call to the static `"WriteLine"` method on the `System.Console` class. The question that might pop into your mind is: "Why on Earth is the `WriteLine` method being called dynamically?" After all, nothing about `System.Console` was declared to be dynamic.

This code highlights one of the biggest concerns that many developers have about dynamic typing in C#. When you pass a type declared as `dynamic` to a method, or if that method returns such a type, that method call will also be implemented dynamically through a `CallSite` that's emitted by the compiler. Most developers expect to pay a price for using the `dynamic` keyword in C#, but they don't expect that it will have a ripple effect, causing other nearby function calls and member accesses to become dynamically invoked as well. The best advice we can give is to be careful when using C#'s `dynamic` keyword.

Now that you have an appreciation for what's happening behind the scenes with dynamic typing in C#, let's turn our attention to a simple but useful class that can be used to implement dynamic property bags.

No dynamic type in C#

You may be surprised to find that there's no backing type for the `dynamic` keyword in C#. The functionality enabled by the `dynamic` keyword is a clever set of compiler actions that emit and use `CallSite` objects in the site container of the local execution scope. The compiler manages what programmers perceive as dynamic object references through those `CallSite` instances. The parameters, return types, fields, and properties that get dynamic treatment at compile time may be marked with some metadata to indicate that they were generated for dynamic use, but the underlying data type for them will always be `System.Object`.

IMPLEMENTING METAOBJECTS IN C#

Many dynamic languages allow any object to be treated like a property bag. Members can be added to or removed from the bag at will. Some dynamic languages even let you modify the class definitions on the fly so that newly instantiated objects of those types will get the updated definition. Using Python, for example, it's simple to add properties to an instance on the fly using code like this:

```
>>> class PyExpandoObject():
...     pass
...
>>> container = PyExpandoObject()
>>> container.Name = 'Jenny'
>>> container.PhoneNumber = 8675309
>>> print container.Name, '-', container.PhoneNumber
Jenny - 8675309
```

In this code, The `PyExpandoObject` class is defined as an empty class using the `pass` keyword. Then an instance of the `PyExpandoObject` named `container` is allocated. What happens next may seem odd to a developer using statically typed languages, but it's common in many metaprogramming environments. Two new members called `Name` and `PhoneNumber` are added by assigning values to their names. A `print` statement is used to report the values of the new members back to the console. Python infers the types of the new members correctly, which you can test by using Python's `type()` function.

```
>>> type(container.Name)
<type 'str'>
>>> type(container.PhoneNumber)
<type 'int'>
```

Internally, Python manages a dictionary of its members, which it can modify on the fly, adding new members or redefining them as necessary. Python even allows the programmer to delete members programmatically. Adding new members to a class instance in C# 4.0 is similarly simple when using the `ExpandoObject` class:

```
dynamic container = new System.Dynamic.ExpandoObject();
container.Name = "Jenny";
container.PhoneNumber = 8675309;
```

```
Console.WriteLine("{0} - {1}",
  container.Name, container.PhoneNumber);
```

This C# code will write the same string to the console that the preceding Python code did. In Python, any object can act as a dynamic property bag. In C#, though, you must use an `ExpandoObject` or build that functionality into one of your own classes. Not surprisingly, the `ExpandoObject` uses a dictionary object internally to mimic the functionality that Python offers. What's unclear, however, is how the C# compiler understands how to interact with the `ExpandoObject` class to enable new name value pairs to get into the internally managed dictionary.

In the example shown earlier involving a dynamic string, the `GetMember` function from the C# runtime binder was invoked to reflect on the string to obtain the value of its `Length` property. The `GetMember` function was called because you were trying to get the value of the `Length` property to display on the console. In the previous C# code using `ExpandoObject`, the assignment to `container.Name` and `container.PhoneNumber` are clearly not going to invoke `GetMember` in the binder, because you're attempting to mutate the values, not fetch them. As you can imagine, the DLR also includes a `SetMember` function in the C# runtime binder for this purpose. The IL that sets the value `"Jenny"` to the `Name` property follows this abbreviated flow:

```
IL_000E:  ldstr       "Name"
//... some code omitted ...
IL_0039:  call        Microsoft.CSharp.RuntimeBinder.Binder.SetMember
//... some code omitted ...
IL_0058:  ldstr       "Jenny"
```

The C# compiler emits a call to `SetMember` for the `"Name"` property to set the value `"Jenny."` The C# compiler seems to have done its part well, but you still don't know how the name value pair `"Name"`, `"Jenny"`) is going to get into the `ExpandoObject`'s internal dictionary. The answer to that comes by looking at the implementation of `ExpandoObject`, which implements six interfaces:

```
IDynamicMetaObjectProvider
IDictionary<string,object>
ICollection<KeyValuePair<string,object>>
IEnumerable<KeyValuePair<string,object>>
IEnumerable
INotifyPropertyChanged
```

The first interface in the list is the one that enables the standard C# runtime binder's `SetMember` function to call custom code to manage `ExpandoObject`'s internal dictionary object. The definition of `IDynamicMetaObjectProvider` is deceptively simple looking:

```
public interface IDynamicMetaObjectProvider
{
  DynamicMetaObject GetMetaObject(Expression parameter);
}
```

In this interface, you're beginning to see common metaprogramming terms that you can recognize from examples in this chapter. You know what *dynamic* means. You

know what `Expressions` are. Metaobjects aren't yet well-defined, but they're almost certainly some kind of type used in metaprogramming.

> **Meta madness!**
>
> The appearance of the prefix *meta* over and over again in metaprogramming jargon can be a bit overwhelming. It's not a prefix that we encounter on English words all that often, so it can be a bit confusing. Remember that in Greek, *meta* means *after* or *beside*. You might read the DLR term *metaobject* to mean *after-object* or *beside-object*. The way the DLR uses metaobjects in conjunction with the runtime binders, they fit the *beside-object* definition better. Metaobjects run alongside other types like the `ExpandoObject` to help the runtime binder in binding up specific methods like `SetMember` and `GetMember` when the code demands to set or get named values, respectively.

By implementing this interface, the `ExpandoObject` can interact with the C# runtime binder by providing handlers for specific events that occur in the lifecycle of those types. The `DynamicMetaObject` returned by the `GetMetaObject` function in the interface has many virtual methods that can be overridden to provide specific types of runtime binding functionality. We cover all of these methods in detail in chapter 8. For now, the two methods required to understand the interface between C#'s runtime binder and `ExpandoObject`'s internal dictionary are

```
BindGetMember
BindSetMember
```

When the runtime binder observes that the dynamic object it's operating on implements the `IDynamicMetaObjectProvider` interface, it defers the binding calls to the methods in the `DynamicMetaObject` that's provided through that interface rather than trying to resolve them with reflection. This multistep process is admittedly arcane sounding, but once you get the hang of using it, you'll understand that it's as simple as it needs to be and flexible enough to handle nearly any metaprogramming scenario.

To remove any remaining mystery, let's implement an expandable property bag called `MyExpandoObject`, providing custom implementations for `GetMember` and `SetMember` at runtime. Rather than implementing the entire `IDynamicMetaObject-Provider` contract, let's take a shortcut. A helper class has been included in the .NET Framework called `DynamicObject` that implements `IDynamicMetaObject-Provider` for you, hiding the somewhat complex `Bind*` methods and exposing a set of similarly named but simpler `Try*` methods instead. To implement the dynamic property bag, you'll need to derive your class from `DynamicObject` and override the `TryGetMember` and `TrySetMember` functions to provide your custom binding code. The following listing shows the definition of the `MyExpandoObject` type.

Listing 1.11 `MyExpandoObject`: a DLR-based dynamic property bag

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

public class MyExpandoObject : DynamicObject
{
  private Dictionary<string, object> _dict =
    new Dictionary<string, object>();

  public override bool TryGetMember(
    GetMemberBinder binder, out object result)
  {
    result = null;
    if (_dict.ContainsKey(binder.Name.ToUpper()))
    {
      result = _dict[binder.Name.ToUpper()];
      return true;
    }
    return false;
  }

  public override bool TrySetMember(
    SetMemberBinder binder, object value)
  {
    if (_dict.ContainsKey(binder.Name.ToUpper()))
      _dict[binder.Name.ToUpper()] = value;
    else
      _dict.Add(binder.Name.ToUpper(), value);
    return true;
  }
}
```

For this implementation, we've decided that the properties inserted into the bag shouldn't have case-sensitive names. Programmers should be able to save a value into the property bag named `JABBERWOCKY` and retrieve it later with the name `jAbBeRwOcKy`, for example. The `ToUpper` function on the string class is used whenever properties are set and fetched from an internally managed dictionary containing the name value pairs. The code in the following listing shows how the `MyExpandoObject` might be used.

Listing 1.12 Exercising `MyExpandoObject`

```
class TestMyExpandoObject
{
  static void Main()
  {
    dynamic vessel = new MyExpandoObject();
    vessel.Name = "Little Miss Understood";
    vessel.Age = 12;
    vessel.KeelLengthInFeet = 32;
    vessel.Longitude = 37.55f;
    vessel.Latitude = -76.34f;
    Console.WriteLine("The {0} year old vessel " +
```

```
        "named {1} has a keel length of {2} feet " +
        "and is currently located at {3} / {4}.",
        vessel.AGE, vessel.name,
        vessel.keelLengthINfeet,
        vessel.Longitude, vessel.Latitude);
    }
}
```

After instantiating a `MyExpandoObject` and assigning the reference to a dynamic variable named `vessel`, properties of different types are placed into the property bag. Each assignment will invoke the overridden `TrySetMember` implementation, which will place them into the internal dictionary object. At the end, the properties are fetched from the property bag by name. To exercise the case-insensitive handling of property names, they've been deliberately cased differently than they were in assignments beforehand. Figure 1.11 shows the result of running the code in listing 1.10 and listing 1.11 in LINQPad.

This little metaprogramming-enabled class does a great job of raising the abstraction level for managing name value pairs, making the code highly reusable. It also increases comprehension by providing a natural interface while reducing the perceived complexity.

1.3 Summary

In this chapter, we spent time trying to convey that metaprogramming might sometimes be a bit complex on the inside, but it can greatly reduce the perceived complexity on the outside of the classes that you provide to your team. You also learned how cohesion and abstraction relate to complexity and how metaprogramming can help to

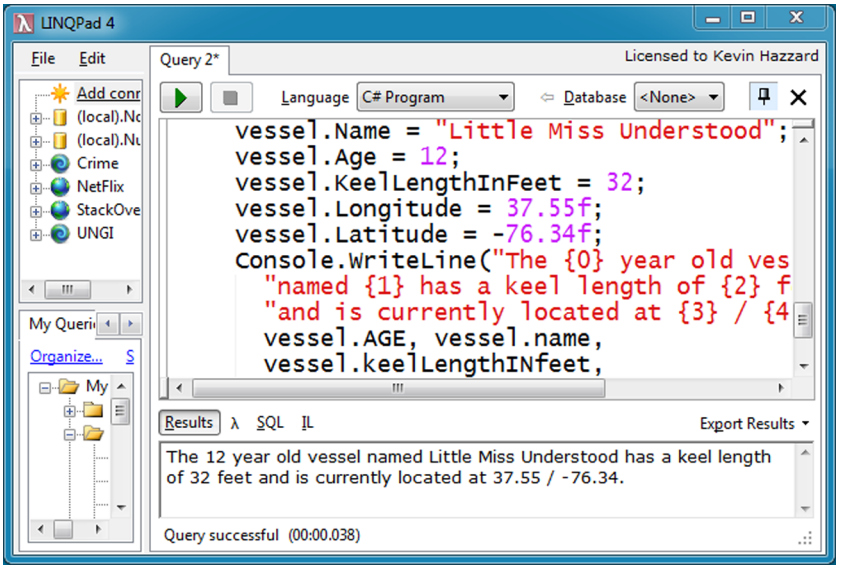

Figure 1.11 The metaprogramming class called `MyExpandoObject` in action

put them in balance. You discovered that you could use the synonyms *after-programming* and *beside-programming* to put the two basic ways in which metaprogramming is often implemented into contrast to enable future learning. Then you dove into common examples of metaprogramming that you may encounter working in and around the .NET Framework.

That's certainly a lot of material, but, to be honest, we've only been able to scratch the surface of the kinds of metaprogramming that can be done using the Microsoft .NET Framework. We made the examples in this chapter deliberately simple to get you started on the journey. We hope that these prototypes will serve you well as you continue your voyage through the remainder of the book.

.NET/PROGRAMMING

Metaprogramming IN .NET

Hazzard • Bock

When you write programs that create or modify other programs, you are metaprogramming. In .NET, you can use reflection as well as newer concepts like code generation and scriptable software. The emerging Roslyn project exposes the .NET compiler as an interactive API, allowing compile-time code analysis and just-in-time refactoring.

Metaprogramming in .NET is a practical introduction to the use of metaprogramming to improve the performance and maintainability of your code. This book avoids abstract theory and instead teaches you solid practices you'll find useful immediately. It introduces core concepts like code generation and application composition in clear, easy-to-follow language, and then it takes you on a deep dive into the tools and techniques that will help you implement them in your .NET applications.

What's Inside

- Metaprogramming concepts in plain language
- Creating scriptable software
- Code generation techniques
- The Dynamic Language Runtime

Written for readers comfortable with C# and the .NET framework—no prior experience with metaprogramming is required.

Kevin Hazzard is a Microsoft MVP, consultant, teacher, and developer community leader in the mid-Atlantic USA. **Jason Bock** is an author, Microsoft MVP, and the leader of the Twin Cities Code Camp.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/Metaprogrammingin.NET

Free eBook
SEE INSERT

"An excellent way to start fully using the power of metaprogramming."
—From the Foreword by Rockford Lhotka, Creator of the CSLA .NET Framework

"A thorough and comprehensive distillation of the vast array of code generation options in .NET."
—Harry Cummings, Softwire

"An extensive collection of *aha!* discoveries on developing applications beyond the mere compiler."
—William Lee, Qualcomm, Inc.

"An excellent reference ... insightful examples."
—Arun Noronha
Guardian Protection Services

ISBN 13: 978-1-617290-26-8
ISBN 10: 1-617290-26-2
54499
9 781617 290268

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]