CHAPTER 7

.NET-to-COM Interoperability—The Basics

The previous five chapters have exposed you to the core characteristics of the COM and .NET type systems. The remainder of this book addresses how these types can be expressed and manipulated across architectural boundaries. In this chapter, you are exposed to the key .NET-to-COM interoperability issues you are likely to encounter on a day-to-day basis (with some more exotic topics thrown in for good measure). For example, you investigate a number of ways to build interoperability assemblies (including “primary” interop assemblies), examine core IDL to .NET data type mappings, and understand how key COM data structures (interfaces, coclasses, enumerations) are expressed in terms of .NET. Along the way, you take a more detailed look at the types contained in the System.Runtime.InteropServices namespace (first introduced in Chapter 1). As you might expect, the materials presented here work as the backbone for more advanced topics found in the remainder of the text.

A High-Level Overview of .NET-to-COM Interoperability

As you have seen in Chapters 5 and 6, languages targeting the .NET runtime satisfy each pillar of object-oriented technology. For example, when you build an assembly using a given managed language, you are able to create classes that support any number of constructors, overloaded methods, and overridden members, and implement any optional interfaces. As well, the .NET platform makes use of a runtime garbage collector, which is responsible for freeing an object from the managed heap when it is no longer rooted in a given application.

In stark contrast, as you have seen in Chapters 2 through 4, COM types do not adhere to each and every pillar of OOP in the classic sense of the topic. For example, COM types are not created using class constructors, but rather using the
IClassFactory interface. In addition, COM classes are not allowed to define overloaded methods and cannot function as a base to other COM types (as COM has no support for classical inheritance). As far as lifetime management of a coclass is concerned, COM does not make use of a garbage-collected heap, but employs a strict reference counting scheme provided courtesy of IUnknown.

Given the fact that COM and .NET types have so little in common, you may have deep-rooted fears regarding interoperability issues. Ideally, a .NET client should be able to use a COM type with no concern for the mechanics of COM. For example, a managed client should be able to create the COM type using constructor semantics, derive new types from the COM wrapper class (given that .NET supports classic inheritance), and should not be required to obtain or release interface references (given that .NET does not demand the use of interface references). In a nutshell, as far as a .NET client is concerned, manipulating a COM type should look identical to the act of manipulating a native .NET type. For example:

```csharp
// COM classes should appear as .NET types.
MyComClass c = new MyComClass();
c.SomeMethod("Hello", 12);
```

Obviously, this cannot be achieved unless you have an intermediary that stands between the .NET client and the existing COM type. In short, what we need is a proxy that is in charge of transparently handling .NET-to-COM communications. To be sure, whenever a .NET application makes use of a legacy COM type, a proxy is created by the .NET runtime. Formally, this proxy is termed a Runtime Callable Wrapper (RCW).

In a similar vein, a COM client should be able to make use of a .NET type without concern for the mechanics of .NET. For example, COM clients should be able to activate a .NET class using CoCreateInstance(); directly call the members of IUnknown, IDispatch, and IClassFactory; and should assume the type is maintaining an internal reference count. When unmanaged code communicates with managed .NET types, a different sort of proxy called a COM Callable Wrapper (CCW) is used to translate COM requests into terms of .NET. Chapters 10 through 12 examine the process of COM-to-.NET interoperability. For the time being, let's concentrate on the role of the RCW.

**Understanding the Role of the RCW**

The RCW is a .NET object that is in charge of marshaling calls between a managed unit of code and a given COM type. While a managed client is making calls to a given COM type, the RCW intercepts each invocation, translates each incoming