

COM and .NET Interoperability

ANDREW TROELSEN

COM and .NET Interoperability
Copyright © 2002 by Andrew Troelsen

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-011-2
Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewers: Habib Heydarian, Eric Gunnerson
Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore,
Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Copy Editors: Anne Friedman, Ami Knox
Proofreaders: Nicole LeClerc, Sofia Marchant
Compositor: Diana Van Winkle, Van Winkle Design
Artist: Kurt Krames
Indexer: Valerie Robbins
Cover Designer: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit <http://www.springer-ny.com>.
Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit <http://www.springer.de>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710.
Phone: 510-549-5930, Fax: 510-549-5939, Email: info@apress.com, Web site: <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 12

COM-to-.NET Interoperability– Advanced Topics

The point of this chapter is to round out your knowledge of exposing .NET types to COM applications by examining a number of advanced techniques. The first major topic is to examine how .NET types can implement COM interfaces to achieve binary compatibility with other like-minded COM objects (a topic first broached in Chapter 7). Closely related to this topic is the process of defining COM types directly using managed code. Using this technique, it is possible to build a binary-compatible .NET type that does not directly reference a related interop assembly (and is therefore a bit more lightweight). As for the next major topic, you examine the process of building a customized version of `tlbexp.exe` while also addressing how to programmatically register interop assemblies at runtime. Finally, you wrap up by taking a deeper look at the .NET runtime environment and checking out how a COM client can be used to build a custom host for .NET types. In addition to being a very interesting point of discussion, you will see that a custom CLR host can simplify COM-to-.NET registration issues.

Changing Type Marshaling Using `MarshalAsAttribute`

Before digging into the real meat of this chapter, let's examine yet another interop-centric attribute. As you have seen, one nice thing about the `tlbexp.exe` utility is that it will always ensure the generated type information is [oleautomation] compatible. When you build COM interfaces that are indeed [oleautomation] compatible, you are able to ensure that all COM-aware languages can interact with the .NET type (as well as receive a free stub/proxy layer courtesy of the universal marshaler). Typically, if you have created a COM interface that is not [oleautomation] compatible, you have either (a) made a mistake, (b) are building a COM server you only intend to use from C++, or (c) wish to define a custom stub and proxy DLL for performance reasons.

Nevertheless, if you wish to create a managed method that is exposed to COM as a non-oleautomation-compatible entity, you are able to apply the MarshalAsAttribute type. The MarshalAs attribute can also be helpful when a single .NET type has the ability to be represented by multiple COM types. For example, a System.String could be marshaled to unmanaged code as a LPSTR, LPWSTR, LPTSTR, or BSTR. While the default behavior (System.String to COM BSTRs) is typically exactly what you want, the MarshalAsAttribute type can be used to expose System.String in alternative formats.

This attribute may be applied to a method return type, type member, and a particular member parameter. Applying this attribute is simple enough; however, the argument that is specified as a constructor parameter (UnmanagedType) is a .NET enumeration that defines a *ton* of possibilities. To fully understand the scope of the MarshalAs attribute, let's check out some core values of this marshal-centric enumeration. First up, Table 12-1 documents the key values of UnmanagedType that allow you to expose a System.String in various formats.

Table 12-1. String-Centric Values of UnmanagedType

String-Centric UnmanagedType Member Name	Meaning in Life
AnsiBStr	ANSI character string that is a length-prefixed single byte.
BStr	Unicode character string that is a length-prefixed double byte.
LPStr	A single-byte ANSI character string.
LPTStr	A platform-dependent character string, ANSI on Windows 98, Unicode on Windows NT. This value is only supported for Platform Invoke, and not COM interop, because exporting a string of type LPTStr is not supported.
LPWSTR	A double-byte Unicode character string.

To illustrate, assume you have a small set of .NET members that are defined as follows:

```
[ClassInterface(ClassInterfaceType.AutoDual)]
public class MyMarshalAsClass
{
    public MyMarshalAsClass(){}

    // String marshaling.
    public void ExposeAsLPStr
        ([MarshalAs(UnmanagedType.LPStr)]string s){}
    public void ExposeAsLPWSTR
        ([MarshalAs(UnmanagedType.LPWStr)]string s){}
}
```

Once processed by `tlbexp.exe`, you find the following COM IDL:

```
interface _MyMarshalAsClass : IDispatch
{
    [id(0x60020004)]
    HRESULT ExposeAsLPStr([in] LPSTR s);
    [id(0x60020005)]
    HRESULT ExposeAsLPWSTR([in] LPWSTR s);
};
```

Table 12-2 documents the key values of `UnmanagedType` that are used to expose `System.Object` types as various flavors of COM types.

Table 12-2. System.Object-Centric Values of UnmanagedType

Object-Centric UnmanagedType Member Name	Meaning in Life
IDispatch	A COM IDispatch pointer
IUnknown	A COM IUnknown pointer

If you extend the `MyMarshalAsClass` type to support the following members:

```
// Object marshaling.
public void ExposeAsIUnk
    ([MarshalAs(UnmanagedType.IUnknown)]object o){}
public void ExposeAsIDisp
    ([MarshalAs(UnmanagedType.IDispatch)]object o){}
```

you find the following COM type information:

```
[id(0x60020006)]
HRESULT ExposeAsIUnk([in] IUnknown* o);
[id(0x60020007)]
HRESULT ExposeAsIDisp([in] IDispatch* o);
```

`UnmanagedType` also provides a number of values that are used to alter how a .NET array is exposed to classic COM. Again, remember that by default, .NET arrays are exposed as COM `SAFEARRAY` types, which is typically what you require. For the sake of knowledge, however, Table 12-3 documents the key array-centric member of `UnmanagedType`.

Table 12-3. Array-Centric Value of UnmanagedType

Array-Centric UnmanagedType Member Name	Meaning in Life
LPArray	A C-style array

As you would guess, the following C# member definition:

```
// Array marshaling.  
public void ExposeAsCArray  
    ([MarshalAs(UnmanagedType.LPArray)]int[] myInts){}
```

results in the following IDL:

```
[id(0x60020008)]  
HRESULT ExposeAsCArray([in] long* myInts);
```

Finally, UnmanagedType defines a number of members that allow you to expose intrinsic .NET data types in various COM mappings. While many of these values are used for generic whole numbers, floating-point numbers, and whatnot, one item of interest is UnmanagedType.Currency. As you recall, the COM CURRENCY type is not supported under .NET and has been replaced by System.Decimal. Table 12-4 documents the key data-centric types.

Table 12-4. Data-Centric Values of UnmanagedType

Data Type-Centric UnmanagedType Member Name	Meaning in Life
AsAny	Dynamic type that determines the Type of an object at runtime and marshals the object as that Type.
Bool	4-byte Boolean value (true != 0, false = 0).
Currency	Used on a System.Decimal to marshal the decimal value as a COM currency type instead of as a Decimal.
I1	1-byte signed integer.
I2	2-byte signed integer.
I4	4-byte signed integer.
I8	8-byte signed integer.
R4	4-byte floating-point number.
R8	8-byte floating-point number.

Table 12-4. Data-Centric Values of *UnmanagedType* (continued)

Data Type-Centric <i>UnmanagedType</i> Member Name	Meaning in Life
<code>SysInt</code>	A platform-dependent signed integer. 4 bytes on 32-bit Windows, 8 bytes on 64-bit Windows.
<code>SysUInt</code>	Hardware natural-size unsigned integer.
<code>U1</code>	1-byte unsigned integer.
<code>U2</code>	2-byte unsigned integer.
<code>U4</code>	4-byte unsigned integer.
<code>U8</code>	8-byte unsigned integer.
<code>VariantBool</code>	2-byte OLE-defined Boolean value (true = -1, false = 0).

Again, the most useful of these data type-centric members of the *UnmanagedType* enumeration is the *UnmanagedType.Currency* value, given that .NET no longer supports the COM CURRENCY type. However, given that a *System.Decimal* provides the same storage, you can apply *MarshalAs* as follows:

```
// Exposing Decimal and Currency.
public void ExposeAsCURRENCY
    ([MarshalAs(UnmanagedType.Currency)]Decimal d){}
```

This results in the following IDL:

```
[id(0x60020008)]
HRESULT ExposeAsCURRENCY([in] CURRENCY d);
```

So, now that you have seen the various ways that the *MarshalAsAttribute* type can be configured, you may be wondering exactly when (or why) you may wish to alter the default interop marshaler. In reality, you typically won't need to alter the default marshaling behavior. The only time it might be beneficial on a somewhat regular basis is when you wish to expose .NET *System.Objects* as a specific COM interface type (*IUnknown* or *IDispatch*) or expose a *System.Decimal* as a legacy COM CURRENCY type.



CODE *The MyMarshalAsLibrary project is included under the Chapter 12 subdirectory.*

.NET Types Implementing COM Interfaces

Recall from Chapter 9 that if a COM coclass implements a COM-visible .NET interface, the coclass in question is able to achieve type compatibility with other like-minded .NET objects. The converse of this scenario is also true: .NET types can implement COM interfaces to achieve binary compatibility with other like-minded COM types. When a .NET programmer chooses to account for COM interfaces in his or her type implementations, there are two possible choices:

- Implement a *custom* COM interface.
- Implement a *standard* COM interface.

As you recall from Chapter 2, although a COM interface always boils down to the same physical form (a collection of pure virtual functions identified by a GUID), standard interfaces are predefined types (published by Microsoft). Furthermore, standard interfaces are already defined in terms of COM IDL, have a predefined GUID, and are recorded in the system registry. Custom interfaces, on the other hand, are authored by a COM developer during the course of a software development cycle. In this case, the programmer is the one in charge of describing the item in terms of COM IDL and registering the resulting type library (all of which is done automatically when using VB 6.0). When a .NET type implements a custom COM interface, the result is that a given COM client is able to interact with the .NET type as if it were a coclass adhering to a specific binary format.

On the other hand, if a .NET type implements a standard interface (such as IDispatch, IConnectionPointContainer, or ITypeInfo), it will be used as a customized *replacement* for the equivalent interface implemented by the CCW. To be sure, the chances that you will need to provide a customized implementation of an interface supported by the CCW are slim to none. Given this likelihood, I focus solely on the process of defining managed versions of custom COM interfaces.

Defining Custom COM Interfaces

Before you can examine how to implement custom COM interfaces on a .NET type, you first need the IDL descriptions of the interfaces themselves. As you will see later in this chapter, it is possible to build a binary-compatible .NET type without a formal COM type description; however, for this example, assume you have created an ATL in-proc COM server (AnotherAtlCarServer). This COM server defines a coclass (CoTruck) by implementing two simple interfaces named

IStartable and IStoppable. Here is the relevant IDL (if you need a refresher on building COM servers with ATL, see Chapter 3):

```
[object,
uuid(7FE41805-124B-44AE-BEAE-C3491E35372B),
oleautomation,
helpstring("IStartable Interface"),
pointer_default(unique)]
interface IStartable : IUnknown
{ HRESULT Start(); };

[object,
uuid(B001A308-8D66-4d23-84A4-B67615646ABB),
oleautomation,
helpstring("IStartable Interface"),
pointer_default(unique)]
interface IStoppable : IUnknown
{ HRESULT Break(); };

[uuid(7B69AEB6-F0B7-46BB-8AD4-1CACD1EA5AE9),
version(1.0),
helpstring("AnotherAtlCarServer 1.0 Type Library")]
library ANOTHERATLCARSERVERLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [uuid(862C5338-8AD7-43A3-A9A7-F21B145D61D0),
    helpstring("CoTruck Class")]
    coclass CoTruck
    {
        [default] interface IStartable;
        interface IStoppable;
    };
};
```

The implementation of the CoTruck::Start() and CoTruck::Break() methods simply triggers a Win32 MessageBox() API to inform the caller which object has been told to do what:

```
STDMETHODIMP CCoTruck::Start()
{
    MessageBox(NULL, "The truck as started.",
        "CoTruck::Start() Says:", MB_OK);
    return S_OK;
}
```

```

STDMETHODIMP CCoTruck::Break()
{
    MessageBox(NULL, "The truck as stopped.",
        "CoTruck::Start() Says:", MB_OK);
    return S_OK;
}

```

That's it. Go ahead and compile this ATL project to ensure that this COM server is properly recorded in the system registry.



CODE *The AnotherAtlCarServer project can be found under the Chapter 12 subdirectory.*

Building and Deploying the Interop Assembly

Now that you have a COM server defining a set of custom interfaces, you need to transform the COM type information into terms of .NET metadata. Thus, assuming you have a valid *.snk file, configure a strongly named interop assembly using `tlbimp.exe` as follows:

```

tlbimp AnotherAtlCarServer.dll /out:interop.AnotherAtlCarServer.dll
/keyfile:theKey.snk

```

Finally, deploy this interop assembly into the GAC (Figure 12-1).

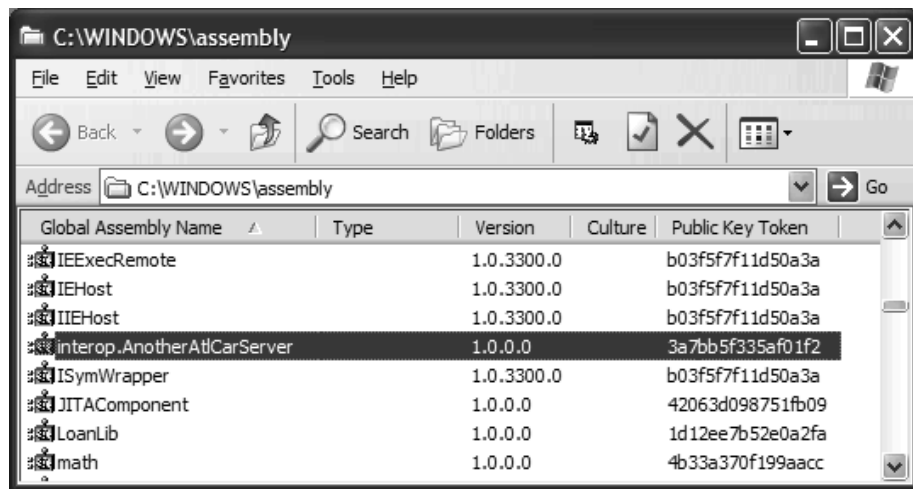


Figure 12-1. Another machine-wide interop assembly



CODE *The interop assembly for AnotherAltCarServer.dll is included under the Chapter 12 subdirectory.*

Building a Binary-Compatible C# Type

To illustrate building a binary-compatible .NET type, let's create a new C# Code Library that defines a simple class (DotNetLawnMower) that supports both interfaces. First, add a reference to `interop.AnotherAltCarServer.dll`, and for simplicity, configure this type to be exposed to COM as an `AutoDual` class interface:

```
namespace BinaryCompatibleDotNetTypeServer
{
    // This .NET class supports two COM interfaces.
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class DotNetLawnMower: IStartable, IStoppable
    {
        public DotNetLawnMower(){}
    }
}
```

Now that `DotNetLawnMower` has defined support for `IStartable` and `IStoppable`, and you are obligated to flesh out the details of the `Start()` and `Break()` methods. While you could manually type the definitions of each inherited member, you do have a shortcut. The Visual Studio .NET IDE supports an integrated wizard that automatically generates stub code for an implemented interface. However, the manner in which you interact with this tool depends on your language of choice. Here, in your C# project, you activate this tool by right-clicking a supported interface using Class View (Figure 12-2).

Again, the implementation of each member is irrelevant for this example, so just set a reference to `System.Windows.Forms.dll` and call `MessageBox.Show()` in an appropriate manner:

```
public void Start()
{
    MessageBox.Show("Lawn Mower starting..." ,
        "DotNetLawnMower says:");
}
public void Break()
{
    MessageBox.Show("Lawn Mower stopping..." ,
        "DotNetLawnMower says:");
}
```

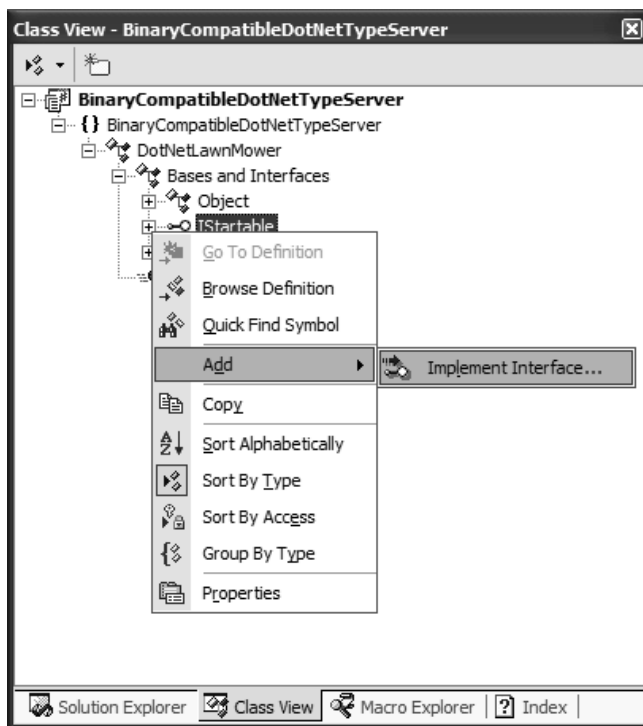


Figure 12-2. The C# IDE Implement Interface Wizard

Because this .NET class library is to be used by a classic COM client, you will want to deploy this binary as a shared assembly. Thus, be sure to set the assembly's version (1.0.0.0 will do) and specify a valid *.snk file. Once you have done so, deploy this assembly to the GAC.



CODE *The BinaryCompatibleDotNetTypeServer project is included under the Chapter 12 subdirectory.*

Building a Binary-Compatible VB .NET Type

Any managed language has the ability to implement COM interfaces, provided they have access to the interface descriptions. To further highlight the process, assume you have a VB .NET Code Library that defines a type named UFO. The UFO type is able to be started and stopped (presumably) and thus wishes to implement the COM interfaces defined in the ATL server. Once you set a reference

to the interop assembly and define support for each interface (via the Implements keyword), the VB .NET IDE provides a simple shortcut to automatically build stubs for each method. Simply select the name of the supported interface from the left drop-down list and the name of the method from the right drop-down list (Figure 12-3).

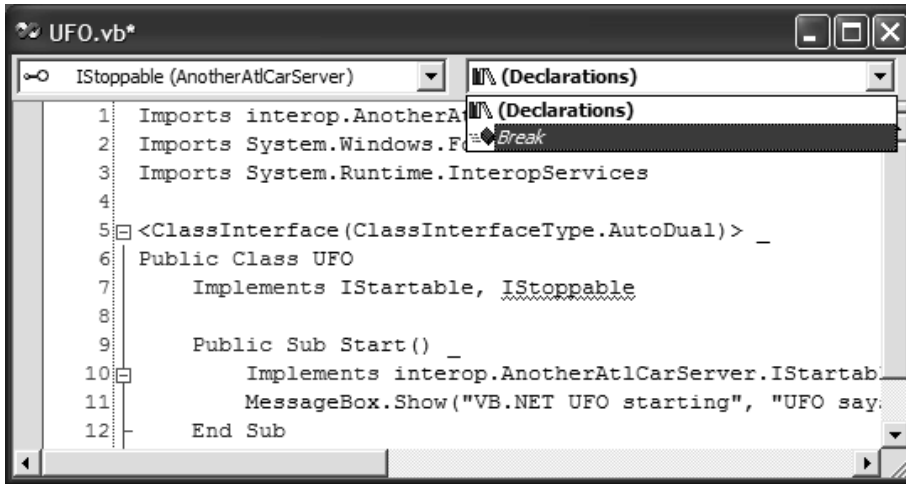


Figure 12-3. The VB .NET IDE Implement Interface Wizard

Here is the complete VB .NET definition of UFO, which also makes use of an AutoDual class interface (again, be sure to assign a strong name to the assembly and deploy this assembly to the GAC):

```
<ClassInterface(ClassInterfaceType.AutoDual)> _
Public Class UFO
    Implements IStartable, IStoppable
    Public Sub Start() _
        Implements ANOTHERATLCARSERVERLib.IStartable.Start
        MessageBox.Show("VB.NET UFO starting", "UFO says:")
    End Sub
    Public Sub Break() Implements ANOTHERATLCARSERVERLib.IStoppable.Break
        MessageBox.Show("VB.NET UFO stopping", "UFO says:")
    End Sub
End Class
```



CODE The *BinaryCompatibleVbNetTypeServer* project is included under the Chapter 12 directory.

Registering the .NET Assemblies with COM

So, to recap the story thus far, at this point you have three objects (CoTruck, LawnMower, and UFO). Each has been created in a specific language (C++, C#, or VB .NET) using two different architectures (COM and .NET) that implement the same two COM interfaces. Furthermore, the interop assembly for the AnotherAtlCarServer.dll COM server and the strongly named .NET assemblies have been deployed to the GAC. Like any COM-to-.NET interaction, however, you must generate COM type information (and register the contents) for each native .NET assembly using regasm.exe. Thus, from the command line, run regasm.exe against both of your .NET assemblies. For example:

```
regasm BinaryCompatibleVbNetTypeServer.dll /tlb
```

Building a VB 6.0 COM Client

Now that each .NET assembly has been configured to be reachable by a COM client, the final step of this example is to build an application that interacts with each object in a binary-compatible manner. While you are free to use any COM-aware programming language, I'll make use of a VB 6.0 Standard EXE project that interacts with each type. The big picture is illustrated in Figure 12-4.

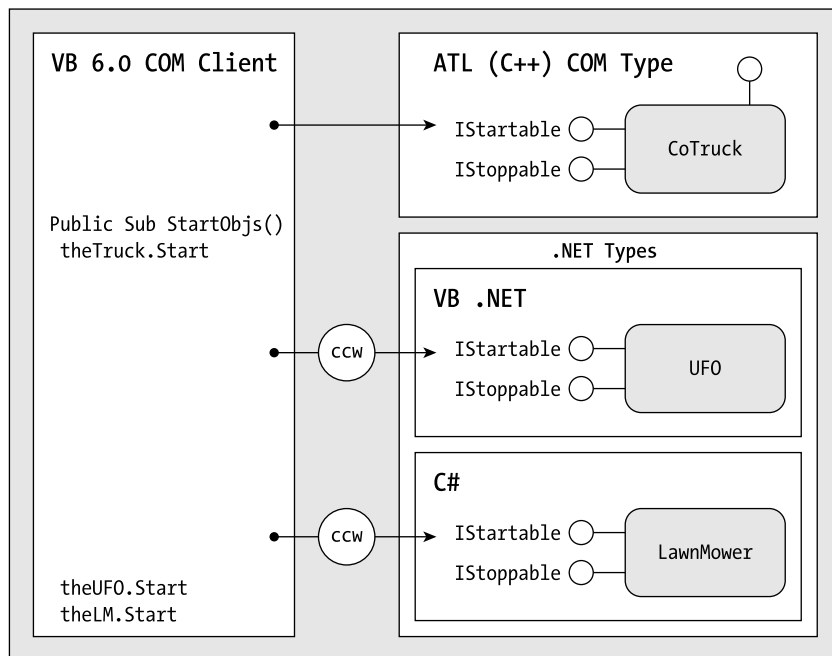


Figure 12-4. Behold, the power of interface-based programming.

The first step (of course) is to set a reference to each type library (Figure 12-5).

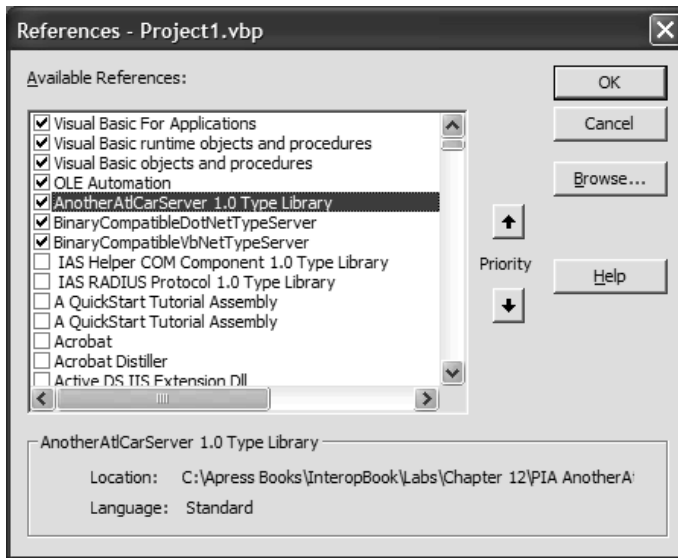


Figure 12-5. Referencing the COM type information

Just to keep things interesting, you will add one additional refinement to the scenario suggested by Figure 12-4. Rather than declaring three Form-level member variables of type UFO, LawnMower, and CoTruck, let's make use of a VB 6.0 Collection type to contain each item (as this will better illustrate the interface-based polymorphism of semantically gluing the types together). Thus, if the main Form has two Button types that start and stop each item in the collection, you are able to author the following VB 6.0 code:

```
Option Explicit
Private theObjs As Collection

' Loop through the collection
' and start everything using IStartable.
Private Sub btnStartObjs_Click()
    Dim temp As IStartable
    Dim i As Integer
    For i = 0 To theObjs.Count - 1
        Set temp = theObjs(i + 1)
        temp.Start
    Next
End Sub
```

```

' Loop through the collection and
' stop everything using IStoppable.
Private Sub btnStopObjs_Click()
    Dim temp As IStoppable
    Dim i As Integer
    For i = 0 To theObjs.Count - 1
        Set temp = theObjs(i + 1)
        temp.Break
    Next
End Sub

' Fill the collection with some
' binary compatible types.
Private Sub Form_Load()
    Set theObjs = New Collection
    theObjs.Add New CoTruck
    theObjs.Add New UFO
    theObjs.Add New DotNetLawnMower
End Sub
' ATL type.
' VB .NET type.
' C# type

```

Notice that you are able to communicate with each type using the custom COM interfaces defined in the original ATL server (thus the binary compatibility nature of the example). If you were to run the client application, you would see a series of message boxes pop up as the types in the collection were manipulated.



CODE *The Vb6COMCompatibleClient project is included under the Chapter 12 subdirectory.*

Defining COM Interfaces Using Managed Code

Although the previous example did indeed allow the .NET types to implement existing COM interfaces, you had to jump through a few undesirable hoops during the process. First, each .NET code library was required to obtain the type information of IStoppable and IStartable via tlbimp.exe. This of course results in an [assembly extern] listing in each assembly manifest. Given this, each .NET assembly now depends on the presence of the interop assembly on the target machine. If the interop assembly is not present and accounted for, the .NET consumer is unable to find the correct metadata and it becomes woefully binary-incompatible with other like-minded COM types.

When you think about it, the C# LawnMower and VB .NET UFO types never needed to directly interact with the CoTruck. All these projects required were the

managed definitions of the raw COM interfaces. To simplify the process, you could have defined `IStartable` and `IStoppable` (using managed code) directly within the .NET assemblies. In this way, your .NET assemblies are no longer tied to an interop assembly and are still binary compatible!

To illustrate, let's see a simple example. Assume you have yet another C# Code Library (ManagedComDefs) that contains a simple class named `DvdPlayer`. Given that DVD players are also startable and stoppable, our goal is to achieve binary compatibility with the `CoTruck`, `UFO`, and `LawnMower` types, *without* referencing the `interop.AnotherAltCarServer.dll` assembly.

When you define COM interfaces directly within managed code, each and every interface must be attributed with the `ComImportAttribute`, `GuidAttribute`, and `InterfaceTypeAttribute` types. Therefore, all your managed interfaces look something like the following:

```
// Some binary compatible COM interface
// defined in managed code.
[ComImport, Guid("<IID>"),
InterfaceType(ComInterfaceType.<type of COM interface>)]
public interface SomeBinaryCompatibleInterface
{ // Members...}
```

The `ComImportAttribute` type is simply used to identify this type as a COM entity when exposed to a COM client. Obviously, the value of the `GuidAttribute` type must be identical to the original IDL IID. As for the `InterfaceTypeAttribute`, you are provided with the following related enumeration to mark the representation of the COM interface you are describing:

```
public enum System.Runtime.InteropServices.ComInterfaceType
{
    InterfaceIsDual,
    InterfaceIsIDispatch,
    InterfaceIsIUnknown
}
```

The `ComInterfaceType` value passed into the `InterfaceTypeAttribute` is used by the .NET runtime to determine how to build the correct vtable for the unmanaged COM interface (more on this tidbit in just a moment). Recall that the `IStartable` and `IStoppable` interfaces were defined in IDL as follows:

```
[object,
uuid(7FE41805-124B-44AE-BEAE-C3491E35372B),
oleautomation,
helpstring("IStartable Interface"),
pointer_default(unique)]
interface IStartable : IUnknown
{ HRESULT Start(); };
```

```
[object,
uuid(B001A308-8D66-4d23-84A4-B67615646ABB),
oleautomation,
helpstring("IStartable Interface"),
pointer_default(unique)]
interface IStoppable : IUnknown
{ HRESULT Break();};
```

Looking at these interface types, it should be clear that the COM-to-.NET data type, type, and type member conversion rules still apply (for example, `System.String` becomes `BSTR` and `whatnot`). In this case, you are happy to find that `Start()` and `Break()` take no parameters, and therefore can be defined in terms of C# in a rather straightforward manner. Here is the complete code behind the binary-compatible `DvdPlayer`:

```
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace ManuallyInterfaceDefsServer
{
    // Managed definition of IStartable.
    [ComImport,
    Guid("7FE41805-124B-44AE-BEAE-C3491E35372B"),
    InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
    interface IStartable { void Start(); };

    // Managed definition of IStoppable.
    [ComImport,
    Guid("B001A308-8D66-4d23-84A4-B67615646ABB"),
    InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
    interface IStoppable { void Break();};

    // A binary compatible DVD player!
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class DvdPlayer : IStartable, IStoppable
    {
        public DvdPlayer(){}
        public void Start()
        { MessageBox.Show("Staring movie...", "DvdPlayer");}

        public void Break()
        {MessageBox.Show("Stopping movie...", "DvdPlayer");}
    }
}
```

Once you compile this .NET assembly, if you (a) deploy this assembly into the GAC and (b) export the metadata to a COM *.tlb file via regasm.exe, you would be able to set a reference to the exported *.tlb file and update the VB 6.0 COM client as follows:

```
' Add a DVD player into the mix.
Private Sub Form_Load()
    Set theObjs = New Collection
    theObjs.Add New CoTruck
    theObjs.Add New UFO
    theObjs.Add New DotNetLawnMower
    theObjs.Add New DvdPlayer
End Sub
```

Sure enough, you are able to make use of IStartable and IStoppable of the DvdPlayer as expected (Figure 12-6).



Figure 12-6. Using the binary-compatible DvdPlayer



CODE The ManagedComDefs project is included under the Chapter 12 subdirectory.

Selected Notes on Manually Defining COM Interfaces Using Managed Code

The previous example was quite straightforward, given that the interfaces you defined were IUnknown-derived entities (thus no DISPIDs) and contained methods with no parameters (thus no [in], [out], or [out, retval] attributes to worry about). As you might expect, if you attempt to manually pound out the details of more complex COM interfaces, you need to apply additional .NET attributes. Furthermore, it is possible (although not altogether likely) that you might need to define other COM types (enums, structures, coclasses) in terms of managed code. To be sure, if the COM type you are attempting to become binary-compatible with

has been defined in terms of COM IDL, you will never need to manually define COM types other than the occasional interface. Even then, if the dependency on a related interop assembly is acceptable, you will not need to bother to do this much.

However, there may be some (hopefully) rare cases in which you will need to manually define COM interfaces via managed code. For example, in C++, it is possible to build a COM class supporting a set of COM interfaces without the use of IDL. Given that the `midl.exe` compiler simply regards IDL interfaces as a collection of C++ pure virtual functions, a C++ developer could choose to define the pure virtual functions directly in terms of C++. The obvious downfall to this approach is that the programmer has effectively created a COM server that can only be used by other C++ clients. If a .NET programmer wished to build a binary-compatible type using an interface described in raw C++, it would demand creating a managed definition of the COM type, given that the COM type library (and thus the interop assembly) doesn't exist!

The process of manually defining a COM type in terms of managed code can be very helpful if you require only a subset of items defined in the type library, or if you need to somehow modify the COM type to work better from a managed environment. As you may recall from Chapter 9, it is possible to crack open an interop assembly and tweak the internal metadata. The same result can often be achieved by directly implementing the COM types using managed code (not to mention, it can be achieved in a much simpler manner). Given these possibilities, let's walk through an extended example.

Manually Defining COM Atoms: An Extended Example

The next COM server you examine (`AtlShapesServer`) defines a coclass (`CoHexagon`) that supports a single [dual] interface (`IDrawable`). `IDrawable` defines a small set of methods, one of which makes use of a custom COM enumeration. Here is the complete IDL:

```
typedef enum SHAPECOLOR
{
    RED, PINK, RUST
}SHAPECOLOR;

[object,
uuid(B1691C03-7EA8-4DAB-86CC-7D6CD859671A),
dual,
pointer_default(unique)]
interface IDrawable : IDispatch
{
    [id(1), helpstring("method Draw")]
    HRESULT Draw([in] int top, [in] int left, [in] int bottom, [in] int right);
```

```

[id(2), helpstring("method SetColor")]
HRESULT SetColor([in] SHAPECOLOR c);
};

[uuid(95FBF6E3-1B03-4904-A5D3-C77A02785F9A),
version(1.0)]
library ATLSHAPESSERVERLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    [uuid(204F9A4B-4D22-451B-BE2F-338F2917E7F5)]
    coclass CoHexagon
    {
        [default] interface IDrawable;
    };
};

```

Defining the Dual Interface (and SHAPECOLOR Enum) Using C#

When you describe a [dual] interface in terms of managed code, you obviously need to supply `ComInterfaceType.InterfaceIsDual` to the `InterfaceTypeAttribute` constructor (given the IDL definition). Additionally, you are required to supply the correct DISPID values for each member. This alone is not too earth-shattering. However, recall that the `IDrawable` interface defines two members:

```

interface IDrawable : IDispatch
{
    [id(1)]
    HRESULT Draw([in] int top, [in] int left, [in] int bottom, [in] int right );
    [id(2)] HRESULT SetColor([in] SHAPECOLOR c);
};

```

Now, as you are aware, COM interfaces are used to construct a vtable for the implementing coclass. A vtable is little more than a listing of addresses that point to the correct function implementation. Given that COM is so dependent on a valid vtable, you must understand that it is *critical* that you define the methods of a managed COM interface in the same order as found in the original IDL (or C++ header file). If you do not, you are most certainly not binary-compatible. Given this, here is the definition of `IDrawable` (and the related `SHAPECOLOR` enum) in terms of C#:

```

// Defining COM enums in managed
// code is painless.
public enum SHAPECOLOR
{ RED, PINK, RUST };

```

```
// The managed version of IDrawable.
[ComImport,
Guid("B1691C03-7EA8-4DAB-86CC-7D6CD859671A"),
InterfaceType(ComInterfaceType.InterfaceIsDual)]
interface IDrawable
{
    [DispId(1)]
    void Draw([In] int top, [In] int left,
              [In] int bottom, [In] int right);
    [DispId(2)]
    void SetColor([In] SHAPECOLOR c);
};
```

Here, you are making use of the `DispIdAttribute` type to define the DISPIDs of each interface. As you are most likely able to figure out, it is critical that the values supplied to each `DispIdAttribute` match the values of the original COM IDL. If you build a .NET type that is binary compatible with the `IDrawable` interface, you might author the following:

```
[ClassInterface(ClassInterfaceType.AutoDual)]
public class Circle: IDrawable
{
    public Circle(){}
    public void Draw(int top, int left, int bottom, int right)
    {
        MessageBox.Show(String.Format("Top:{0} Left:{1} Bottom:{2} Right{3}",
            top, left, bottom, right));
    }

    public void SetColor(SHAPECOLOR c)
    {
        MessageBox.Show(String.Format("Shape color is {0}", c.ToString()));
    }
}
```

If you view the .NET metadata descriptions of the `IDrawable` interface using `ILDasm.exe`, you find that the `ComImportAttribute` type is not listed directly with the `GuidAttribute` and `InterfaceType` values. The essence of the `ComImport` attribute is cataloged, however, using the `[import]` tag on the interface definition:

```
.class interface private abstract auto ansi import IDrawable
{
    ...
} // end of class IDrawable
```

Assuming you have processed this .NET assembly using regasm.exe, you would now be able to build an unmanaged COM client that interacts with the ATL CoHexagon and C# Circle type in a binary-compatible manner (using either early or late binding).

So, to wrap up the topic of building binary-compatible .NET types, understand that just because you can define COM interfaces in managed code does not mean you have to. Typically speaking, you simply set a reference to the correct interop assembly. However, if you are building a managed application that needs to communicate to a COM class using an interface for which there is no interop assembly, it is often necessary to manually define the type in terms of managed code (recall, for example, your C# COM type library viewer in Chapter 4).

Interacting with Interop Assembly Registration

As you recall from Chapter 2, a COM in-process server defines two function exports that are called by various installation utilities (regsvr32.exe) to register or unregister the necessary registry entries. As well, when a .NET assembly is to be used by COM, the system registry must be updated using regasm.exe to effectively fool the COM runtime. As you have seen, regasm.exe catalogs the correct entries automatically. What happens, however, if you want to insert custom bits of information into the registry during the default process performed by regasm.exe?

The System.Runtime.InteropServices namespace defines two attributes for this very reason. To illustrate, assume you have a new C# code library (CustomRegAsm) that defines some number of types. When you want to allow regasm.exe to trigger a custom method during the registration process, simply define a static (or Shared in VB .NET) method that is adorned with the ComRegisterFunctionAttribute. Likewise, if you wish to provide a hook for the unregistration process, define a second static member that supports the ComUnregisterFunctionAttribute. For example:

```
public class SomeClass
{
    public SomeClass(){}

    // This method will be called when
    // regasm.exe is run against this assembly.
    [ComRegisterFunction()]
    private static void CustomReg(Type t)
    {
        MessageBox.Show(String.Format("Registering {0}",
            t.ToString()));
    }
}
```

```

// This method will be called when
// regasm.exe is run against this
// assembly using the /u flag.
[ComUnregisterFunction()]
private static void CustomUnReg(Type t)
{
    MessageBox.Show(String.Format("Registering {0}",
        t.ToString()));
}
}

```

As you can see, the target methods must provide a single argument of type `System.Type`, which represents the current type in the assembly being registered for use by COM. As you might guess, `regasm.exe` passes in this parameter automatically.

Inserting Custom Registration Information

So, when might you need to interact with the assembly's registration process? Assume that you wish to record the date and time on which a given .NET assembly has been registered on a given user's machine. To do this, you can make use of the `Microsoft.Win32` namespace, which contains a small number of types that allow you to programmatically read from and write to the system registry. For example, the `CustomReg()` and `CustomUnReg()` methods could be retrofitted as follows:

```

[ComRegisterFunction()]
private static void CustomReg(Type t)
{
    RegistryKey k =
        Registry.CurrentUser.CreateSubKey(@"Software\Intertech\CustomRegAsm");
    k.SetValue("InstallTime", DateTime.Now.ToShortTimeString());
    k.SetValue("InstallDate", DateTime.Now.ToShortDateString());
    k.Close();
}

[ComUnregisterFunction()]
private static void CustomUnReg(Type t)
{
    Registry.CurrentUser.DeleteSubKey(@"Software\Intertech\CustomRegAsm");
}

```

When you register this .NET assembly via `regasm.exe`, you find the following information inserted under `HKEY_CURRENT_USER\Software\Intertech\CustomRegAsm` (Figure 12-7).

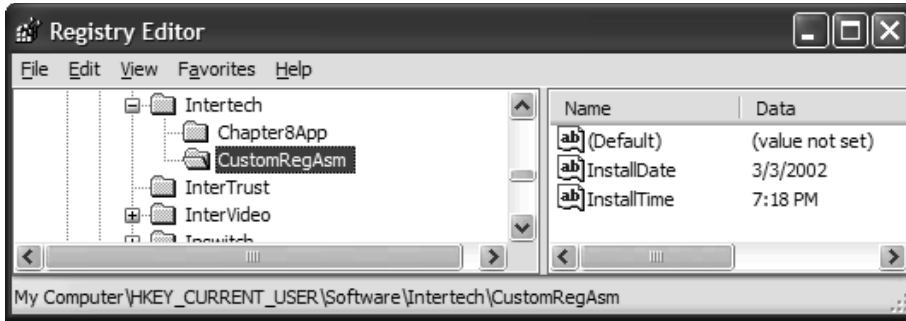


Figure 12-7. Getting involved with assembly registration

If you specify the /u flag, the information is correctly removed from the same subkey.



CODE The *CustomRegAsm* project is included under the Chapter 12 subdirectory.

Programmatically Converting Assemblies to COM Type Information

Recall from Chapter 9 that the `System.Runtime.InteropServices.TypeLibConverter` type allows you to programmatically convert COM *.tlb files into .NET interop assemblies. As mentioned at that time, this same class provides the ability to convert .NET assemblies into COM type information programmatically. Given this, let's examine the process of building a customized version of the `tlbexp.exe` command line utility (which as you will see looks much like the customized `tlbimp.exe` utility).

To begin, assume that you have a new C# console application named `MyTypeLibExporter`. The goal here is to allow the user to enter the path to a given .NET assembly and, using `TypeLibConverter`, to build a corresponding COM type library. The application's `Main()` method prompts for the assembly to export and passes this string into a static helper function named `GenerateTLBFromAsm()`.

Once the *.tlb file has been generated (and stored in the application directory), the user is again prompted to determine if the .NET assembly should be registered for use by COM. If the user wishes to do so, make use of the `System.Runtime.InteropServices.RegistrationServices` type. Here then, is the complete implementation behind `Main()`:

```

static void Main(string[] args)
{
    // Get the path to the assembly.
    Console.WriteLine("Please enter the path to the .NET binary");
    Console.WriteLine(@"Example: C:\MyStuff\Blah\myDotNetServer.dll");
    Console.Write("Path: ");
    string pathToAssembly = Console.ReadLine();

    // Generate type lib for this assembly.
    UCOMITypeLib i = GenerateTLBFromAsm(pathToAssembly);

    // Ask if user wants to register this server with COM.
    int regValue;
    Console.WriteLine("Would you like to register this .NET library with COM?");
    Console.Write("1 = yes or 0 = no ");
    regValue = Console.Read();

    if(regValue == 1)
    {
        RegistrationServices rs = new RegistrationServices();
        Assembly asm = Assembly.LoadFrom(pathToAssembly);
        rs.RegisterAssembly(asm, AssemblyRegistrationFlags.None);
        Console.WriteLine(".NET assembly registered with COM!");
    }
}

```

As you can see, the real workhorse of this application is the `GenerateTLBFromAsm()` helper function. Like the custom `tlbimp.exe` application you created earlier in this text, the `TypeLibConverter.ConvertAssemblyToTypeLib()` method requires you to pass in an instance of a class that will be called by the `TypeLibConverter` type to resolve references to additional assemblies as well as general reporting information. In this case, however, the class type is required to adhere to the behavior defined by `ITypeLibExporterNotifySink`:

```

public interface ITypeLibExporterNotifySink
{
    void ReportEvent(ExporterEventKind eventKind,
        int eventCode, string eventMsg);
    object ResolveRef(System.Reflection.Assembly assembly);
}

```

Much like the `ITypeLibImporterNotifySink` interface seen in Chapter 9, the implementation of `ITypeLibExporterNotifySink` delegates the work of resolving the referenced assembly to the static `MyTypeLibExporter.GenerateTLBFromAsm()` helper function:

// The callback object.

```
internal class ExporterNotifierSink : ITypeLibExporterNotifySink
{
    public void ReportEvent(ExporterEventKind eventKind,
        int eventCode, string eventMsg)
    {
        Console.WriteLine("Event reported: {0}", eventMsg);
    }

    public object ResolveRef(System.Reflection.Assembly assembly)
    {
        // If the assembly we are converting references another assembly,
        // we need to generate a *tlb for it as well.
        string pathToAsm;
        Console.WriteLine("MyTypeLibExporter encountered an assembly");
        Console.WriteLine("which referenced another assembly...");
        Console.WriteLine("Please enter the location to {0}", assembly.FullName);
        pathToAsm = Console.ReadLine();
        return MyTypeLibExporter.GenerateTLBFromAsm(pathToAsm);
    }
}
```

Before you see the details behind `MyTypeLibExporter.GenerateTLBFromAsm()`, you need to define some low-level COM types in terms of managed code. As you may recall from Chapter 4, when you create a custom COM type library generation tool, you need to call `ICreateTypeLib.SaveAllChanges()` to commit the type information to file. The trouble, however, is that the `System.Runtime.InteropServices` namespace does not define a managed equivalent of this method. Thus, using the tricks presented in this chapter, here is a makeshift version. It is makeshift in that I am representing the `ICreateTypeInfo` interface returned from the `CreateTypeInfo()` method (also recall from Chapter 4 that the `ICreateTypeInfo` interface is *huge*).

```
[ComImport,
GuidAttribute("00020406-0000-0000-C000-000000000046"),
InterfaceTypeAttribute(ComInterfaceType.InterfaceIsIUnknown),
ComVisible(false)]
internal interface UCOMICreateTypeLib
{
    // IntPtr is a hack to avoid having
    // to define ICreateTypeInfo (which is HUGE).
    IntPtr CreateTypeInfo(string name, TYPEKIND kind);
    void SetName(string name);
    void SetVersion(short major, short minor);
    void SetGuid(ref Guid theGuid);
    void SetDocString(string doc);
}
```

```

void SetHelpFileName(string helpFile);
void SetHelpContext(int helpCtx);
void SetLcid(int lcid);
void SetLibFlags(uint flags);
void SaveAllChanges();
}

```

Now that you have a managed definition for use by the `GenerateTLBFromAsm()` method, you can flesh out the details as follows:

```

public static UCOMITypeLib GenerateTLBFromAsm(string pathToAssmebly)
{
    UCOMITypeLib managedITypeLib = null;
    ExporterNotiferSink sink = new ExporterNotiferSink();

    // Load the assembly to convert.
    Assembly asm = Assembly.LoadFrom(pathToAssmebly);
    if (asm != null)
    {
        try
        {
            // Create name of type library based on .NET assembly.
            string tlbname = asm.GetName().Name + ".tlb";

            // Convert the assembly.
            ITypeLibConverter TLBConv = new TypeLibConverter();
            managedITypeLib = (UCOMITypeLib)
                TLBConv.ConvertAssemblyToTypeLib(asm, tlbname, 0, sink);

            // Save the type library to file.
            try
            {
                UCOMICreateTypeLib managedICreateITypeLib =
                    (UCOMICreateTypeLib)managedITypeLib;
                managedICreateITypeLib.SaveAllChanges();
            }
            catch (COMException e)
            {
                throw new Exception("Error saving the type lib : "
                    + e.ErrorCode.ToString("x"));
            }
        }
        catch (Exception e)
        {
            throw new Exception("Error Converting assembly" + e);
        }
    }
    return managedITypeLib;
}

```

I'd bet the details of this method are not too shocking by this point in the text. Basically, you load the assembly based on the incoming string parameter and define a name for the type library you are creating using the assembly's name as a base. Once you have an Assembly reference, you call `ConvertAssemblyToTypeLib()` and specify the reference to the loaded assembly, the name of the type library to create, any additional flags (or in our case, a lack thereof), and an instance of the sink implementing `ITypeLibExporterNotifySink`.

The `System.Object` that is returned from `ConvertAssemblyToTypeLib()` actually represents a reference to the in-memory representation of the COM type information, which is to say, an `UCOMTypeLib` interface. Once you cast this type into your version of the unmanaged `ICreateTypeLib` type, you are able to call `SaveAllChanges()` to commit the information to file.

Do note that your `GenerateTLBFromAsm()` helper function returns the `UCOMTypeLib` interface to the caller. You really don't need to do so. Using this type, however, you could interact with the internal COM types defined by this type library (as illustrated in Chapter 4). In any case, this wraps up the implementation of your custom `tlbexp.exe` utility. Figure 12-8 shows a test drive by importing the `CSharpCarLibrary.dll` assembly created in Chapter 6.

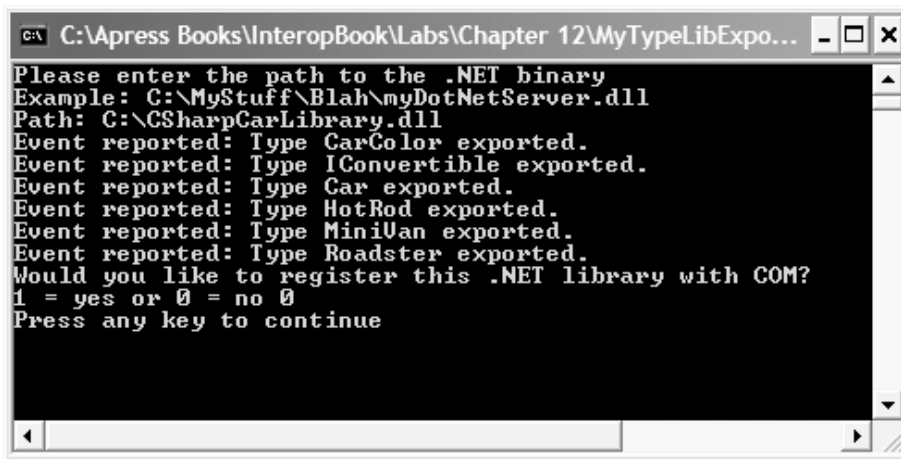


Figure 12-8. Exporting `CSharpCarLibrary.dll`

If you opened the generated `*.tlb` file using `oleview.exe`, you would find the COM definitions for each .NET type (Figure 12-9).

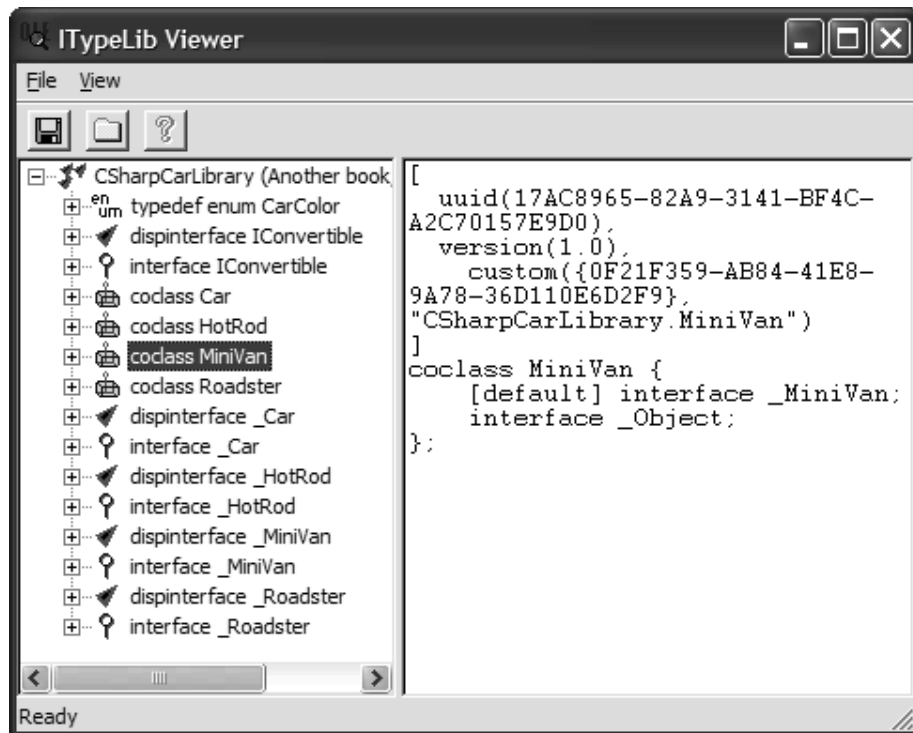


Figure 12-9. The exported *.tlb file



CODE The *MyTypeLibExporter* project is included under the Chapter 12 subdirectory.

Hosting the .NET Runtime from an Unmanaged Environment

The final topic of this chapter is a rather intriguing one: building a custom host for the .NET runtime (aka the CLR). Like all things under the .NET platform, the runtime engine is accessible using a set of managed types. In this case, the assembly in question is *mscorlib.dll* (where “ee” stands for execution engine). It may surprise you to know that when you install the .NET platform, you receive a corresponding *.tlb file for *mscorlib.dll* (*mscorlib.tlb*) that has been properly configured in the system registry.

Because the content of `mscorlib.dll` has been expressed in terms of COM metadata, it is possible to build a custom host using any COM-aware programming language (within the realm of the language's limitations). Do understand that regardless of which COM language you choose, when you make use of `mscorlib.tlb`, you are also required to reference the related `mscorlib.tlb` file. For the example that follows, assume that you have created a new Standard EXE application using VB 6.0. This assumption aside, set a reference to each *.tlb file using the IDE's Project | References menu option (see Figure 12-10).

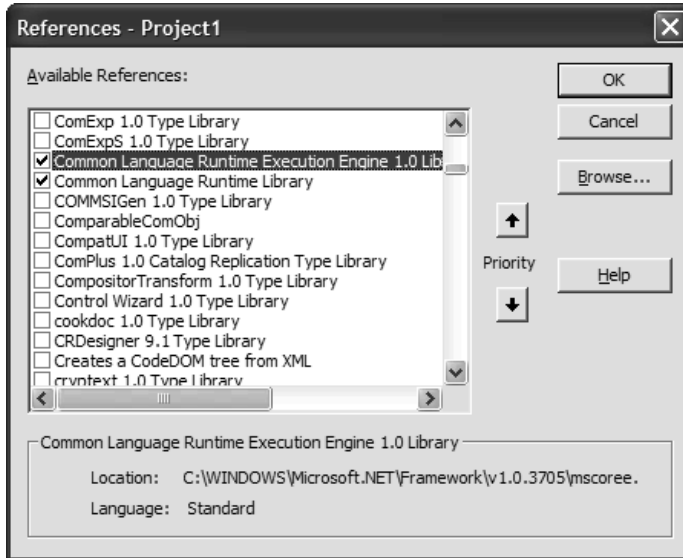


Figure 12-10. Referencing `mscorlib.tlb/mscorlib.tlb`

Various chapters of this text have already examined some types contained within `mscorlib.tlb`, but what of `mscorlib.tlb`? Like any loaded type library, the VB 6.0 Object Browser allows you to view the contained types. As you can see from Figure 12-11, despite the exotic nature of this exported assembly, `mscorlib.tlb` defines a surprisingly small number of items.

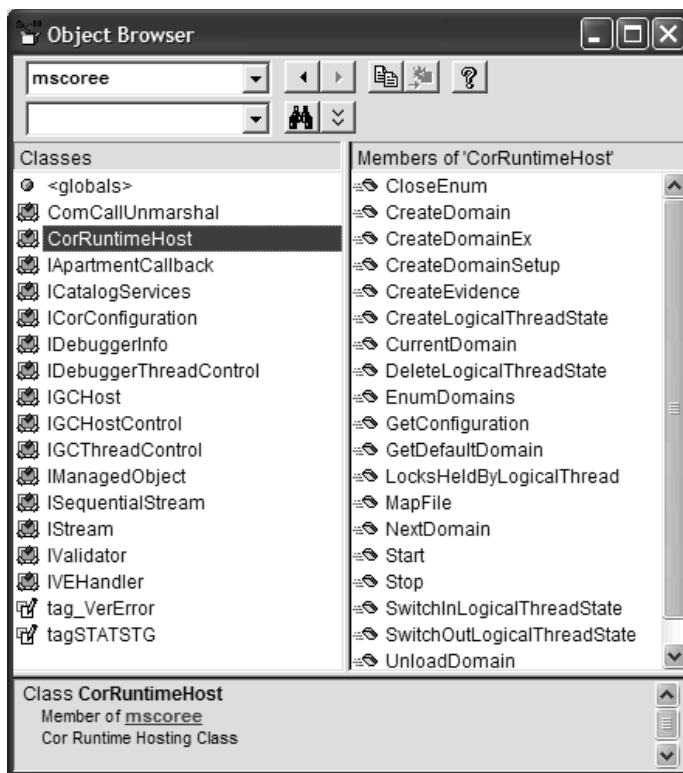


Figure 12-11. The crux of the CLR in terms of COM type information

A full treatment of each and every type defined in `mscorlib.dll` is beyond the scope of this text. Luckily, you are able to build a custom CLR host using a single type: `CorRuntimeHost`. This single .NET class type implements a set of interfaces (also defined within `mscorlib.tlb`) that provide the following functionality:

- The ability to load and unload .NET application domains
- The ability to manipulate the .NET garbage collector
- The ability to validate code within a given .NET assembly
- The ability to interact with a given debugger attached to the current process

So, given that `mscorlib.tlb` defines the types you need to build a custom CLR host, the next logical question is when you might want to do this. Besides the fact that building a custom host is extremely interesting in its own right, there is a practical reason to do so. When you build a custom host from unmanaged code,

you are able to dynamically load .NET assemblies for use by COM, without having to register the assembly using regasm.exe.

Building a Custom Host

The first detail of your VB 6.0 host is to establish a valid application domain to host the loaded assemblies. As you may know, under the .NET platform an application domain is a unit of isolated execution within a Win32 process (similar in function to the apartment architecture of classic COM). Just as a process may contain numerous application domains, a given application domain may contain numerous .NET assemblies. You are able to represent a given application domain using the `System.AppDomain` type.

Given this, the `Form_Load()` event handler creates an instance of `CorRuntimeHost`. Once the host has started, obtain a valid `AppDomain` via `CorRuntimeHost.GetDefaultDomain()`. The `Form_Unload()` event handler shuts down the CLR via the aptly named `CorRuntimeHost.Stop()`. Here is the story thus far:

```
' The types we need to host the CLR.
Private myAppDomain As AppDomain
Private myCLRHost As CorRuntimeHost

' Load the CLR and set app domain.
Private Sub Form_Load()
    Set myCLRHost = New CorRuntimeHost
    myCLRHost.Start
    myCLRHost.GetDefaultDomain myAppDomain
End Sub

' Unload the CLR.
Private Sub Form_Unload(Cancel As Integer)
    myCLRHost.Stop
End Sub
```

Now assume that the main Form has three VB 6.0 Button types. The Click event handler of the first button (`btnListLoadedAsms_Click()`) obtains and displays the list of each assembly currently hosted by the default application domain. To do this, you are able to obtain an array of Assembly types from the `GetAssemblies()` method of the `AppDomain` type. To display the name of each assembly, you are able to simply make use of the `Assembly.FullName` property:

```
' List all the loaded assemblies.
Private Sub btnListLoadedAsms_Click()
    Dim loadedAsms() As Assembly
    loadedAsms = myAppDomain.GetAssemblies()
```

```

Dim theAsms As String
Dim i As Integer
For i = 0 To UBound(loadedAsms)
    theAsms = theAsms + loadedAsms(i).FullName + vbCrLf
Next
MsgBox theAsms
End Sub

```

The next Button type is responsible for loading the System.Collections.dll assembly from the GAC to exercise the ArrayList type. Note how the CreateInstance() method requires you to send in (a) the friendly name of the assembly containing the type and (b) the fully qualified name of the type itself. What is returned from AppDomain.CreateInstance() is an ObjectHandle type, which provides the ability to obtain the underlying type using the Unwrap() method:

```

' Load a type from the GAC.
Private Sub btnLoadFromGAC_Click()
    Dim arLst As ArrayList
    Dim obj As ObjectHandle
    Set obj = myAppDomain.CreateInstance("mscorlib",
        "System.Collections.ArrayList")
    Set arLst = obj.Unwrap

    arLst.Add "Hello there!"
    arLst.Add 12
    arLst.Add True

    Dim items As String
    items = items + arLst(0) + vbCrLf
    items = items + CStr(arLst(1)) + vbCrLf
    items = items + CStr(arLst(2)) + vbCrLf

    MsgBox items
End Sub

```

If you run the application at this point, once you load `System.Collection.dll`, you find the message displayed in Figure 12-12.



Figure 12-12. Interacting with System.Collections.dll

The final button of your VB 6.0 Form type is responsible for loading a private, and unregistered, .NET assembly. To ensure that this example illustrates the point of loading unregistered .NET binaries, assume you have the following trivial C# class definition, defined in an assembly named (of course) `UnregisteredAssembly`:

```
using System;
using System.Runtime.InteropServices;

namespace UnregisteredAssembly
{
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class AnotherAdder
    {
        public AnotherAdder(){}
        public int Add(int x, int y)
        { return x + y;}
    }
}
```

Now, although you do not need to register this assembly, you still need to generate type information for your VB 6.0 client. Thus, run `tlbexp.exe` against this binary, and place the *.tlb and `UnregisteredAssembly.dll` files in the same directory as the current VB 6.0 project (Figure 12-13).

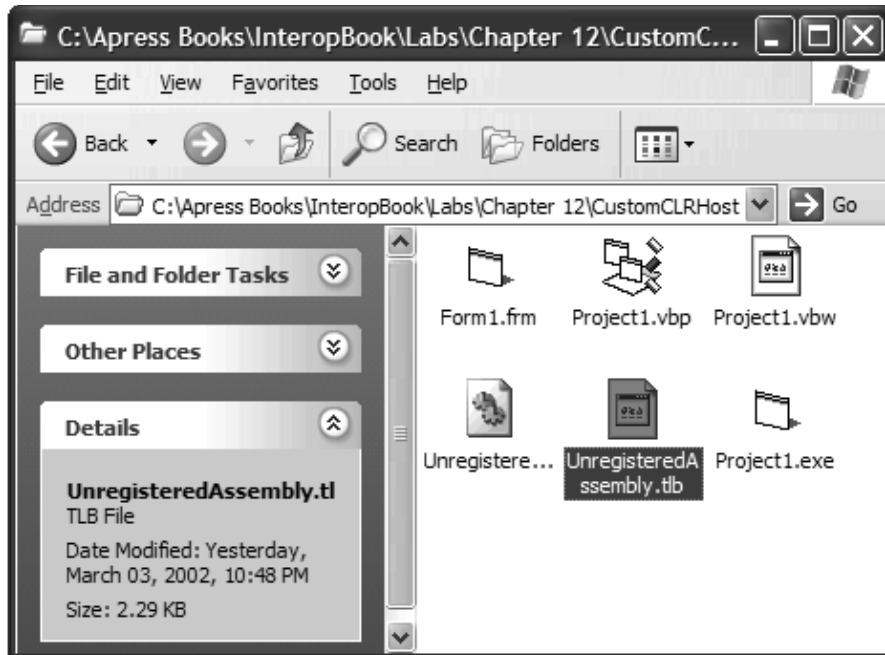


Figure 12-13. Configuring the unregistered assembly and related *.tlb file

Now that you have a private assembly, you are able to write the following event handler for the Form's final Button type:

```
' NOTE!!! Because VB projects do not directly
' run from the application directory within
' the IDE, you will
' need to run the EXE to use this function.
Private Sub btnLoadFromPrivateAsm_Click()
    Dim adder As AnotherAdder
    Dim obj As ObjectHandle
    Set obj = myAppDomain.CreateInstance("UnregisteredAssembly", _
        "UnregisteredAssembly.AnotherAdder")
    Set adder = obj.Unwrap
    MsgBox adder.Add(99, 3)
End Sub
```

As you can gather from the lengthy code comment, before you can test this final bit of functionality, you need to build the VB 6.0 application (File | Make) and run the application outside the VB IDE. Once you have built the EXE, simply double-click the executable file. If you loaded UnmanagedAssembly.dll and System.Collections.dll via the correct Button types, you would now find the results shown in Figure 12-14 when you click on the “list all loaded assemblies” Button type.

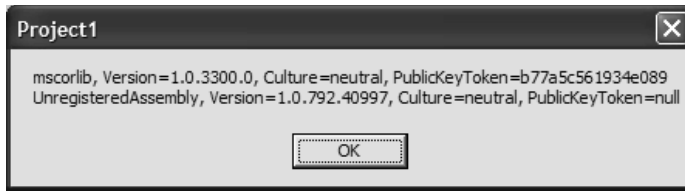


Figure 12-14. Documenting loaded assemblies

With your custom host complete, you come to the end of Chapter 12. As illustrated by this example, when you build a custom host for the CLR, you are able to avoid the process of registering .NET assemblies prior to building COM clients that consume them. If you want to dive into further details of the functionality of `mscorlib.dll`, be sure to check out the tool-builders documents included with the .NET SDK (installed by default under `C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Tool Developers Guide\docs`).



CODE *The `UnmanagedAssembly` and `CustomCLRHost` projects are located in the Chapter 12 subdirectory.*

Summary

The chapter wraps up your investigation of COM-to-.NET interoperability issues. As you have seen, just as a COM type can implement .NET interfaces to achieve type compatibility, a .NET type can implement COM interfaces to achieve binary compatibility with related coclasses. Using managed code, you are able to build managed representations of COM types to avoid creating a dependency with a related interop assembly.

Another key aspect of this chapter illustrated how you are able to build a customized version of `tlbexp.exe`. While you may never be in the position of needing to do so, this should solidify your understanding of what this tool does on your behalf. The final major topic presented here illustrated how you can interact with the CLR via `mscorlib.tlb` to build a custom host from unmanaged code.

At this point in the text, you have drilled quite deeply into the COM and .NET type systems, and you have seen numerous aspects of the interoperability layer. Before I wrap things up, the next (and final) chapter addresses the topic of building COM+ types (i.e., configured components) using managed code.

Index

*

- *.cab file, 686
- *.cls file, 147–148
- *.cpp file, 132–133
- *.def files, 90, 133
- *.dll files, 672
- *.idl files, 65
 - compiling with MIDL compiler, 77
 - format of, 162–163
 - manually editing, 135
 - regions of, 162
- *.il files, 510–512, 525
- *.msi file, 686
- *.pdb file, 468–469
- *.reg files, 137, 574
- *.res files, 513, 524
- *.rgs file, 139
- *.snk files, 261, 392, 640
- *.tlb files, 202, 535, 549, 586, 649, 655,
659–661, 666, 695

A

- Abstract base class
 - building in C#, 238–239
 - translating, 551
- Abstract method and property, defining, 258
- ACID properties of a transaction, 717
- Activator class, 331–332
- Activator.CreateInstance() method, 332
- ActiveX controls
 - building, 490–493

- consuming from managed code,
490–504
- consuming using VS .NET, 495–501
- generated assemblies, 498–501
- generated IDL, 493–494
- importing using AxImp.exe, 501–504
- ActiveX interop assemblies, 497–500
- Add Method tool (ATL), 145
- Add Reference dialog box, COM tab, 343
- Add() method, overloaded, 586
- AddArray() function, 6
- AddNumbers() function, 6, 34, 41
- [.addon] directive, 448, 454
- AddRef() method, 81, 82, 85
- ADO (active data objects), 399
 - accessing from a managed
application, 400
 - Connection type, 400
 - Recordset, 400–401
- ADO type library, reading, 211
- ADO.NET types, in custom type viewer,
322
- Advise() method, 438–439, 614
- Alias, mapping a function to, 31
- Allocated structures, receiving, 37–39
- AppDomain type, 663
- AppDomain.CreateInstance() method,
664
- AppID (COM server application ID), 70
- Application configuration file, 252–253
- Application directory
 - defined, 12
 - viewing, 251
- Application domain, 663

- Application object (VB 6.0), 176
- Application proxy (COM+), 686
- [Appobject] coclass, mapping, 387–388
- AppWizard utility (ATL COM), 128–129
- Array of pointers, in C++, 57
- Array type (System.Array), 348, 414, 416–417
- Array-centric value of UnmanagedType, 636
- ArrayList type, 478–479, 616, 619–620
- ArrayList.RemoveAt() method, 619–620
- Arrays
 - of blittable items, 350
 - in COM IDL, 179–184
 - C-style, 419–420
 - functions using, 6–7
 - marshaling, 34–35
 - of non-blittable items, 351
- ASAP deactivation, 709, 710–711
- ASP.NET Web Service client, 734–736
- Assemblies (.NET binaries), 233. *See also*
 - Interop assembly; Shared assemblies
 - accessing from C++ COM client, 590
 - of ActiveX controls, 498–501
 - vs. COM binaries, 234
 - compiling, 284
 - for complex C# code library, 302–304
 - composition of, 233–234
 - configured as COM invisible, 572
 - configuring private, 251–253
 - deploying, 582–583
 - displaying information about, 316–322
 - documenting loaded, 667
 - dumping to a file, 512–513
 - dynamically loading, 310–312
 - enumerating types in referenced, 311
 - installed in the GAC, 262
 - late binding, 285, 332–338
 - late binding to private, 332–334
 - late binding to shared, 335–338
 - logical view of, 237
 - referencing, 236, 584–586
 - registering with COM, 545–546, 644
 - referencing via VB 6.0, 584–586
 - specifying locations for, 253–254
 - strongly named, 255–256, 335, 692
 - using friendly names, 368
 - using the /primary flag, 393
 - viewing type names in, 313
- Assembly class, 310
- Assembly details, displaying, 317
- Assembly (interop). *See* Interop assembly
- Assembly manifest, 233, 242–245, 514
- Assembly metadata, viewing, 245–246
- Assembly statement, 367–371
- Assembly types, viewing, 243
- [Assembly:] prefix, 328
- Assembly.FullName property, 663
- Assembly.GetCustomAttribute()
 - method, 330
- Assembly.GetCustomAttributes()
 - method, 330
- Assembly.GetType() method, 311
- Assembly.Info.* file, 328
- AssemblyKeyFile attribute, 692
- Assembly-level attributes, 328–329
- Assembly.Load() method, 310
- Assembly.LoadFrom() method, 310
- ATL (active template library), 127–146
 - Add Method tool, 145
 - autogeneration of DLL exports, 130
 - implementation of coclass, 136–137
- ATL COM AppWizard utility, 128–129
- ATL COM map, 136
- ATL COM server
 - reading, 227
 - testing, 156–159

- ATL COM server project, 128
 - *.cpp file, 132–133
 - *.def file, 133
 - adding interface methods, 141
 - adding methods, 140–143
 - code updates, 135–136
 - initial IDL file, 131–132
 - inserting COM objects, 133–135
- ATL error server, building, 462–464
- ATL 4.0, SAFEARRAY helper templates, 184
- ATL Object Wizard, 133–135
- ATL Object Wizard Properties
 - Attributes tab, 134–135
 - Names tab, 134
- ATL project files, 129–133
- ATL project workspace, 129–131
 - files generated, 130
 - FileView, 131
 - with initial files, 131
- ATL registration support, 137–140
- ATL Simple Object methods, 169
- ATL string conversion macros, 179
- ATL 3.0, .NET type compatible coclass, 481–484
- AtIAdderClass type, 515, 522
- ATL-based coclass, COM interfaces for, 143–146
- AttachInterfaces() function, 501
- Attribute class, 323–324, 517
- Attribute class core members, 324
- Attribute metadata (.NET), 517–519
- Attribute-derived type
 - (System.Attribute), 525–526
- Attributes
 - assembly (and module) level, 328–329
 - defined, 72
 - IDL vs. .NET, 323, 326
 - .NET, 323–325

- reading at runtime, 330–335
 - restricting use of, 327–328
 - that take attributes, 327
- AttributeTarget enumeration, 327
- AttributeUsage type, 327
- A2W (ANSI to Unicode) macro, 179
- AutoComplete attribute, 712–713, 723
- AutoDual class interface, 643
- Auxiliary interfaces, defining in VB 6.0, 148–149
- Ax- prefixed assemblies, 497, 499–500
- AxHost base class, 499–500
- AxHost-derived type code, modifying, 504–508
- AxImp.exe utility, 501–504, 508–509

B

- Base class, specifying for a new type, 80
- Base client component design, 737
- Basic data types, functions using, 6–7
- Behavior of a class, explained, 52
- Binaries (.NET). *See* Assemblies
- Binary compatibility, VB 6.0 COM, 151–152
- Binary-compatible C# type, building, 641–642
- Binary-compatible VB .NET type, building, 642–643
- Binding (late), 331–338
- Binding process (.NET), 269–270
- BindingFlags enumeration, 333–334
- Bit reading/writing-centric members of Marshal type, 24
- Blittable data types, 349–350
- BSTR (BASIC String) data type, 178, 181, 183, 214
- COM strings as, 177–179
- translating to System.String type, 348, 351

BSTR COM library functions, 178
 ByRef keyword (VB .NET), 377–378,
 380–381
 Byte length prefixed Unicode characters,
 178
 Byte type (System.Byte), 595
 ByVal keyword (VB .NET), 377–381

C

C#

- accessing a configured .NET
 component, 699
- applying .NET attributes, 326
- building a binary-compatible .NET
 type, 641–642
- custom attribute, 517
- defining a dual interface, 651–653
- defining an event, 445
- foreach keyword, 434
- intercepting incoming COM events,
 455
- late binding to shared assemblies,
 337
- out keyword, 378
- params keyword, 421
- ref keyword, 377–378, 422
- serviced component example,
 724–736
- struct keyword, 598
- Windows Forms client application,
 732–733
- C# abstract type/base class, building,
 238–239
- C# callback client, building, 46–49
- C# class library, 235
- C# client application underlying IL,
 41–42
- C# client interacting with Custom DLL,
 40

- C# code library
 - building, 235–242
 - building complex, 302–304
- C# COM server client interop assembly,
 342–345
- C# COM type information viewer
 - building, 220–227
 - displaying COM types, 224–227
 - loading the type library, 221–224
- C# derived types/classes, building,
 240–242
- C# IDE, Implement Interface Wizard,
 642
- C# typeof operator, type reference from,
 306–307
- C#-style destructors, translating,
 549–551
- C++
 - #import directive, 102–103
 - defining an interface in, 54
 - dynamically writing and reading
 COM IDL, 161
 - implementing an interface in, 55–57
 - private class members, 54
 - public structure members, 54
 - VARIANTS in, 114–115
- C++ class, implementing, 119–122
- C++ class header file, 119
- C++ COM client, 101
 - accessing an assembly from, 590
 - building, 589–590
 - developing, 97–105
- C++ COM to .NET main() function, 613
- C++ COM-centric macros, 80
- C++ equality operator (==), overloading,
 71
- C++ event client
 - building, 610–614
 - client-side sink, 611–614
- C++ IDispatch example, 116–117

- C++ interface-based programming,
 - 52–62
- C++ with managed extensions, defined,
 - 232
- C++ not equal operator (!=), overloading,
 - 71
- Call objects (COM+), 674
- Callback
 - triggering using a function pointer,
 - 43–44
 - unmanaged, 42–43
- Callback client, building, 46–49
- Callback example, 43–44
- Callback function, 42, 44–45
- Callback pattern, 42
- CALLBACK tag, 43
- Calling conventions, specifying, 30
- CallingConvention field, 30
- Call-level context, explained, 671
- Categories, grouping COM objects into,
 - 577–578
- Categories (.NET), 578
- CategoryAttribute type, 506
- CATID (COM category ID), 70, 95,
 - 577–578
- C-based DLL, building custom, 5–9
- CComBSTR class, 178–179
- CComModule helper class, 137–138
- CComSafeArray helper template, 184
- CComSafeArrayBounds helper template,
 - 184
- CCW (COM Callable Wrapper), 539–544
 - simulation of COM identity, 543–544
 - simulation of implemented COM
 - interfaces, 542–543
- Character set, specifying, 29–30
- CharSet values, 29
- CheckThisVariant() method (VB 6.0),
 - 469–470
- Class, 51–62. *See also* Coclass
 - building, 281–284
 - building .NET type compatible,
 - 476–479
 - defined, 299
 - defining in IDL, 174–176
 - implementing in C++, 119–122
 - nested, 300
 - support for IUnknown, 72
 - supporting multiple behaviors, 60
 - supporting multiple interfaces, 59–60
- Class behavior, 52, 60
- Class characteristics (.NET), 299
- Class definition, 81
- Class details, displaying, 317
- Class factory
 - building, 85–87, 123
 - explained, 63
- Class interface (.NET), 557–562
 - autogenerated, 485–487
 - the case against, 563–564, 593
 - defined, 485
 - establishing, 559–561
 - registering, 580
- Class keyword (C#), 36–37
- Class library (C#), 235
- Class member information, displaying,
 - 320–322
- Class member parameters, displaying,
 - 321–322
- Class members, enumerating, 311
- Class members (C++), 54
- Class object, 63, 84
- Class structure declaration, 514
- Class types
 - building in C++, 55–58
 - defined, 299
 - functions using, 8–9
 - managed representation of, 36–37
 - .NET, 299–300, 357–358

- Class-centric members of
 - InteropServices, 364
- ClassInterface attribute, 690
- ClassInterfaceAttribute type, 486
- ClassInterfaceType enumeration, 558
- Client-side sink (C++ event client), 611–614
- Clone method(), 480–481
- Cloneable COM object, using, 481
- Cloneable COM type, building, 480–481
- CLR (Common Language Runtime), 231, 662
- CLR host, building, 662, 663–667
- CLS (Common Language Specification), 231
- CLSCTX, core values of, 98
- CLSID (class ID), 70, 75, 96, 140, 397
- CLSID key, 92–93, 575–577
- Coclass, 63. *See also* Class
 - accessing with CoCreateInstance(), 100
 - ATL implementation of, 136–137
 - building, 200–201
 - configuring attributes of, 135
 - default constructor for, 389
 - defined, 51
 - defining in IDL, 75–76, 155–156, 175
 - implementing .NET interfaces, 475–484
 - managed, 453–454
 - naming, 134
 - naming in VB 6.0, 147
 - RCW for, 341
 - support for IUnknown, 72
- Coclass (ATL), COM interfaces from, 143–146
- Coclass conversion, 385
- Coclass IDL attributes, 175
- Coclass keyword, 75
- Coclass statistics, listing, 208
- CoCreateGuid() function, 69, 191
- CoCreateInstance() method, 97, 100, 105
- Code provider, building, 279
- CodeDOM (Code Document Object Model), 270
 - languages supported by, 272
 - member-building types of, 275
 - namespace-building types of, 274
 - type-building types of, 274
 - types of, 274–275
- CodeDOM example, 276–284
- CodeDOM namespace, 270–284
- CoGetCallContext() API function, 674
- CoGetClassObject() API function, 97–98, 100
- CoGetObjectContext() API function, 674
- Collection member variable (private), 429
- Collection type (VB 6.0), 429, 645
- Collections (COM), 426–436
 - from managed code, 433–436
 - typical members of, 429
- Collections (custom .NET), 614–619
- Collections namespace
 - (System.Collections), 615
- Collections.dll (System), 665–666
- COM array representation, 179–184
- COM atoms, manually defining, 650–653
- COM binaries, vs. .NET binaries
 - (assemblies), 234
- COM classes. *See* Class; Coclass
- COM client
 - accessing System.Type from, 588
 - C++, 97–105, 589–590
 - obtaining .NET type's enumerator, 622
 - VB 6.0, 103–105, 644–646
 - VBScript, 590–591
- COM coclass. *See* Coclass

- COM collections, 426–436
 - from managed code, 433–436
 - typical members of, 429
- COM (Component Object Model), 51
 - exposing custom .NET interfaces to, 564–566
 - language-independence of, 51, 84
 - path of, 230
 - registering .NET assemblies with, 644
- COM connection points, 437–440
- COM data types
 - vs. COM types, 163–164
 - defined, 163
 - primitive, 164–167
- COM DLL, composition of, 63–64
- COM DLL function exports, 64
- COM DLL project workspace, creating, 67–68
- COM enum statistics, listing, 209–210
- COM enums. *See also* Enums
 - as name/value pairs, 597
 - converting to .NET, 391–392
 - converting .NET enums to, 593–598
 - 32-bit storage, 596
- COM error information, handling from managed code, 466–468
- COM error objects, 459–464
- COM event interface, creating, 605–606
- COM events
 - intercepting incoming in C#, 455, 456
 - intercepting incoming in VB .NET, 456–457
 - from managed environment, 437–440
- COM IDL, dynamically writing and reading (C++), 161
- COM IDL data types, conversion to managed data types, 346–351
- COM interface types, IDL definitions of, 171–185
- COM interfaces, 51, 68–79, 136
 - as strongly typed variables, 58–59
 - for an ATL-based coclass, 143–146
 - CCW simulation of, 542–543
 - class support for multiple, 59–60
 - consumed by RCW, 351–353
 - containing methods, 106
 - converting to managed equivalent, 371–374
 - defining auxiliary in VB 6.0, 148–149
 - defining in C++, 54
 - defining in IDL, 73
 - defining and supporting multiple, 76
 - derived, 487–488
 - hidden from managed client, 351–352
 - implementing in C++, 55–57
 - implementing explicit, 567
 - implementing in VB 6.0, 149–151
 - from multiple base interfaces, 301
 - [oleautomation]-compatible, 633
 - parent interface of, 624
 - registered for universal marshaling, 580–582
 - supported by VB 6.0, 154
 - versioned/versioning, 61–62, 373–374
 - versioning existing, 62
 - viewing metadata for, 247
- COM invisible, .NET assemblies
 - configured as, 572
- COM late binding syntax, vs. .NET, 336–338
- COM library definition, controlling, 568
- COM library information, displaying, 205–207
- COM map, defined, 136
- COM memory management, 72
- COM metadata, translating into .NET metadata, 249–250
- COM method parameter, conversion to .NET method parameter, 377–381

- COM objects, 72
 - activating, 97–100
 - defined, 51
 - destroying using Marshal class, 474–475
 - grouping into well-known categories, 577–578
 - language- and location-neutral, 84
- COM programming frameworks, 127–159
- COM properties, 105–107
 - defining in VB 6.0, 155
 - represented internally as methods, 106
- COM registration, 94–95
- COM SAFEARRAYs. *See* SAFEARRAYs
- COM server, anatomy of, 51–126
- COM server registration, 91–97
- COM server registration file, updating, 124
- COM strings, as BSTR data types, 177–179
- COM structures, 421–425
 - converting .NET structures to, 598–603
 - from managed code, 424–425
- COM subsystem, initializing, 97
- COM type
 - building cloneable, 480–481
 - building connectable, 441–442
- COM type definitions, generating, 545
- COM type information, 161–228
 - dumping, 207–208
 - generating at runtime, 161
 - generating programmatically, 189–191
 - loading programmatically, 531–533
 - reading programmatically, 203–212
- COM type information generation, testing, 201–203
- COM type information viewer, in C#, 220–227
- COM type libraries
 - library statement section, 368–371
 - loading, 221–224
 - registering, 582
 - setting references to, 156
 - [version] identifier, 368
- COM type and managed code, marshaling calls between, 340
- COM type to .NET type conversion rules, 371–392
- COM types
 - vs. COM data types, 163–164
 - defined, 163
 - displaying, 224–227
 - manipulating using `_Class` types, 357–358
 - manipulating using discrete interfaces, 358–359
 - with multiple [source] interfaces, 457–459
 - table of, 164
 - using [default] interface to interact with, 361–362
 - using managed interfaces to interact with, 359–361
- COM VARIANT. *See* VARIANT data type
- COM wrapper types, 355, 398
- COM+ application
 - activation level, 680
 - configuring using .NET attributes, 703–704
 - creating, 679–681
 - deploying, 685–687
 - defined, 670
 - installing a .NET type, 693–694
 - library or server, 680
 - loading within `dllhost.exe`, 698
 - registering, 696–697

- COM+ Application Export Wizard, 686
- COM+ Application Install Wizard, 679, 681
- COM+ Catalog, 671
 - new COM+ application in, 697
 - role of, 675–677
- COM+ client, VB 6.0, 683–685
- COM+ (Component Services), 669–738
 - application-level attributes, 703
 - component statistics, 699–700
 - instance management, 708–709
 - location transparency, 683
 - object construction strings, 704–706
 - poolable objects, 715–717
 - transactional programming, 720–724
- COM+ example, 682–683
- COM+ Explorer. *See* Component Services Explorer
- COM+ interop, explained, 738
- COM+ 1.0, 670
- COM+ 1.5, 670
- COM+ 1.5 private components, 733–734
- COM+ runtime environment, 672–675
- COM+ shared property manager, 688
- COM+ type
 - activated, 684
 - installed, 683
 - stateless, 709
- COM+-specific behaviors, 671–672
- COMAdminCatalogClass type, 676
- ComAdmin.dll, 675–676
- COM-aware .NET types
 - guidelines for building, 547–554
 - type member visibility, 548
 - type visibility, 547
- COM-centric macros, 79–80
- COM-centric members of Marshal type, 20–21, 472
- _com_error type, 623
- ComEventInterfaces attribute, 607
- COMException type, 466
- ComImportAttribute type, 647, 652
- ComInterfaceAttribute type, 605
- ComInterfaceType values, 372, 564, 647
- ComInterfaceType.InterfaceIsDispatch, 566
- ComInterfaceType.InterfaceIsDual, 565
- ComInterfaceType.InterfaceIsUnknown, 565, 629
- Communications proxy (.NET-to-COM). *See* RCW
- CompileCode() method, 284
- Compiling an assembly, 284
- Component design, base clients and, 737
- Component housing, generating, 128–129
- Component Services. *See* COM+
- Component Services Explorer, 671, 675, 678–681, 685
- Component statistics (COM+), enabling, 699–700
- ComRegisterFunctionAttribute type, 653
- ComSourceInterfaces attribute, 606, 609
- COM-to-.NET communications, core requirements, 544–546
- COM-to-.NET data type mappings, 347
- COM-to-.NET interoperability
 - advanced, 633–667
 - basic, 539–592
 - intermediate, 593–631
- ComUnRegisterFunctionAttribute type, 653
- ComVisible attribute, 475
- ComVisibleAttribute type, 548–549
- Configured component, defined, 671
- Conformant C-style arrays, 419–420
- Connectable COM type, building, 441–442
- Connectable object, 437–439

- Connection point architecture (classic COM), 437
- Connection point container, 437
- Connection points (COM), 437–440, 604–609
- Consistency bit (JITA), 710
- Const keyword, 165
- Constant members, translating, 554
- ConstructionEnabled attribute, 705
- Constructors
 - parameterized, 300
 - translating, 549–551
- Context object (COM+), 671–673, 675
- ContextUtil type, 706–708
- ContextUtil.DeactivateOnReturn attribute, 723
- ContextUtil.MyTransactionVote property, 723
- ContextUtil.SetAbort() method, 722
- ContextUtil.SetComplete() method, 722
- Conversion rules, COM type to .NET type, 371–392
- ConvertAssemblyToTypeLib() method, 659
- ConvertTypeLibToAssembly() method, 533–536
- CorRuntimeHost type, 662
- CorRuntimeHost.GetDefaultDomain() method, 663
- CorRuntimeHost.Stop() method, 663
- COSERVERINFO structure, 98
- CoUninitialize() method, 97
- CreateFile() helper function, 280
- CreateInstance() method, 84, 86–87, 332, 664
- CreateInterface() helper method, 195–200
- CreateTypeInfo() method, 196, 201
- CRMs (compensating resource managers), 737
- C-style arrays, 419–420
- C-style DLLs, 2–5, 27
- CTS (Common Type System), 231, 294
- Culture information (strong name), 260
- Currency value (UnmanagedType), 637
- CurrencyWrapper, 409
- CUSTDATA structure, 216
- CUSTDATAITEM array, 216–217
- Custom C-based DLL, building, 5–9
- Custom CLR host, building, 662, 663–667
- Custom COM interfaces, managed versions of, 638–644
- Custom database, building, 725–726
- Custom dialog GUI, 318
- Custom DLL
 - C# client interacting with, 40
 - exports of, 10–11
 - imported modules used by, 10
 - interacting with, 33–42
- Custom DLL path, 14
- Custom IDL attributes
 - AxImp.exe and, 503–504, 508–509
 - defining, 212–218
 - for namespace naming, 370
 - for ProgID of COM type, 561
 - reading, 214–218
 - tlbimp.exe and, 508–509
- Custom members, exporting, 3–4
- Custom metadata, viewing, 326
- Custom .NET attributes, building, 325–329
- Custom .NET collections, 614–619
- Custom .NET data types, building, 297–301
- Custom .NET exceptions, 620–621
- Custom .NET interfaces
 - exposing to COM, 564–566
 - registering, 580
- Custom .NET type viewer, building, 312–322

- Custom stub and proxy DLL, 581–582
- Custom type library importer, 528–538
- Custom wrapper (.NET), creating, 519–521
- Customize Toolbox, COM Components tab, 496
- CustomReg() method, 654
- CustomUnreg() method, 654

D

- Data type conversions, 18–19, 540–542
- Data type language mappings, 296–297
- Data type mappings, COM-to-.NET, 347
- Data type representation, Win32 and .NET, 19
- Data type system (.NET), 294–297
- Data types (*see also* COM type information; .NET types)
 - blittable, 349–350
 - building custom, 297–301
 - COM, 163
 - COM primitive, 164–167
 - converting between managed and COM IDL, 346–351
 - .NET, 298
 - non-blittable, 349–351
- Data types (.NET), categories of, 298
- Data-centric values of UnmanagedType, 636–637
- DataGrid type (.NET), 401–402
- DataTable type, building with ADO recordset, 401
- Debugging COM servers using VS .NET, 468–470
- Declarative programming, COM+, 671
- Declarative transactional settings, 720
- DECLARE_REGISTRY_RESOURCE macro, 138
- Declspec (declaration specification), 4

- Default constructor (private), 387
- Default context, defined, 675
- [Default] IDL attribute, 76
- [Default] interface, 361–362
 - adding members to, 140–141
 - defining, 76
- Default interop marshaler, changing, 637
- Default parameters (COM IDL), 381–383
- Default stub code, VB 6.0 IDE used to generate, 150
- [Defaultvalue] keyword (COM IDL), 381–383
- Delegate keyword, 444
- Delegates (.NET), 298, 443–445
 - converting to COM connection points, 604–609
 - defined, 443
 - generated by tlbimp.exe, 450–451
- Derived interfaces, 487–488
- Derived type, viewing metadata for, 247–248
- Derived types/classes, building, 240–242
- DescriptionAttribute type, 506, 518–519, 522
- DescriptionAttributes type, 504
- Destructors (C#-style), translating, 549–551
- Digital signature (strong name), 260
- Discrete interface references, 358–359
- Discrete interfaces, using to manipulate COM types, 358–359
- DispEventAdvise() method, 611, 614
- DispEventUnadvise() method, 611
- DispGetIDsOfNames() method, 122
- DispIdAttribute type, 502, 652
- DISPID_BACKCOLOR, 492–494, 502
- DISPID_NEWENUM, 435

- DISPIDs, 109–110, 173, 397, 652
 - assigning to event interface
 - members, 605
 - cataloging, 388–389
 - controlling generation of, 566
 - obtaining, 111–112
 - setting, 432
 - Dispinterface (oleautomation interface),
 - 109–110, 166
 - defined, 108–109, 171
 - defining, 171–172
 - raw, 566, 604, 611
 - DISPPARAMS structure, 112, 115–116
 - DLL client, dynamic C++, 15–17
 - DLL component housing,
 - implementing, 88–90
 - DLL exports, ATL autogeneration of, 130
 - DLL functions, exporting, 90–91
 - DLL project type, selecting, 67
 - DllCanUnloadNow() method, 88–89
 - Dllexport declaration specification, 4–5
 - DllGetClassObject() method, 88,
 - 123–124
 - Dllhost.exe, 672, 698
 - DllImport statement, 34–35, 37, 47
 - DllImportAttribute type, 25–26, 28, 41
 - DllImportAttribute type fields, 28
 - DllMain() method, 2–3, 68
 - DllRegisterServer() method, 90, 95
 - DLLs (dynamic link libraries), 129
 - core system-level, 12
 - C-style, 2–5, 27
 - custom, 10–11, 33–42
 - deploying traditional, 12–14
 - location of core Win32, 13
 - DllUnregisterServer() method, 90
 - Domain (application), 663
 - Done bit (JITA), 709–710, 712–713, 722
 - Doomed bit (in transaction processing),
 - 719
 - Dot notation, for nested namespace
 - definitions, 371
 - DotNetCalcWithInterface type, 589
 - Dual attribute, 172
 - Dual interfaces, 118–119
 - defining, 172–173, 696
 - defining using C#, 651–653
 - explained, 118
 - VB 6.0, 155
 - Dumpbin.exe
 - flags, 9
 - viewing imports and exports with,
 - 9–11
 - DumpComTypes() helper function, 207
 - DumpLibraryStats() method, 206
 - DWORD parameter, 2
 - Dynamic C++ DLL client, 15–17
 - Dynamically loading an assembly,
 - 310–312
 - Dynamically loading an external library,
 - 15–16
 - DynamicInvoke() method, 444
- ## E
- EnterpriseServices core types, 687–688
 - EnterpriseServices namespace, 687–690,
 - 701, 703, 736–737
 - EnterpriseServices.AutoComplete
 - attribute, 712–713
 - EnterpriseServices.ContextUtil type,
 - 706–708
 - EnterpriseServices.dll, 669, 691, 699
 - EnterpriseServices.ServicedComponent
 - type, 689–690
 - EntryPoint field, 31
 - EntryPointNotFoundException, 29
 - Enum base class type (System.Enum),
 - 597
 - Enum details, displaying, 317

- Enum keyword, 176
- Enum type, 301
 - extracting underlying name of, 597
 - underlying, 596
- Enumerating class members, 311
- Enumerating method parameters, 311–312
- Enumerating types in a referenced assembly, 311
- Enumeration, viewing metadata for, 246
- Enumeration fields, displaying, 320
- Enumeration types (.NET), 301
- Enums (*see also* COM enums; .NET enums)
 - as name/value pairs, 176, 213, 597
 - assigned alternative numeric values, 594
 - assigned a default numeric value, 594
 - naming convention, 594
- Environment variables, viewing, 14
- Equality tests, 290, 292–294
- Equals() method, 290, 293–295
- Error 1400, 33
- Error handling (.NET), 464–468
- Error information (COM), handling from managed code, 466–468
- Error Lookup utility, 32
- Error objects (COM), 459–464
- Error-centric members of Marshal type, 24
- Event client (C++), building, 610–614
- Event client (VB 6.0), building, 609–610
- Event interface
 - creating, 605–606
 - name string, 606
 - using ComSourceInterfaces attribute, 606
- Event keyword (VB .NET), 609
- Event keyword (VB 6.0), 442
- Event metadata (.NET), 447

- Event server (.NET), building using VB .NET, 608–609
- Event-centric generated types, 449–450
- Events (*see also* COM events)
 - defining in C#, 445
 - defining and sending in VB 6.0, 441
 - loosely coupled, 670–671
 - .NET events, 445–448, 604–606, 608–609
- EventTrackingEnabled attribute, 700
- ExactSpelling field, specifying, 29
- Exceptions (.NET), 619–621
- Explicit interfaces, implementing, 567
- Exported class types, interacting with, 39–40
- Exporting custom members, 3–4
- Exporting DLL functions, 90–91
- Exports, viewing using dumpbin.exe, 9–11
- EXPORTS tag, 4
- Extending COM types, 390–391
- External library, dynamically loading, 15–16

F

- Fields (database)
 - converting .NET, 557
 - reordering at runtime, 35
- FillListBoxes() helper function, 224–226
- Fixed-length C-style arrays, 419
- Foreach keyword (C#), 434
- FormatMessage() API function, 33
- Form_Load() event handler, 663
- Form_Unload() event handler, 663
- Friendly alias, mapping a function to, 31
- Friendly name, 260, 310, 368
- Friendly salutation, 258
- Friendly string name, 310
- FullName property (assembly), 663

- FUNCDesc structure, 198–199, 209
- Function entry points, specifying, 31
- Function pointers
 - array of, 57
 - smart, 102
 - using to trigger a callback, 43–44
- Functions
 - mapping to friendly aliases, 31
 - pure virtual, 54
 - receiving structures, 7–8
 - using basic data types and arrays, 6–7
 - using class types, 8–9

G

- GAC (Global Assembly Cache), 12
 - binding to an assembly in, 335
 - loading an interop assembly from, 396
 - machine-wide interop assembly in, 640
 - placing interop assemblies in, 392
 - served assembly in, 692
 - shared assemblies in, 255
 - VB .NET binary installed in, 262
- Garbage Collector (.NET), 714–715
- GenerateAssemblyFromTypeLib()
 - method, 536–538
- GenerateGuidForType() method, 474
- GenerateTLBFromAsm() helper
 - function, 655–656, 659
- GetCollection() method
 - (COMAdminCatalogClass), 676
- GetCustomAttributes() method, 330
- GetDefaultDomain() method, 663
- GetDescription() method, 620
- GetDocumentation(), 532–533
- GetEnumerator() method, 615, 622–623
- GetHashCode() method, 290, 293
- GetIDsOfNames() method, 111–122
- GetUnknownForObject() method, 474
- GetLastWin32Error() method, 32–33
- GetMembers() method, 311
- GetMethod(), overloading, 336
- GetObjectForIUnknown() method, 474
- GetOcx() method, 501
- GetParameters() method, 311–312
- GetProcAddress() method, 16–17
- GetType() method, 290, 306–308, 398
- GetTypeInfo() method, 122
- GetTypeInfoCount() method, 122
- GetTypeLibName() method, 536
- GetTypes() method, 311
- Global counters, 64
- Global ULONG, 87
- Global variables, 64
- Global-level attributes, 328
- GUID helpers, 70–71
- GUID structure, 69
- GuidAttribute type, 372
- GUID-centric members of
 - InteropServices, 366
- Guidgen.exe utility, 69
- GUIDs, 213–214, 216–217
 - comparison of, 70
 - defined, 69
 - freezing current values of, 152
 - mapping managed, 348–349
 - obtaining at design time, 69
 - preventing generation of, 151
 - role of in COM interfaces, 68–71
- Guid.ToString() method, 348

H

- Handles keyword (VB .NET), 456–457
- Happy bit (JITA), 709–710, 722–724
- Heap-based entities, 598
- Helper sink, building, 535–536
- [Helpstring], 157

- [Helpstring] attributes, 131–132, 492–494, 506
- Helpstrings (.NET), 509–510
 - AxImp.exe and, 503–504, 508
 - tlbimp.exe and, 508–509
- Hives (registry), 91
- HKCR (HKEY_CLASSES_ROOT) hive, 91
 - CLSID key, 92–93, 575
 - Component Categories key, 95, 579
 - ProgIDs key, 92
 - TypeLib key, 94
- HRESULTS (COM), 74, 82, 459–460
- Hungarian notation, defined, 19

I

- IAdd interface, 516
- IBaseInterface interface, 487, 489
- IBasicMath interface, 565, 567
- IClassFactory interface, 84–87, 136
 - CreateInstance() method, 86–87
 - LockServer() method, 87
- ICloneable interface, 480–481, 483
- ICloneable.Clone method(), 480–481
- ICodeCompiler interface, 280
- ICodeGenerator interface, 279
- IComparable interface, 476–477, 479
- IConnectionPoint interface, 438–439, 441, 610, 614
- IConnectionPointContainer interface, 437–438, 441, 609–610, 614
- IContextState interface, 673, 710
- ICreateErrorInfo interface, 460–461
- ICreateTypeInfo interface, 657
- ICreateTypeInfo2 interface, 190
- ICreateTypeLib interface, 191–193, 201
- ICreateTypeLib interface members, 192
- ICreateTypeLib.SaveAllChanges() method, 657, 659
- ICreateTypeLib2 interface, 190

- IDerivedInterface interface, 487–488
- IDispatch client (VB 6.0), 117
- IDispatch example (C++), 116–117
- IDispatch interface, 124
 - in action, 125
 - for building scriptable objects, 108–112
 - helper functions, 120
 - IDL definition of, 108–109
 - implementing, 121
 - methods, 109
- IDispatch-based interface statistics, listing, 209
- IDL
 - meta language used to describe COM items, 65
 - viewing with VB 6.0 Oleview.exe, 152–156
- IDL attributes, 72
 - defining custom, 212–218
 - vs. .NET attributes, 323, 326
 - reading custom, 214–218
- IDL COM types, viewing in Object Browser, 157
- IDL const keyword, 165
- IDL core data types, 165–166
- IDL data type conversion, 346–351
- IDL enumeration, defining, 176
- IDL interface attributes, 174
- IDL interface definition, 177
- IDL interface modifiers, 173–174
- IDL method parameter attributes, 167–171
- IDL parameter attributes, 74
- IDL structures, defining, 176–177
- IDL syntax to define COM interfaces, 171–185
- IDLCustomAttribute type, 521–524
- IDR_ custom resources, 138–139
- IE (Internet Explorer), 124

- IEnumConnectionPoints interface, 438
- IEnumerable interface, 434–435, 615, 617, 621–623
- IEnumerable.GetEnumerator() method, 622–623
- IEnumerator interface, 436
- IEnumVARIANT interface, 431–432, 436, 615, 617, 621
- IEnumXXXX interface, 431–432
- IErrorInfo interface, 460–461, 466, 620
- IErrorInfo.GetDescription() method, 620
- IHello interface, creating, 193–198
- IID (interface ID), 70, 566
- IL (Intermediate Language), 41
- ILDasm.exe tool, 243–250
 - building a version of, 312–322
 - underlying IL code, 249–250
- Implement Interface Wizard, 144, 481–483, 642, 643
- Implemented interface, 386
- Implements definitions, 149
- [Implements] directive, 385
- ImportedFromTypeLibAttribute type, 369
- ImporterEventKind enumeration, 535
- Importlib() statement, 586
- Imports, viewing using dumpbin.exe, 9–11
- InAttribute type, converting, 556
- IndexOutOfBounds exception, 620
- IndexOutOfRangeException exception, 619
- Inheritance, multiple, 60
- InstallAssembly() method, 701
- Instance of a class, in class definition, 36
- Instance management (COM+), 708–709
- Interface details, displaying, 317
- Interface hierarchies
 - implementing, 487–489
 - importing, 373–374
- Interface members to .NET method
 - conversion, 375–377
- Interface methods, 83, 141
- Interface properties, IDL syntax for, 106–107
- Interface references, 57, 184
- Interface types as method parameters, 184–185
- Interface types (.NET), 301
- InterfaceIsDispatch COM interface type, 566
- InterfaceIsDual COM interface type, 565
- InterfaceIsUnknown COM interface type, 565, 629
- Interfaces. *See* COM interfaces; Dual interfaces; .NET interfaces
- Interfaces from a non-COM perspective, 52–62
- Interface-based programming, 52–62
- Interface-centric members of
 - InteropServices, 364–365
- InterfaceTypeAttribute type, 372, 629, 647
- [Internalcall] directive, 377
- Interop assembly, 448–459
 - building, 342–346, 519–526
 - building and deploying, 640
 - building with tlbimp.exe, 354–355
 - defined, 342
 - deploying, 392–393
 - dumping to a file, 512–513
 - editing, 510–517
 - generating, 511
 - IL/metadata definitions, 514–517
 - loading from the GAC, 396
 - modifying manually, 508–510
 - namespace naming, 369–371
 - .NET types in, 356–362
 - obtaining, 353–355
 - placing in the GAC, 392
 - primary, 393–396
 - private, 343–344
 - recompiling the IL code, 524–526

- strongly named, 640
 - two ActiveX-generated, 497–500
 - updating, 522–524
- Interop assembly attributes, 396–399
- Interop assembly metadata, 398, 514–517
- Interop assembly registration,
 - interacting with, 653–655
- Interop assembly-centric members of
 - InteropService, 363
- Interop marshaler, changing the default, 637
- InteropServices namespace, 18, 218–220, 362–367
 - class-centric members, 364
 - GUID-centric members, 366
 - InAttribute and OutAttribute types, 556
 - interface-centric members, 364–365
 - interop assembly-centric members, 363
 - managed attributes, 378–379
 - method-centric members, 365
 - parameter-centric members, 365
 - registration-centric members, 363–364
 - runtime-centric members, 366
 - type library-centric members, 363
 - types to handle VARIANTS, 409
 - visibility-centric members, 363–364
- InteropServices.COMException type, 466
- InteropServices.RegistrationServices
 - type, 655
- InteropServices.RuntimeEnvironment
 - role, 366–367
- InteropServices.TypeLibConverter type, 528–530, 655–656
- Invoke() method, 111, 333–335
- Invoking members, 16–17
- IObjectConstruct interface, 672
- IObjectContext interface, 673
- IObjectContextActivity interface, 673
- IObjectContextInfo interface, 673
- IObjectControl interface, JITA and, 713–715
- IObjectControl interface methods, 710, 713
- ISecurityCallContext interface, 674
- IStartable and IStoppable, 647
- ISupportErrorInfo interface, 462–463
- ITypeInfo interface, 121, 185–189, 207
 - core members of, 187
 - data types, 188–189
 - related structures and enums, 188–189
- ITypeInfo2 interface, 188, 215, 217
- ITypeInfo2 interface core members, 215
- ITypeLib interface, 204–206, 532–533
- ITypeLib interface core members, 206
- ITypeLibConverter interface, 530
- ITypeLibExporterNotifySink interface, 656, 659
- ITypeLib.GetDocumentation(), 532–533
- ITypeLibImporterNotifySink interface, 533
- ITypeLib2 interface, 217
- ITypeLib2 interface core members, 215
- IUnknown interface, 63
 - AddRef() method of, 81–82
 - formal IDL definition of, 73
 - implementing, 81–83
 - interacting with using Marshal class, 473–474
 - methods, 72
 - Release() method of, 81–82
 - role of, 71–73
- IUnknown-based interface statistics,
 - listing, 209
- IUnknown-derived interfaces, building, 173

J

Java, path of, 230
 JITA (just-in-time activation), 671,
 708–715
 enabling, 710–711
 and IObjectControl, 713–715
 JITAwareObject class type, 714
 JustInTimeActivation attribute, 711

K

Keys (registry), 91
 Keywords, language-specific, 84,
 296–297

L

Language files (MIDL output), 66
 Language mappings of system data
 types, 296–297
 Language- and location-neutral COM
 object, 84
 Language-independence of binary IDL,
 65
 Language-independence of COM
 components, 51, 84
 Languages supported by CodeDOM, 272
 Language-specific keywords, 84,
 296–297
 Late binding, 331–338
 Activator class and, 331
 invoking a member using, 333
 .NET platform, 331
 to a private assembly, 332–334
 to shared assemblies, 335–338
 Late binding syntax, COM vs. NET,
 336–338
 Late-bound clients, 110–111, 117,
 124–126, 155

Late-bound VB 6.0 IDispatch client, 117
 Late-bound VBScript client, building,
 124–126
 LayoutKind enumeration, 35, 603
 LayoutKind.Auto, 603
 LayoutKind.Explicit, 603
 Lazy (automatic) registration, 693,
 700–701
 LCE (loosely coupled events), 670–671
 Legacy binary modules, accessing using
 PInvoke, 49
 LIBID (COM type library ID), 70
 Libraries (type). *See* Type libraries
 Library applications (COM+), 680
 Library of C# code, building complex,
 302–304
 Library statement, 163, 367–371
 LIBRARY tag, 4
 Library version attribute, 75
 Library-centric Win32 API functions, 15
 LoadAndRunAsm() helper function, 285
 LoadCOMTypeInfo() helper function,
 532
 Loading an assembly dynamically,
 310–312
 Loading an external library dynamically,
 15–16
 LoadLibrary() method, 15
 LoadLists() helper function, 313–314
 LoadTypeLib() COM library function,
 204
 LoadTypeLibEx() method, 217
 LoadTypeLibrary() helper function, 223
 Location transparency (COM+), 683
 Lock counter, 64
 LockServer() method of IClassFactory, 87
 LONG, 109

M

- Macros (COM-centric), 79–80
- Managed client, building, 250–253, 526–528
- Managed coclass, 453–454
- Managed code, defined, 1
- Managed COM wrapper types, 355
- Managed data types
 - COM IDL conversion to/from, 346–351
 - explained, 27
- Managed delegates, 450
- Managed GUID mappings, 348–349
- Managed interfaces
 - tlbimp.exe-generated, 451–453
 - using to interact with COM types, 359–361
- Managed languages, working with, 232–233
- Manifest (assembly), 233, 242–245, 514
- Marshal class, 20–25, 471–475
 - destroying COM objects, 474–475
 - interacting with IUnknown, 473–474
 - type library-centric members of, 21
- Marshal type
 - bit reading/writing-centric members of, 24
 - COM-centric members of, 20–21, 472
 - error-centric members of, 24
 - memory/structure-centric members of, 23
 - string conversion members of, 22
- Marshal.AddRef() method, 474
- Marshal.AsAttribute type, 420, 633–637
- Marshal.GenerateGuidForType() method, 474
- Marshal.GetIUnknownForObject() method, 474
- Marshal.GetLastWin32Error() method, 32–33
- Marshal.GetObjectForIUnknown() method, 474
- Marshal.GetTypeLibName() method, 536
- Marshaling arrays, 34–35
- Marshaling calls between managed code and COM type, 340
- Marshal.Release() method, 474
- Marshal.ReleaseComObject() method, 474
- MC++ (Managed C++), 232–233
- Member-building types of CodeDOM, 275
- Members, invoking, 16–17
- Memory/structure-centric members of Marshal type, 23
- Message boxes, 170 639–640
- MessageBox() Win32 API function, 639–640
- Metadata
 - translating COM into .NET, 249–250
 - viewing for an assembly, 245–246
 - viewing custom, 326
 - viewing for a derived type, 247–248
 - viewing for an enumeration, 246
 - viewing for an interface, 247
- Metadata descriptions, 248–249
- Metadata dump, 249
- Method parameters
 - COM to .NET conversion, 377–381
 - enumerating, 311–312
 - interface types as, 184–185
- Method signatures, converting, 555–556
- Method-centric members of InteropServices, 365
- MethodInfo.GetMethod() method, 336
- MethodInfo.GetParameters() method, 311–312

- MethodInfo.Invoke() method, 333
 - Methods
 - COM, 105
 - COM properties as, 106
 - handling overloaded, 569–570
 - interfaces containing, 106
 - invoking parameterized, 334–335
 - .NET, 375–377
 - Microsoft.Win32 namespace, 654
 - MIDL compiler, 65, 77
 - configuring, 77
 - core base types, 165–166
 - output, 66
 - MIDL compiler-generated files, 78–79
 - MIDL (Microsoft IDL), 165
 - Mixed mode debugging, 468–470
 - MMC (Microsoft Management Console)
 - snap-ins, 678
 - _Module (instance of CComModule), 614
 - Module-level attributes, 328–329
 - More Details menu
 - building, 316–322
 - submenus, 316
 - Mscoree.dll, 660–661
 - Mscoree.tlb, 660–662
 - Mscorlib.dll, 475, 571
 - Mscorlib.tlb, 477
 - importing, 570–572
 - interacting with, 586–589
 - referencing, 587
 - MSMQ (Microsoft Message Queue), 672
 - MTS (Microsoft Transaction Server), 669–670
 - MulticastDelegate class, 443–445
 - Multifile assemblies, 233
 - Multiple base interfaces
 - interfaces derived from, 301
 - .NET interface with, 624–627
 - Multiple behaviors, class supporting, 60
 - Multiple inheritance, 60
 - Multiple interfaces
 - class support for, 59–60
 - defining and supporting, 76
 - Multiple [source] interfaces
 - COM types with, 457–459
 - establishing, 607–608
- ## N
- Named mangling, defined, 11
 - Namespace definitions
 - dot notation for nested, 371
 - programming custom, 369–371
 - Namespace-building types of
 - CodeDOM, 274
 - Namespaces, that have existing
 - attributes, 324
 - Name/value pairs, enums as, 176, 213, 597
 - Nested classes, 300
 - Nested namespace definitions, dot notation for, 371
 - .NET
 - building blocks of, 231–232
 - [custom] wrapper, 519–521
 - error handling, 464–468
 - Garbage Collector, 714–715
 - [helpstrings], 509–510
 - late binding under, 331
 - philosophy of, 230–231
 - value type vs. reference type entities, 598
 - variable declarations in, 296
 - .NET application, running, 285–288
 - .NET application code, 287
 - .NET assemblies. *See* Assemblies
 - .NET attribute metadata, 517–519
 - .NET attributes, 323–325
 - building custom, 325–329
 - vs. IDL attributes, 323, 326
 - restricting use of, 327–328

- .NET binaries. See Assemblies
- .NET binding process, 269–270
- .NET Category, type assignment to, 578
- .NET class characteristics, 299
- .NET class interface, establishing, 559–561
- .NET class types, 299–300, 607–608
- .NET collection client (VB 6.0), 617–619
- .NET collections, custom, 614–619
- .NET collections and exceptions
 - handled by C++ COM, 621–623
- .NET component
 - accessing from C#, 699
 - accessing from VB 6.0, 698
- .NET data type language mappings, 296–297
- .NET data type system, 294–297
- .NET data types, building custom, 297–301
- .NET DataGrid type, 401–402
- .NET delegates, 298, 443–445
 - converting to COM connection points, 604–609
 - defined, 443
 - generated by tlbimp.exe, 450–451
- .NET enums, 301
 - converting to COM enums, 593–598
 - inheriting from System.Object, 597
 - mapping to COM IDL, 593–594
 - use of System.Int32 type, 595
- .NET event metadata, 447
- .NET event server, building using VB
 - .NET, 608–609
- .NET events, 445–448, 604–606
- .NET exceptions, 619–621
- .NET fields, converting, 557
- .NET interface hierarchies, converting, 627–630
- .NET interface inheritance, simulating, 627
- .NET interface types, 301
- .NET interfaces, 453
 - COM coclasses implementing, 475–484
 - COM type compatibility, 476
 - discrete, 358–359
 - exposing custom to COM, 564–566
 - implementing twice, 485
 - with multiple base interfaces, 624–627
 - registering custom, 580
 - tlbimp.exe-generated, 451–453
 - using to interact with COM types, 359–361
- .NET late binding syntax, vs. COM, 336–338
- .NET metadata, translating COM metadata into, 249–250
- .NET methods, converting interface members to, 375–377
- .NET namespace existing attributes, 324
- .NET project workspace, 256–257
- .NET properties, converting, 556
- .NET runtime, 331, 367, 660
- .NET runtime spy, 367
- .NET server, anatomy of, 229–288
- .NET shared assembly, versioning, 265–267
- .NET source code file format, 273
- .NET structure types, 300
- .NET structures, 300
 - as IDL unions, 603
 - converting to COM structures, 598–603
- .NET type assignment to .NET Category, 578
- .NET type compatible coclass, building, 476–479, 481–489
- .NET type viewer, building custom, 312–322

- .NET types, 289–338
 - binary-compatible C#, 641–642
 - binary-compatible VB .NET, 642–643
 - categories of, 298
 - COM-aware, 547–554
 - COM+-aware, 669–738
 - creating and configuring, 690–694
 - enumerating, 311, 622
 - exposing to COM applications, 633–667
 - implementing COM interfaces, 638
 - installing in a COM+ application, 693–694
 - in interop assembly, 356–362
 - managed representation, 36–37
 - viewing, 243
- .NET UDTs mapped to COM IDL
 - structures, 600
- .NET and Win32 data type
 - representation, 19
- .NET-to-COM communications proxy.
 - See RCW
- .NET-to-COM conversion, critical
 - details, 554–557
- .NET-to-COM IDL data type
 - conversions, 540–542
- .NET-to-COM interoperability
 - advanced, 471–538
 - basic, 339–402
 - high-level overview, 339–342
 - intermediate, 403–470
- New keyword, re-listing inherited
 - members using, 626, 629–630
- _NewEnum() method, 431–432
- NewGuid() method, 348
- Non-blittable data types, 349–351
- Nonconfigured component, defined, 671
- [Noncreatable] coclass, mapping, 387–388
- [Noncreatable] IDL keyword, 428
- Nonpoolable object lifecycle, 714–715

O

- Object Browser
 - IDL COM types in, 157
 - interop assembly in, 344
 - type information in, 104
- Object construction strings (COM+), 672, 704–706
- Object context, 672–673, 675
- _Object interface, 562–563, 695
- _Object interface members, 562
- Object map (server-wide), 136
- Object pooling, 715–717
- Object references, testing for equality, 292
- Object variables, scoped at class level, 708
- OBJECT_ENTRY macro, 136, 138
- ObjectHandle type, 664
- ObjectPooling attribute, 717
- ObsoleteAttribute type, 324–325
- Oleautomation, defined, 166
- Oleautomation data types, 166–167
- Oleautomation interface (dispinterface), 109–110, 166
 - defined, 108–109, 171
 - defining, 171–172
 - raw, 566, 604, 611
- [Oleautomation]-compatible COM
 - interface, 633
- [Oleautomation]-compatible types,
 - mapping, 348
- Oleaut32.dll (universal marshaler), 565, 581
- Oleview.exe utility (VB 6.0), 152–156
- OnTheEvent() method, 612
- OpenFileDialog type (Windows Forms), 530
- Out keyword (C#), 378
- OutAttribute type, converting, 556
- Outbound interface, 439–440

Overloaded Add() method, 586
 Overloaded methods, handling, 569–570
 Overridable members, translating,
 551–553

P

ParamArray keyword (VB .NET), 421
 ParamArrayAttribute type, 421
 Parameter arrays (COM), 420–421
 Parameter conversions, 379
 Parameter modifier decoder, 556
 Parameter-centric members of
 InteropServices, 365
 Parameterized constructors, explained,
 300
 Parameterized methods, invoking,
 334–335
 Parameters passed by reference (VB 6.0),
 379–381
 Params keyword (C#), 421
 Parent interface of COM interface, 624
 Partial strong name of an assembly, 335
 Passing structures, 35–37
 Path of COM (.NET philosophy), 230
 Path to custom DLL, 14
 Path of Java (.NET philosophy), 230
 PInvoke COM library function, 531
 PInvoke example, 26–33
 PInvoke (Platform Invocation), 1–49
 to access legacy binary modules, 49
 atoms of, 18–26
 Platform Invocation Services, 1–49
 Pointers
 array of, 57
 smart, 102
 using to trigger a callback, 43–44
 Policy assemblies, 267–270
 Polymorphism, 58, 61
 Poolable objects (COM+), 715–717

Populate() method
 (COMAdminCatalogClass), 676
 PopulateNamespace() helper method,
 281–284
 Primary interop assembly
 creating, 393–396
 determining, 396
 registering, 395–396
 strong name for, 394
 PrimaryInteropAssemblyAttribute type,
 394–395
 Primitive COM data types, 164–167
 Private assembly
 configuring, 251–253
 late binding to, 332–334
 prefixed with Interop, 343–344
 relocating, 252
 Private class members (C++), 54
 Private Collection member variable, 429
 Private components (COM+ 1.5),
 733–734
 Private default constructor, 387
 Private interop assemblies, 343–344, 392
 Procedure Attributes dialog box, 432,
 492–493
 ProgIDs (Programmatic Identifiers),
 91–92, 96, 140, 561, 575
 Project workspace (VB .NET), 256–257
 Project-wide imports, setting up, 257
 Properties (COM), 105–107
 from client's point of view, 107
 defined, 105
 mapping to .NET equivalent, 375–376
 Properties (.NET), converting, 556
 Proxy (.NET-to-COM communications).
 See RCW
 Public default constructor, explained,
 549
 Public entity, explained, 547
 Public key, 260

- Public members
 - exported structure field data as, 599
 - inheriting, 553–554
- Public structure members (C++), 54
- [publickey] tag, 692
- PublicNotCreatable (Instanting property), 428
- Publisher, explained, 267
- Publisher policy, explained, 267
- Publisher policy assemblies, 267–270
- Pure virtual functions, defined, 54

Q

- QC (Queued Components), 672
- QueryInterface() method, 73, 82, 86, 105

R

- RaiseEvent keyword (VB .NET), 608
- Random type, 478
- Raw disinterface, 171–172, 566, 604, 611
- RCW (Runtime Callable Wrapper), 340, 539
 - for each coclass, 341
 - interfaces consumed by, 351–353
 - responsibilities of, 342
 - role of, 340–342
- RCW translator, 218
- Ref keyword (C#), 377–378, 422
- Reference type (heap-based) entities, 598
- ReferenceEquals() method, 290, 292
- Reflection, defined, 203
- Reflection namespace, 304, 309
- Reflection namespace members, 309
- Reflection.Emit, 323
- Regasm.exe utility, 395, 572–574, 578
 - interacting with, 653–655
 - key flags, 573
 - updated entries, 574–582
- Registering (in the registry)
 - a COM server, 95–97
 - the COM type library, 582
 - a COM+ application, 696–697
 - exposed interfaces, 579–582
 - a .NET assembly, 545–546, 644
 - a primary interop assembly, 395–396
 - a type, 124
- Registration, lazy (automatic), 700–701
- Registration of COM server, VB 6.0
 - automatic, 151
- Registration of interop assembly,
 - interacting with, 653–655
- Registration-centric members of
 - InteropServices, 363–364
- RegistrationHelper type, 693, 701–703
- RegistrationServices type, 655
- Registry, 91. *See also* Registering (in the registry)
 - role of, 66
 - updated entries in, 574–582
- Registry Editor (regedit.exe), 91
- Registry hives, 91
- Registry keys, 91
- Registry subkeys, 91
- REGKIND enumeration, 531–532
- Regsvcs.exe utility, 681, 693, 694–698
 - /appname flag, 695
 - core flags, 694
 - /fc flag, 695
 - updating the COM+ Catalog, 697
 - updating the registry, 696
- Release() method, 81–82, 85, 100, 196
- ReleaseComObject() method, 474–475
- [.remove] directive, 454
- RemoveAt() method (ArrayList), 619–620
- [.removeon] directive, 448
- ReportCOMError() helper function, 467
- ReportEvent() method, 535
- ResolveRef() method, 535–536

Root object, in transaction processing, 718

Runtime

COM type generation at, 161, 189–191, 201–203
.NET, 331, 367, 660
reading attributes at, 207, 330–335
reordering fields at, 35

Runtime environment (COM+), 672–675

Runtime spy (.NET), 367

Runtime-centric members of

InteropServices, 366

RuntimeEnvironment type, 366–367

S

SAFEARRAY COM library functions, 181–182

SAFEARRAY helper templates (ATL 4.0), 184

SAFEARRAY structure, 180, 183

SAFEARRAYBOUND structure, 180

SAFEARRAYs, 180–181, 410–418, 424

from managed code, 413–418

mapped to System.Array, 348, 414

SayHello() method, building, 198–200

Scriptable object, 108–112, 118–122

Secondary objects (in transactions), 718–719

SecurityCallContext type, 674

SEH (structured exception handling), 464

Self-describing entities, 234

Server lifetime, managing, 88–89

Serviced component example, 724–736

ASP.NET Web Service client, 734–736

C# code library, 726

CarInventory class type, 728–731

CarsSold table, 726

custom database, 725–726

design notes, 724

Inventory table, 725

LogSale type, 727–728

Windows Forms front end, 732–734

Serviced components, building, 669–738

ServicedComponent type

(EnterpriseServices), 689–690

Serviced.Component.Construct()

method, 704

SetErrorInfo() COM library function, 461

SetLastError field (DllImportAttribute), 32–33

SetType() helper method, 319

Shared assembly, 254–267, 261, 393

late binding to, 335–338

placed into the GAC, 255

recording, 263

using, 262–263

versioning, 264–267

Shared interop assemblies, 393

Shared name. *See* Strong name

ShowMemberStats() helper function, 318–319

ShowTypeStats() helper method, 315

Single-file assemblies, 233

Smart pointers, 102

Sn.exe utility, 392

Solution Explorer, 497

[Source] interface, 439, 442

COM types with multiple, 457–459

establishing multiple, 607–608

IDL definition of, 440

[Source] keyword (IDL), 604

SPM (shared property manager), COM+, 688

Square brackets ([]), use of, 72

Stack-based entities, 598

Stateless COM+ type, explained, 709

Stateless entities, configured

components as, 671

Static members, translating, 554

String conversion macros (ATL), 179

- String conversion members of Marshal type, 22
- String name, friendly, 310
- String type (System.String), 348, 351, 634
- String-centric values of
 - UnmanagedType, 634
- Strong name, 255, 260–262
 - for an interop assembly, 394, 640
 - for a .NET assembly, 255–256, 335, 640, 692
 - for a primary interop assembly, 394
- Strongly typed variables, interfaces as, 58–59
- Struct keyword (C#), 598
- StructLayout attribute, 35, 603
- Structure details, displaying, 317
- Structure field data exported to COM IDL, 599
- Structure keyword (VB .NET), 598
- Structure members (C++), 54
- Structure types (.NET), 300
- Structures
 - with blittable fields, 350
 - COM, 421–425
 - converting .NET to COM, 598–603
 - functions receiving, 7–8
 - with non-blittable fields, 351
 - passed by reference, 602
 - passing, 35–37, 602
 - receiving allocated, 37–39
- Structures containing structures, 7–8
- Stub and proxy DLL, custom, 581–582
- Stub code, VB 6.0 IDE used to generate default, 150
- Stub/proxy files (MIDL output), 66
- Subkeys (registry), 91
- System data type language mappings, 296–297
- System path variable, 13
- System registry. *See* Registering (in the registry); Registry
- System.Activator class, 331–332
- System.Activator class members, 332
- System.Array type, 416–417
 - mapping SAFEARRAYS to, 348, 414
 - members of, 414
- System.Attribute base class, 517
- System.Attribute core members, 324
- System.Attribute-derived type, 525–526
- System.Byte type, 595
- System.CodeDOM namespace 270–284.
 - See also* CodeDOM
- System.Collections namespace, 434–435, 615–616
- System.Collections.dll, 665–666
- System._ComObject, role of, 399
- System.EnterpriseServices namespace, 687–690, 701, 703, 736–737
- System.EnterpriseServices.dll, 669, 691, 699
- System.Enum base class type, 301, 597
- System.Exception base class, 465, 619
- System.Exception type members, 465
- System.Guid mappings, 348–349
- System.IComparable interface, 476
- System.IntPtr type, 420
- System.Int32 type, 595
- System-level DLLs, 12
- System.MulticastDelegate class, 443–445
- System.Object
 - coclasses derived from, 389
 - methods of, 290
 - .NET enum inheriting from, 597
 - role of, 289–294
 - variable data types, 404
 - VARIANTs mapped to, 407
- System.Object members, inherited, 390, 597

- System.Object-centric values of
 - UnmanagedType, 635
- System.Object.Finalize() method, 714–715
- System.Object.GetType() method, 306, 398
- System.Object.ToString() method, 239, 563
- System.ObsoleteAttribute type, 324–325
- System.ParamArrayAttribute type, 421
- System.Random type, 478
- System.Reflection namespace, 304, 309, 323
- System.Reflection namespace members, 309
- System.Reflection.Emit, 323
- System.Runtime.InteropServices. *See* InteropServices namespace
- System.String type, 348, 351, 634
- System.Type class, 304–308, 418, 588
- System.Type class members, 305
- System.Type reference, obtaining, 306–307
- System.Type.GetCustomAttributes() method, 330
- System.Type.GetCustomAttributes() method, 330
- System.Type.GetType() method, 307–308
- System.Type.Missing read-only field, 384–385
- System.ValueType, 294, 422
- System.ValueType-derived types, tlbexp.exe and, 599
- System.Windows.Forms.AxHost base class, 499

T

- TheEnum type, 326, 330
- Tlbexp.exe (Type Library Exporter)
 - utility, 475
 - building a custom version of, 655–660
 - [dual] interface with DISPIDs, 565
 - and System.ValueType-derived types, 599
 - using, 546–547
- Tlbimp.exe (Type Library Importer)
 - utility, 342, 353–355, 367, 448–459, 508–509
 - building an interop assembly with, 354–355
 - core options of, 354
 - custom IDL attribute for ProgID, 561
- ToString() method, 239, 290, 597
 - overriding, 290–291
 - transforming, 563
- TPM (Transaction Processing Monitor), 718
- Transaction
 - ACID properties of, 717
 - defined, 717
 - enlisting multiple objects, 719
 - single object, 718
- Transaction attribute, 721–722
- Transaction processing, and root object, 718
- Transactional COM+ settings, 721
- Transactional programming, 717–724
- Transactional programming (COM+), 720–724
- TransactionOption enumeration, 721–722
- Type class, 80, 304–308
- Type compatible (COM type with .NET interface), 476

- Type information, 161–228
 - as binary IDL, 65
 - displaying details, 315–316
 - dumping, 207–208
 - generating programmatically, 189–191
 - located under HKCR\TypeLib, 94
 - obtaining for a COM wrapper type, 398
 - reading programmatically, 203–212
 - viewing in the Object Browser, 104
- Type information generation, testing, 201–203
- Type information viewer, in C#, 220–227
- Type libraries
 - as binary IDL, 78
 - building, 191–193
 - defined, 65
 - library statement section, 368–371
 - registering, 582
 - role of, 65–66
 - [version] identifier, 368
- Type library attributes, reading at runtime, 207
- Type library browser application
 - building, 203–212
 - displaying information, 205–207
 - dumping COM type information, 207–208
 - listing coclass statistics, 208
 - listing COM enumeration statistics, 209–210
 - listing IDispatch interface statistics, 209
 - listing IUnknown interface statistics, 209
 - program skeleton, 204–205
 - reading, 210–212
- Type library creation elements, 189
- Type library importer utility, building, 528–538. *See also* Tlbimp.exe
- Type library statement name, changing, 568
- Type library-centric COM library items, 204
- Type library-centric members of InteropServices, 363
- Type library-centric members of Marshal class, 21
- Type marshaling, 633–637
- Type member visibility
 - controlling, 548–549
 - establishing, 548
- Type members, displaying details about, 316–322
- Type metadata, viewing, 245–246
- Type names in an assembly, displaying, 313
- Type reference
 - from C# typeof operator, 306–307
 - from System.Object.GetType(), 306
 - from System.Type.GetType(), 307–308
- Type viewer (custom), 312–322
 - ADO.NET types in, 322
 - custom dialog GUI, 318
 - displaying assembly details, 317
 - displaying assembly information, 316–322
 - displaying class member information, 320
 - displaying class member parameters, 321–322
 - displaying enumeration fields, 320
 - displaying type details, 315–316
 - displaying type names, 313, 315
 - More Details menu, 316–322
- Type visibility
 - controlling with ComVisibleAttribute, 548–549
 - establishing, 547
- Type-building types of CodeDOM, 274
- TYPEFLAGS enumeration, 197–198, 373

TYPEFLAGS values, 197–198
 Type.GetMembers() method, 311
 TYPEKIND enumeration, 196, 207
 TYPEKIND structure, 201, 207, 226
 TYPelibATTR structure, 224
 TypeLibConverter class, 528–530,
 533–535, 655–656
 TypeLibConverter.ConvertTypeLibToAss-
 embly(), 533–535
 TypeLibImporterFlags enumeration, 534
 TypeLibTypeAttribute type, 373
 Typeof operator (C#), 306–307
 Types. *See* COM type; Data types; .NET
 types; Type information
 Types hierarchy, 237, 295

U

UCOM (unmanaged COM) prefix, 220
 UCOMTypeLib interface, 537, 659
 UDTs (user-defined types), 3, 163, 600.
 See also Structures
 ULONG, global, 87
 Unadvise() method, 438–439
 Unicode characters, 18, 178
 UninstallAssembly() method, 701
 Unions, .NET structures as, 603
 Universal marshaler (oleaut32.dll), 565,
 581
 Universal marshaling, 565, 580–581
 Unmanaged callbacks, 42–43
 Unmanaged code, 1–2, 232
 UnmanagedAssembly.dll, 666
 UnmanagedType
 array-centric value of, 636
 data-centric values of, 636–637
 string-centric values of, 634
 System.Object-centric values of, 635
 UnmanagedType.Currency value, 637
 UnregisteredAssembly namespace,
 665–666
 Unsigned char mapped into a VB 6.0
 Byte, 596
 Unwrap() method, 664
 Updating interop assemblies, 522–524
 USES_CONVERSION macro, 179

V

Value type (stack-based) entities, 598
 ValueType type, 294
 [Vararg] IDL attribute, 420–421
 Variable declarations, in .NET, 296
 Variables, scoped at class level, 708
 VARIANT array, 621
 VARIANT COM library functions, 115
 Variant compliant types, 166
 VARIANT data type, 112–116, 166–167,
 217, 384–385, 403–410
 in C++, 114–115
 from managed code, 407–409
 mapped to System.Object, 407
 in VB 6.0, 115
 VARIANT field, 216
 VARIANT structure, 112–114
 VARIANT vt field, .NET data types
 setting, 404
 VARIANT wrappers, 409
 VARIANT-centric COM server, building,
 405–410
 VariantInit() COM library function, 114
 Varying C-style arrays, 419
 VB COM type, preventing direct creation
 of, 428
 VB .NET application code, 287
 VB .NET binary installed in the GAC, 262
 VB .NET client interop assembly, 346
 VB .NET code library hierarchy, 256
 VB .NET IDE, Implement Interface
 Wizard, 643
 VB .NET .NET event server, 608–609
 VB .NET project workspace, 256–257

- VB .NET shared assembly, versioning, 265–267
- VB .NET type, building binary-compatible, 642–643
- VB .NET (Visual Basic .NET)
 - as a managed language, 232
 - byRef keyword, 377–378, 380–381
 - byVal keyword, 377–381
 - completed application, 286
 - Event keyword, 609
 - Handles keyword, 456–457
 - intercepting incoming COM events, 456–457
 - ParamArray keyword, 421
 - RaiseEvent keyword, 608
 - running application, 285–288
 - Structure keyword, 598
 - WithEvents keyword, 456–457
- VB 6.0 Byte
 - building, 644–646
 - unsigned char mapped into, 596
- VB 6.0 client methods, 169–170
- VB 6.0 COM client, 103–105, 157, 584–589, 644–646
- VB 6.0 COM server
 - reading, 212
 - testing, 156–159
- VB 6.0 COM types, locating, 153
- VB 6.0 COM+ client, building, 683
- VB 6.0 COM-supported COM interfaces, 154
- VB 6.0 custom CLR host, 663–667
- VB 6.0 event client, building, 609–610
- VB 6.0 form, code behind, 158
- VB 6.0 IDE, using to generate default stub code, 150
- VB 6.0 .NET collection client, 617–619
- VB 6.0 structure server, building, 423–424
- VB 6.0 (Visual Basic 6.0)
 - accessing configured .NET component, 698
 - ActiveX control, 490–493
 - application object, 176
 - applying IDL [helpstrings], 492
 - automatic registration of COM server, 151
 - binary compatibility, 151–152
 - building COM servers using, 146–148
 - CheckThisVariant() method, 469–470
 - coclass COM event atom support, 442
 - Collection type, 429, 645
 - core COM project types, 147
 - defining auxiliary interfaces, 148–149
 - defining and sending events, 441
 - disallowing structures passed by value, 601–602
 - Event keyword, 442
 - IDispatch client, 117
 - implementing interfaces in, 149–151
 - and interfaces with underbars, 489
 - LameColorControl, 495–496
 - Oleview.exe utility, 152–156
 - parameters passed by reference, 379–381
 - role of, 146–159
 - setting DISPID_BACKCOLOR, 492, 493
 - VARIANTs in, 115, 481
 - WithEvents keyword, 610
- VBScript COM client, building, 590–591
- VBScript late bound client, building, 124–126
- _VBStructObject interface, 424
- [Version] identifier of COM type library, 368
- Version number (strong name), 260
- Versioned interfaces, 61–62, 373–374
- Versioning shared assemblies, 264–267

- Virtual functions, pure, 54
- Visibility-centric members of
 - InteropServices, 363–364
- VS .NET IDE, 584
- VS. NET (Visual Studio .NET), 343
 - consuming ActiveX controls, 495–501
 - debugging COM servers, 468–470
 - managed languages, 232
 - private interop assemblies, 344
 - referencing a COM server using, 343
- Vtable, 651

W

- Web Service client (ASP.NET), 734–736
- Well-known category, grouping COM
 - objects into, 577–578
- Win32 *.def file, assembling standard, 90
- Win32 API functions, library-centric, 15
- Win32 callback functions, 42
- Win32 console application project,
 - creating, 52–53
- Win32 DLLs, location of core, 13
- Win32 error, obtaining the last, 32
- Win32 error code as friendly text string,
 - 32
- Win32 namespace, 654
- Win32 and .NET data type
 - representation, 19
- Win32 structure, managed equivalent of,
 - 35
- WithEvents keyword (VB .NET), 456–457
- WithEvents keyword (VB 6.0), 610
- WSDL (Web Service Description
 - Language), 270–272
- Wsdll.exe utility, 270–272
- W2A (Unicode to ANSI) macro, 179



<http://www.springer.com/978-1-59059-011-9>

COM and .NET Interoperability

Troelsen, A.

2002, XXIX, 816 p., Softcover

ISBN: 978-1-59059-011-9

A product of Apress