CHAPTER 26

Extending ColdFusion with COM

IN THIS CHAPTER

One commonly overlooked feature of ColdFusion is its ability to interact with objects external to the ColdFusion Application Server. These objects can be third-party software packages such as Microsoft Office, or they can be custom objects created by the developer. Although external objects come in different flavors, such as COM/DCOM and CORBA, this chapter focuses on the implementation of COM in ColdFusion.

COM (Component Object Model) and DCOM (Distributed Component Object Model) are architecture specifications developed by Microsoft, used primarily on the Windows platform. COM enables applications to “talk” with one another through a set of interfaces such as methods and properties. DCOM, similarly, is for remote distribution of these same interfaces; for example, accessing an object remotely over a network. By developing COM objects that adhere to the COM specification, you can overcome some of the issues inherent in building any application:

- Extending functionality without damaging the application
- Removing, upgrading, and replacing features easily
- Integration of new applications with existing applications
- Building applications with more than one programming language

Understanding COM

In the context of a ColdFusion Application, ColdFusion MX 7 is the client, the object is the server, and the COM automation system is the liaison between the two. With a fundamental understanding of `<cfobject>` and the objects you are using, you can easily employ COM in a ColdFusion application. Following are some of the most basic benefits of using COM in your applications:

- Accessing functionality otherwise unavailable to ColdFusion

- Building files on the fly with applications such as Microsoft Word and Microsoft Excel
- Performing complex operations that are better suited for the speed benefits of compiled objects, such as EXE files and DLL files

NOTE

Although third-party implementations of COM do exist on UNIX systems, ColdFusion's support for COM is limited to the Windows platform. In short, this chapter applies only to ColdFusion MX 7 for Windows.

Working with COM Objects in ColdFusion

There are two ways to work with COM objects in ColdFusion. Both accomplish the same result: returning an instance of the COM object that you want to work with. The method you choose is mainly a matter of personal preference. You can use either of the following interchangeably:

- The `<cfobject>` tag
- The `CreateObject()` function

To work with COM using `<cfscript>` syntax, you need to use the `CreateObject()` function; otherwise, the `<cfobject>` tag will probably feel more familiar to you and look more consistent with the rest of your CFML code. You could just as easily use the `CreateObject()` function within normal CFML code by using the `<cfset>` tag.

Using COM with `<cfobject>`

You can use the `<cfobject>` tag as you would any other ColdFusion tag, except that `<cfobject>` does not require a closing tag. Table 26.1 lists the attributes available to the `<cfobject>` tag, and their descriptions.

Table 26.1 `<cfobject>` Tag Syntax

ATTRIBUTE	DESCRIPTION
`action`	Required. Values are `CREATE` or `CONNECT`. Use `CREATE` to instantiate a COM object (typically a DLL) prior to invoking its methods and properties. Use `CONNECT` to connect to a COM object (typically an EXE) that is already running on the server, specified in the `server` attribute.
`class`	Required. The program identifier (`PROGID`) for the object you want to create or connect to. If the object resides on a remote server, you will use the `server` and `class` attributes, specifying the class identifier (`CLSID`) for the object. If a Java stub is used, use the `PROGID` of the COM object.
`name`	Required. An arbitrary value used to reference the object. This acts as the scope for all the object's operations in the code following the call to `<cfobject>`.

Table 26.1 (continued)

ATTRIBUTE	DESCRIPTION
`type`	Optional. The type of object represented by the `class` attribute. Values are `COM`, `CORBA`, or `Java`. `COM` is the default and is the value discussed in this chapter. You will learn about `type="CORBA"` in Chapter 28, “Extending ColdFusion with CORBA,” and `type="Java"` in Chapter 29, “Extending ColdFusion with Java.”
`context`	Optional. Possible values are `INPROC`, `LOCAL`, and `REMOTE`. `INPROC` is an In-Process server object (typically a DLL) that is running in the same process space as the calling process, such as ColdFusion. `LOCAL` is an Out-of-Process server object (typically an EXE) that is running outside the process space, locally on the server. `REMOTE` is the same as `LOCAL`, except that the object resides on a remote server, specified in the `server` attribute.

`server`	Optional. It is required when `context="Remote"`. This represents the server hosting the object you want to instantiate. Enter a valid server name using UNC (Universal Naming Convention) or DNS (Domain Name Server) conventions, in one of the following forms:
	`server="\\lanserver"`
	`server="lanserver"`
	`server="http://www.servername.com"`
	`server="www.servername.com"`
	`server="127.0.0.1"`

Correct `<cfobject>` usage looks like the following:

```
<cfobject
 type="COM"
 action="Create"
 class="ProgID"
 name="Object Name"
 server="Server"
 context="InProc">
```

Using COM with `CreateObject()`

An alternative way of using COM objects is through the `CreateObject()` function. For users more comfortable using scripting, this syntax might be preferable. If you are porting COM code from another scripting language such as ASP, you will find `CreateObject()` to be simpler and cleaner to implement. Table 26.2 lists the parameters available to the `CreateObjct()` function, in the order that they must be included, and their descriptions.

Table 26.2 `CreateObject()` Function Syntax

PARAMETER	DESCRIPTION
`type`	Required. The type of object represented by the `class` attribute. Values are `COM`, `CORBA`, `Java, component, and werbservice`.
`class`	Required. The program identifier (`ProgID`) for the object you want to create or connect to. If the object resides on a remote server, you will use the `server` and `class` attributes, specifying the class identifier (`CLSID`) for the object. If a Java stub is used, use the `ProgID` of the COM object.
`context`	Optional. Possible values are `INPROC`, `LOCAL`, and `REMOTE`. `INPROC` is an In-Process server object (typically a DLL) that is running in the same process space as the calling process, such as ColdFusion. `LOCAL` is an Out-of-Process server object (typically an EXE) that is running outside the process space, locally on the server. `remote` is the same as `LOCAL`, except that the object resides on a remote server, specified in the `server` attribute.

`server`	Optional. It is required when `context="Remote"`. This represents the server hosting the object you want to instantiate. Enter a valid server name using UNC (Universal Naming Convention) or DNS (Domain Name Server) conventions, in one of the following forms:
	`"\\lanserver"`
	`"lanserver"`
	`"http://www.servername.com"`
	`"www.servername.com"`
	`"127.0.0.1"`

When using `COM` as the value in the `type` attribute, `CreateObject()` takes four parameters, as follows (the third and fourth parameters are optional):

```
<cfscript>
 objectName = CreateObject(
 "COM",
 "ProgID",
 "InProc",
 "Server"
 );
</cfscrijpt>
```

➜ To learn more about `<cfscript>`, see Chapter 12, "ColdFusion Scripting."

Table 26.1 discusses the purpose of each of these parameters. At a minimum, you must specify the `TYPE` and `CLASS` of the object; optionally, you can also specify the `server` and `context` attributes.

The `objectName` variable shown in the preceding example represents the name you are assigning as the instance of that object. Use this variable to refer to the object later in your code:

```
<cfscript>
 // Create the object instance
 objectName = CreateObject(
 "COM",
 "ProgID"
 );

 // Set a variable to a method's result
 myVar = objectName.Method();
</cfscript>
```

The first step in any situation that requires COM is to connect to or create the object on the server. As you have already seen, the `<cfobject>` tag or the `CreateObject()` function handles this process:

```
<cfobject
 action="Create"
 class="Car.Builder"
 name="objCarBuilder"
 type="COM">
```

This code creates an instance of a fictitious `"Car Builder"` object on the server, at which point you may begin accessing its properties and methods. You will see real-world examples using COM in Chapter 27, "Integrating with Microsoft Office," but for now, the examples in the next few sections will use an imaginary object to show COM's syntax and object hierarchies.

Setting and Retrieving Properties

A *property* is essentially a single attribute or *characteristic* of an object. To set or get a property, you must know which object you are using, as well as the properties that object exposes.

➜ See the documentation for your object to view the object hierarchies and supported properties in more depth.

If you have never used an object-oriented language, such as Java or C++, then the idea of properties might be new to you. To see how this works, you could use a car as an example. Following is a simple object "road map" for getting a car's exterior color:

```
extColor = objCarBuilder.Car.Body.Paint.Color
```

The variable `extColor` is set to the value held within the `Color` property. To get to the car's color property, you first have to drill down through the object's hierarchy until you have reached the object that contains the property. In this example, the `Paint` object contains the `Color` property and could possibly contain several other properties as well. The `Color` property is merely one possible property defined in the `Paint` object:

```
objCarBuilder.Car.Body.Paint.Color
objCarBuilder.Car.Body.Paint.Brand
objCarBuilder.Car.Body.Paint.Finish
```

As you can see, the theoretical `Paint` object contains not only the `Color` property but the `Brand` and `Finish` properties as well.

NOTE

For the discussion that follows, the term "nested attribute" is used to refer to any attribute that needs more than one dot (period character) after the actual COM object reference to identify the attribute. In this theoretical example, then, `Body`, `Paint`, `Color`, `Brand`, and `Finish` are all nested attributes.

Nested Attributes and Previous Versions of ColdFusion

In past versions of ColdFusion (ColdFusion version 5 or below), getting the `Color` property for the `Paint` object has required a little more work than in the first example. You would need to use several separate `<cfset>` lines, as shown here:

```
<cfset objCar = objCarBuilder.Car>
<cfset objBody = objCar.Body>
<cfset objPaint = objBody.Paint>
<cfset extColor = objPaint.Color>
```

Previous versions of ColdFusion did not have the capability to access the `Color` property directly from the `Paint` object. You would have to drill down each object level, one level at a time, by setting arbitrary variables for each level as shown above. This example uses the variables `objCar`, `objBody`, and `objPaint` to represent each object level. (You could use any valid variable name you wanted, but for consistency, it was best to use a name that described the object level it represented.)

To set the value of the `Color` property, you would do the following:

```
<cfset objCar = objCarBuilder.Car>
<cfset objBody = objCar.Body>
<cfset objPaint = objBody.Paint>
<cfset objPaint.Color = "red">
```

Nested Attributes and ColdFusion MX 7

ColdFusion MX 7 allows you to access nested attributes directly without having to drill down within the object hierarchy. Here is an example in ColdFusion MX 7:

```
<cfset extColor = objCarBuilder.Car.Body.Paint.Color>
```

Similarly, you can set a property's value using a single line:

```
<cfset objCarBuilder.Car.Body.Paint.Color = "Purple">
```

Occasionally, you might find that ColdFusion has trouble knowing how to work with nested objects when accessed in this manner. Such problems generally occur when one or more of the intermediary objects (in this example, the parts between `objCarBuilder` and `Color`) do not always return the same type of value. The problem usually manifests in an error message that reads "Method selection error" or something similar. If you have this problem when accessing nested properties, try the older, multistep syntax (several `<cfset>` tags, one for each nesting level).

Using Methods

The `Color` example in the last section shows you how to fetch and set a property of an object. Like properties, objects also contain methods (functions) that perform specific tasks. Methods can take optional and required parameters, or they can be stand-alone routines. Either way, you invoke them in the same fashion.

NOTE

Some programming languages allow you to use a method without parentheses if it takes no parameters, as in `myObject.Close`. ColdFusion, however, requires that all methods end in parentheses, as in `myObject.Close()`, even if the object does not require them.

To invoke a method, you must first know which object hosts the method you want to use. The `Paint` object used earlier also contains a method called `getDefaultColor()`:

```
<cfset extColor = objPaint.getDefaultColor()>
```

If the default color for our car builder happens to be `"blue"`, then the `getDefaultColor()` method would set `extColor` to `"blue"`. In this context, the method specifically performs a single task, which is to contact the `Paint` object, ask what the default color is, and then return it to the `extColor` variable. This type of method is a one-way process. In other words, it returns a value and does not allow you to set the value.

To set the default color, the `Paint` object also has a method called `setDefaultColor()`. In this situation, the method accepts a string value representing a valid color name:

```
<cfset objPaint.setDefaultColor("red")>
```

This example may seem wrong to you. Unlike the `getDefaultColor()` operation, the `setDefaultColor()` expression does not use a variable before the invoked method. That is, when setting a one-way value with a method, you don't have to create a variable representing the operation. If you prefer, you can set a variable to the method call, but creating a variable in this context is unnecessary:

```
<cfset Temp = objPaint.setDefaultColor("red")>
```

The last type of method you will encounter accepts and returns values. Depending on the specific method, it might accept a single or multiple arguments, and it might return a single value or possibly a collection of values. To illustrate this, imagine that the `Paint` object has a method called `getColorShades()` that returns a collection of information based on the values it receives. The syntax for its arguments is as follows:

```
getColorShades("Color", intMaxRecords, boolReturnPrices)
```

This method takes three arguments: a string representing a color to compare and find similar colors for, an integer that sets the maximum number of records to return, and a Boolean value indicating whether to return pricing information.

The return value for this method is a collection of values that matches the criteria specified in the method's arguments. To see this, follow the next example:

```
<cfset objShades = objPaint.getColorShades("red", 10, True)>
```

NOTE

Some documentation may indicate a method argument is optional. With some objects, you must still supply the optional arguments even if they are not explicitly required. Without adequate documentation, this may be a matter of trial and error, so you will need to test to see which optional arguments throw an error when omitted.

The newly created object `objShades` now represents a collection of information returned from the method. To view the information, you have to use `<cfloop>`'s `collection` attribute:

```
<table>
 <tr>
 <td>Shade</td>
 <td>Price</td>
 </tr>
```

```
<cfloop collection="#objShades#" item="Shade">
 <tr>
 <td>#Shade.Name#</td>
 <td>#Shade.Price#</td>
 </tr>
</cfloop>
</table>
```

Collections returned from COM objects are arrays of structures. When looping through a COM collection, each item (`item="Shade"`) is a structure that has its own properties (`Shade.Name` and `Shade.Price`).

NOTE

With Visual Basic syntax, values sent to methods often consist of named values, such as `myObject.Open(vbOption)`. In ColdFusion, you must use the numerical equivalent of that named value, as in `myObject.Open(2)`, where the `(2)` represents the numerical equivalent of `vbOption`. More information on the numerical values is available in Microsoft Visual Studio or online at the MSDN Library site: `http://msdn.microsoft.com`.

ColdFusion Is Mostly Typeless; COM Is Mostly Not

Like any application accessing COM objects, ColdFusion must present data to the object in a format that is recognizable by that object's interfaces. An interface defines the format and type of data that an object can receive and send back to client, and in this case, the client is ColdFusion. Think of an interface as being a value sent or received from a function—some functions pass or fetch simple values, such as strings and numbers; however, others might use complex values such as arrays or structures. Without knowing an object's interfaces, you may accidentally send data to an object in a format that is incorrect for that particular interface, resulting in an error or inaccurate returned data.

To put this in perspective, recall that ColdFusion is a typeless environment, meaning that you do not explicitly set a variable's data type. Because of this feature, it's up to you to make sure that the data sent to the object matches the object's requirements. On the other hand, the COM object may provide an interface to a method, for example, that expects an ambiguous value such as a variant (a data type commonly used in Visual Basic). So how does ColdFusion know what data type to send to the object? For objects, arrays, and strings, ColdFusion casts the values correctly. For integers, however, there is no guarantee that ColdFusion is sending the type expected by the object. If the object's method is expecting a data type of `short`, ColdFusion may pass a data type of `real`, due to the way ColdFusion internally represents numbers.

TIP

If you happen to be the developer creating the COM objects, it is easy to change the variable typing information. As a rule of thumb, always use strong variable typing when you know what the object is expecting.

Registering Objects

To begin using COM objects, make sure that the objects exist on the server or on a remote server to which your Web server has trusted access. If you have purchased a third-party software package, follow its instructions for installation and setup. In most cases, this will be enough to get you started, but sometimes you will be required to register an object manually on the server.

As noted earlier, objects come in two flavors, In-Process (InProc) and Out-of-Process (Local). InProc object servers are typically DLL and OCX files; Local object servers are usually EXE files located on the server. The methods used to register each type of object differ slightly.

To register an InProc object server manually (DLL and OCX files), you use the `regsvr32.exe` file included with the Windows operating systems currently supported by ColdFusion. You can run the command through a standard command prompt (DOS) or by choosing Start, Run, and then typing in the command.

NOTE

OCX files are runtime ActiveX controls primarily used and scripted on the client side using JScript or VBScript. To use an OCX control, use the HTML `<embed>` or `<object>` tags. See the HTML reference included in Macromedia Dreamweaver for more information. As a rule, do not use OCX files with `<cfobject>`.

Following is the usage for the `regsvr32.exe` file, with the parameters explained in Table 26.3:

```
regsvr32 [/u] [/s] [/n] [/i[:cmdline]] dllname
```

Table 26.3 `regsvr32.exe` Switches

SWITCH	DESCRIPTION
`/u`	Un-register the server
`/s`	Silent; display no message boxes
`/i`	Call `DllInstall`, passing it an optional `[cmdline]`; when used with `/u`, it calls a DLL uninstall
`/n`	Do not call `DllRegisterServer`; used with `/i`

Putting this to use, you call the actual DLL filename:

```
regsvr32 c:\path\servername.dll
```

The `servername.dll` represents the file you want to register, found in the `path` directory. The file does not have to be located on the `c:\` drive—you can specify any valid location to the object.

To register Local (Out-of-Process) object servers manually (EXE files), either you start them by double-clicking the executable file itself, or you can run a command line:

```
c:\path\servername.exe -register
```

Again, you specify the path to the executable file, and use the `-register` switch to register the object.

Upon issuing either registration command, you see a message saying that the object registered successfully. When all objects required for your application are registered, you are ready to put them to use.

NOTE

Some COM+ and DCOM objects on Windows 2000, XP, and 2003 systems require that you use the `clireg32.exe` program rather than `regsvr32.exe`. See the object's documentation for specific information regarding registration.

Viewing Objects with OLEView

Upon installing your object, you will need a way to see what methods, properties, and collections the object supports. Understanding this necessity, Microsoft created a program called the Object Viewer, more commonly known to developers as OLEView.

NOTE

OLEView is included with Microsoft Visual Studio. If you use Visual Basic, Visual C++, or other tools in the suite, you may already have OLEView on your system. If you do not have it, you can download a copy from Microsoft's Web site at `http://www.microsoft.com/windows2000/techinfo/reskit/tools/existing/oleview-o.asp`

OLEView gathers information about all the installed COM objects on your system. From this information, you can retrieve the `ProgID`, `CLSID`, and other data for the object you are using. As a feature, OLEView groups and sorts objects based on each component category, such as Document Objects or Automation Objects. For example, you will find Microsoft Word and Excel are listed under the Document Objects category. This is handy if you have a general idea of what your object performs; otherwise, it can be time consuming to sift through the literally hundreds of objects listed.

NOTE

Versions of Microsoft Office before XP and 2000 place the Excel object in the Automation Objects category in OLEView.

The default OLEView screen consists of two panes: The left shows the objects and categories; the right displays object information (Figure 26.1).

The first thing you'll need to gather is the object's `ProgID`. Think of the `ProgID` as a kind of map that tells the `<cfobject>` tag where to look for the object in the system Registry. For instance, the object information pane shows that the `ProgID` for the Microsoft Word object is `Word.Application.10` (Figure 26.2).

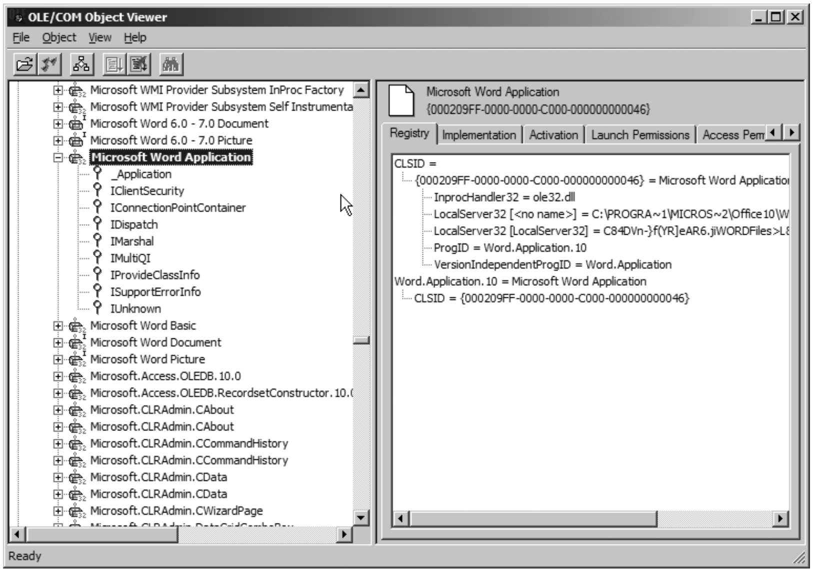


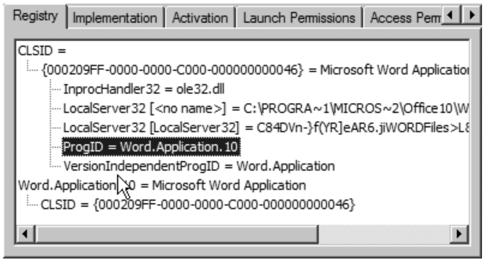
Figure 26.1

The OLEView program has two sections: the object categories and object information.

Microsoft Word Application
{000209FF-0000-0000-C000-000000000046}

```
CLSID =
  {000209FF-0000-0000-C000-000000000046} = Microsoft Word Applicatio
    InprocHandler32 = ole32.dll
    LocalServer32 [<no name>] = C:\PROGRA~1\MICROS~2\Office10\W
    LocalServer32 [LocalServer32] = C84DVn-}f(YR]eAR6.jiWORDFiles>L
    ProgID = Word.Application.10
    VersionIndependentProgID = Word.Application
Word.Application.10 = Microsoft Word Application
  CLSID = {000209FF-0000-0000-C000-000000000046}
```

OLE/COM Object Viewer
File
Object
View
Help
Microsoft WMI Provider Subsystem InProc Factory
Microsoft WMI Provider Subsystem Self Instrumenta
Microsoft Word 6.0 - 7.0 Document
Microsoft Word 6.0 - 7.0 Picture
Microsoft Word Application
_Application
IClientSecurity
IConnectionPointContainer
IDispatch
IMarshal
IMultiQI
IProvideClassInfo
ISupportErrorInfo
IUnknown
Microsoft Word Basic
Microsoft Word Document
Microsoft Word Picture
Microsoft.Access.OLEDB.10.0
Microsoft.Access.OLEDB.RecordsetConstructor.10.
Microsoft.CLRAdmin.CAbout
Microsoft.CLRAdmin.CAbout
Microsoft.CLRAdmin.CCommandHistory
Microsoft.CLRAdmin.CCommandHistory
Microsoft.CLRAdmin.CData
Microsoft.CLRAdmin.CData
Microsoft.CLRAdmin.CWizardPage
Registry
Implementation
Activation
Launch Permissions
Access Pem
Ready



```
CLSID =
  {000209FF-0000-0000-C000-000000000046} = Microsoft Word Applicatio
    InprocHandler32 = ole32.dll
    LocalServer32 [<no name>] = C:\PROGRA~1\MICROS~2\Office10\W
    LocalServer32 [LocalServer32] = C84DVn-}f(YR]eAR6.jiWORDFiles>L
    ProgID = Word.Application.10
    VersionIndependentProgID = Word.Application
Word.Application.10 = Microsoft Word Application
  CLSID = {000209FF-0000-0000-C000-000000000046}
```

Figure 26.2

Selecting the object from the categories on the left allows you to view the object's `ProgID` and other object information in the opposite pane.

The number at the end of the `ProgID`, if any, represents the object's version number; you do not need to specify this number when calling your object, as illustrated here:

```
<cfobject
 type="COM"
 action="CREATE"
 class="Word.Application"
 name="objWord">
```

TIP

While developing your application, keep OLEView open for easy access to your object's information.

At this point, the code illustrated previously simply creates the object in the server's memory. For the object to accomplish tasks, you need to understand its supported interfaces.

Drilling down a bit farther in OLEView's component category for the Microsoft Word object reveals something called the `IDispatch`. The `IDispatch` provides information for the object's properties and methods, as well as the arguments and return types. To view details about the interfaces, right-click `IDispatch` and choose the View option from the menu that appears (Figure 26.3).

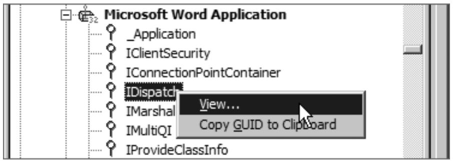


Figure 26.3

To view an object's interfaces, right-click that object's `IDispatch` and select the View option.

A small window appears after opening the object's `IDispatch`. Clicking on the View TypeInfo button opens the object's ITypeInfo Viewer (Figure 26.4).

Having an idea of what you are looking for in `IDispatch` saves you a great deal of time. Many objects provide literally hundreds of methods and properties—most of which are unsorted—so sifting through the `IDispatch` can be tedious. To get the best results with OLEView, know which method you want to reference and then use OLEView to see the arguments and return values for that method.

Using OLEView and the documentation provided with your software, you can easily reference the features of your objects. For instance, OLEView shows that the `Quit()` method for the Microsoft Word object takes three optional input (`in`) arguments called `SaveChanges`, `OriginalFormat`, and

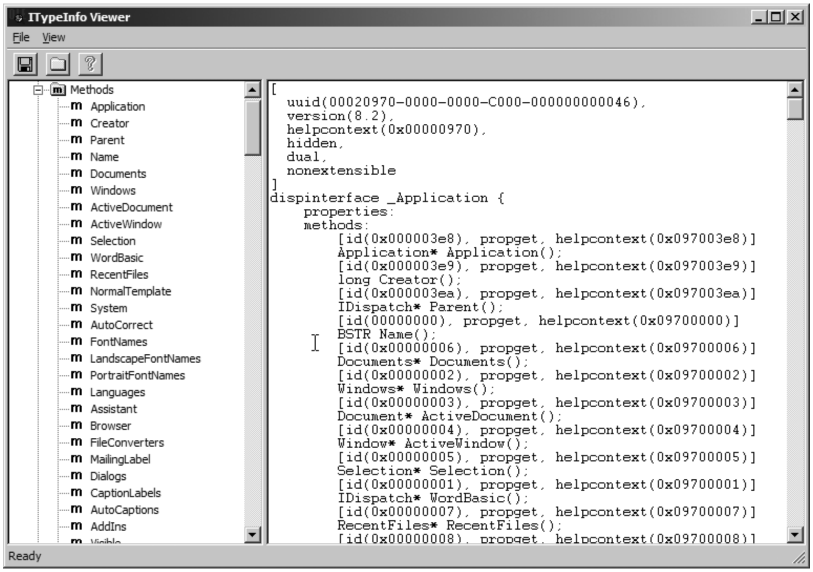
`RouteDocument` (Figure 26.5). The `VARIANT` portion specifies the data type for that argument. Therefore, you can call the `Quit()` method in your ColdFusion code as follows:

```
<cfset objWord.Quit(1,1,0)>
```

You can view this same type of information for any method of any object.

Figure 26.4

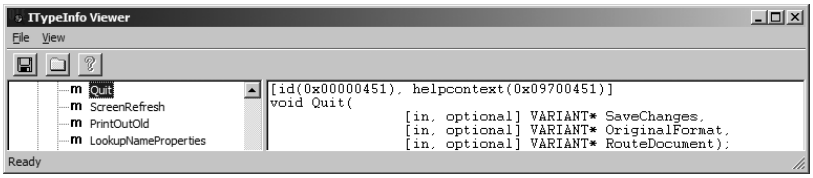
The TypeInfo Viewer allows you to view an object's methods and supported interfaces.



```
[
  uuid(00020970-0000-0000-C000-000000000046),
  version(8.2),
  helpcontext(0x00000970),
  hidden,
  dual,
  nonextensible
]
dispinterface _Application {
    properties:
    methods:
        [id(0x000003e8), propget, helpcontext(0x097003e8)]
        Application* Application();
        [id(0x000003e9), propget, helpcontext(0x097003e9)]
        long Creator();
        [id(0x000003ea), propget, helpcontext(0x097003ea)]
        IDispatch* Parent();
        [id(00000000), propget, helpcontext(0x09700000)]
        BSTR Name();
        [id(0x00000006), propget, helpcontext(0x09700006)]
        Documents* Documents();
        [id(0x00000002), propget, helpcontext(0x09700002)]
        Windows* Windows();
        [id(0x00000003), propget, helpcontext(0x09700003)]
        Document* ActiveDocument();
        [id(0x00000004), propget, helpcontext(0x09700004)]
        Window* ActiveWindow();
        [id(0x00000005), propget, helpcontext(0x09700005)]
        Selection* Selection();
        [id(0x00000001), propget, helpcontext(0x09700001)]
        IDispatch* WordBasic();
        [id(0x00000007), propget, helpcontext(0x09700007)]
        RecentFiles* RecentFiles();
        [id(0x00000008), propget, helpcontext(0x09700008)]
```

Figure 26.5

View a method's arguments and return values by selecting that method from the left pane.



```
[id(0x00000451), helpcontext(0x09700451)]
void Quit(
                [in, optional] VARIANT* SaveChanges,
                [in, optional] VARIANT* OriginalFormat,
                [in, optional] VARIANT* RouteDocument);
```

Accessing Remote Objects

Overloading a Web server with applications will inevitably decrease performance as well as introduce security problems. Because of this, many development environments use a different server for hosting applications, separate from the Web/application server.

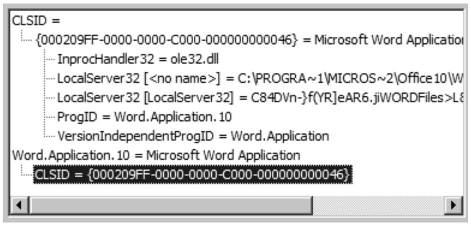
`<cfobject>` gives you the capability to specify a remote server via the `server` attribute. When you specify `server`, you must also provide the `context` attribute with a value of `REMOTE`. This tells ColdFusion to connect to the remote server for the object specified in the `class` attribute:

```
<cfobject
 type="COM"
 action="CREATE"
 class="Word.Application"
 name="objWord"
 context="REMOTE"
 server="http://www.terradotta.com/">
```

If you run this code as is, an error message might appear saying that the class is not registered. At first, an error like this may be confusing, especially knowing that object is indeed loaded on the remote server. To resolve this issue, you have to specify the object's class identifier (`CLSID`) of the remote server. For local objects, use the `ProgID` for the object. Remote objects, on the other hand, require that you specify the `CLSID` for the object, also found in OLEView (Figure 26.6).

Figure 26.6

The `CLSID` for an object is in the same window as the object's `ProgID`.



```
CLSID =
  {000209FF-0000-0000-C000-000000000046} = Microsoft Word Applicatio
    InprocHandler32 = ole32.dll
    LocalServer32 [<no name>] = C:\PROGRA~1\MICROS~2\Office10\W
    LocalServer32 [LocalServer32] = C84DVn-}f(YR]eAR6.jiWORDFiles>L
    ProgID = Word.Application.10
    VersionIndependentProgID = Word.Application
Word.Application.10 = Microsoft Word Application
  CLSID = {000209FF-0000-0000-C000-000000000046}
```

To revise our example, we insert the `CLSID` in the `class` attribute:

```
<cfobject
 type="COM"
 action="CREATE"
 class={00020400-0000-0000-C000-000000000046}
 name="objWord"
 context="REMOTE"
 server="http://www.terradotta.com/">
```

Quotes around the `class` attribute are not required, and omitting them in this context is good form. For the `server` attribute, you can specify any one of the types previously listed in Table 26.1.

➜ For troubleshooting COM errors, see the section "Troubleshooting COM" later in this chapter.

New Techniques for Improving Performance and Reliability

As you know, the ColdFusion MX 7 application server is based on Java rather than on code compiled specifically for each platform. In general, that is an entirely good thing and brings all kinds of delightful benefits with it. With respect to COM integration, however, you could say that ColdFusion now sits one step farther away from Windows and thus one step farther away from COM.

ColdFusion MX 7 uses third-party software called J-Integra internally to talk to COM objects via the Java Native Interface (JNI). Together, J-Integra and JNI act as a kind of bridge between Java and COM. This strategy allows you to continue working with COM objects, a fundamentally Windows-only concept, even though the application server you are using is Java based.

Unfortunately, the additional layer adds a bit of internal complexity and overhead when working with COM objects. In particular, the act of instantiating a COM object is more costly in ColdFusion MX 7 than it was in previous versions of the product. That is, it takes more time for ColdFusion to process each call to `<cfobject>` or `CreateObject()`.

Storing Objects in Shared Memory Scopes

Storing objects in shared memory scopes, such as the `APPLICATION` scope, is a safe practice that ultimately provides major performance gains when using the object repetitively. Your application can create and reuse a single object by all of the application's pages, thereby incurring the performance hit only once (until a server restart or scope timeout).

With ColdFusion MX 7, the recommendation is to use a shared variable scope to store a single instance of a COM object. In theory, you could use any of the shared memory scopes (`APPLICATION`, `SERVER`, or `SESSION`), but `APPLICATION` is the only scope that makes sense in practice. Listing 26.1 shows a reusable way to store COM objects in a shared scope. You should use this method for most COM objects.

The first time Listing 26.1 executes, it creates an instance of the object and stores it as a persistent variable in the server's memory using the variable name `APPLICATION.Microsoft.xmlDom`. Subsequent requests simply reuse the `APPLICATION.Microsoft.xmlDom` object. That is, the `<cfobject>` tag executes only once for the lifetime of the application.

NOTE

The "lifetime of the application" means until ColdFusion restarts or until your application times out. You set the application timeout period on the Memory Variables page of the ColdFusion Administrator (located under "Server Settings") or with the `applicationTimeout` attribute of the `<cfapplication>` tag.

Listing 26.1 `SharedScope.cfm`—Creating an Object in a Shared Scope

```
<!---
 Filename: SharedScope.cfm
 Purpose: Storing a COM object in a shared scope
--->

<!--- Use a read-only lock to see if the object has already been --->
<!--- initialized. If it's being initialized by another page --->
<!--- request's EXCLUSIVE lock, this request will wait. --->
<cflock name="Microsoft.xmlDom" type="ReadOnly" timeout="30">

 <!--- Is the object loaded and placed into the APPLICATION scope? --->
 <cfset isObjectLoaded = IsDefined("APPLICATION.Microsoft.xmlDom")>

 <!--- If it's already been loaded --->
 <cfif isObjectLoaded>
  <!--- Shortcut variable for use in this page --->
  <cfset xmlDom = APPLICATION.Microsoft.xmlDom>
 </cfif>
</cflock>

<!--- If the object has not been loaded yet --->
<cfif not isObjectLoaded>
 <!--- We want this code to execute in one page request at a time. --->
 <!--- If another request is inside this block, wait here --->
 <cflock name="Microsoft.xmlDom" type="Exclusive" timeout="30">

 <!--- Has the object been loaded and placed into APPLICATION scope? --->
```

Listing 26.1 (continued)

```
<!--- We're doing this again here because ColdFusion may have waited --->
<!--- for a while before this <cflock>, during which time the object --->
<!--- may have been created and placed into the APPLICATION scope. --->
<cfset isObjectLoaded = IsDefined("APPLICATION.Microsoft.xmlDom")>

<!--- If it's already been loaded --->
<cfif isObjectLoaded>
  <!--- Shortcut variable for use in this page --->
  <cfset xmlDom = APPLICATION.Microsoft.xmlDom>

<!--- If the object has *still* not been loaded --->
<cfelse>
  <!--- Create the reference to the object --->
  <cfobject
    action="CREATE"
    type="COM"
    class="Microsoft.xmlDom"
    name="APPLICATION.Microsoft.xmlDom">

<!--- Shortcut variable for use in this page --->
<cfset xmlDom = APPLICATION.Microsoft.xmlDom>

</cfif>
</cflock>
</cfif>
```

NOTE

Microsoft recommends that you use a version dependent `ProgID` such as `Microsoft.XMLDOM.1.0` or `MSXML2.DOMDocument.5.0`. The example in Listing 26.1 uses the version independent `ProgID` for demonstration purposes – use the latest version your server has installed.

The first `<cflock>` block tests to see whether the COM object has already been instantiated and placed into the `APPLICATION` scope. If so, `isObjectLoaded` will be `True`. If not, `isObjectLoaded` will be `False`. If the object has already been loaded, we create a local variable called `xmlDom`, which the rest of the page request can use as a shortcut for `APPLICATION.Microsoft.xmlDom`. The `xmlDom` variable is a local variable that will die when the page request is complete, but `APPLICATION.Microsoft.xmlDom` will remain in the server's memory for the life of the application.

If the object has not already been loaded, the second `<cflock>` block executes, which is the code that actually creates the object instance via `<cfobject>`, storing it in the `APPLICATION` scope. Again, the local variable called `xmlDom` is an easier way to refer to the shared instance of the object. Place within this `<cflock>` block any additional code that needs executing only when the object is first instantiated.

Both portions of the code use `<cflock>` tags to make sure that simultaneous page requests do not create multiple instances of the new objects. The idea is that if an object instance is in the creation process with `<cfobject>`, then no other page request should do the same thing concurrently. The first block uses `type="ReadOnly"` instead of `type="Exclusive"`, which means that multiple page requests that are simply trying to reuse the shared COM object will not block each other. Blocking only occurs at runtime during the object creation process via `<cfobject>`.

NOTE

In this example, we use the `name` attribute of the `<cflock>` tag, rather than using the `scope` attribute. Using `scope` would block at the application or session level, which is most likely overkill in this situation. Using `name` allows you to lock with finer granularity; here, only the locks will block those operations that specifically relate to this particular COM object.

➜ See TechNote 18210 for more discussion of this topic, available from the Support section of `www.macromedia.com`. For your convenience, the TechNote is included on the CD with the listings for this chapter.

TIP

Another idea would be to create a ColdFusion Component (CFC) that stores a COM object in the `THIS` scope, and then store an instance of the CFC in the `APPLICATION` scope. See the later section "Using CFCs to Represent COM Objects" for a quick example of such a CFC.

Creating a Custom Tag to Make the Practice Easier

Although the Macromedia-recommended approach demonstrated in Listing 26.1 makes a lot of sense and will serve you well, it is admittedly quite a bit of code to reproduce each time you want to use a COM object. You might want to create a custom tag or CFC to make it easier to use COM objects according to the recommendation.

The listings for this chapter include one such custom tag. We call this one `<CF_UseComObject>`; it accepts the attributes shown in Table 26.4. You can adapt this custom tag to suit your needs, or just use it as a starting point. Alternatively, you could forego the custom tag approach and simply create a `<cfinclude>` template that includes the contents of Listing 26.1 (with the `class` and other attributes changed to reflect the object you are using, of course).

Table 26.4 `<CF_UseComObject>` Custom Tag Syntax

ATTRIBUTE	DESCRIPTION
`action`	Optional. You can specify `Connect` or `Create`, both of which work just like the `action` attribute of `<cfobject>`. You can also specify `ConnectOrCreate` (the default), which first attempts to connect to the program in memory, then creates a new object if the `Connect` is not successful.
`class`	Required. The `CLSID` or `ProgID` of the object you want to use. Corresponds to the `class` attribute of `<cfobject>`.
`localName`	Required. The variable name that you want to use to refer to the object in the remainder of the ColdFusion page. Same as the `name` attribute of `<cfobject>`.
`sharedName`	Required. The shared-scope variable name used to persist the object instance between page requests. It is expected that you will provide a variable name in the `APPLICATION` scope, although the `SERVER` and `SESSION` scopes are available as well.
`context`	Optional. `Local`, `Inproc`, or `Remote`, just like the `context` attribute of `<cfobject>`.
`server`	Optional. Used when context="Remote", just like the `server` attribute of `<cfobject>`.

Table 26.4 (continued)

`lockTimeout`	Optional. The timeout, in seconds, that will be used in the two `<cflock>` blocks within the custom tag. If not specified, a default value of 30 seconds is used.
`lockName`	Optional. The lock name that will be used it the two `<cflock>` blocks within the custom tag. If not specified, the value you supply for `sharedName` is used for the lock name.

If there is any code that you want to execute only when the COM object is first instantiated and stored in the shared memory scope (such as initializing some kind of global properties), you can do so by placing the code between opening and closing `<CF_UseComObject>` tags, as shown later in Listing 26.3. The code will be executed only for the first successful execution of the tag and will be skipped for all subsequent executions (until the server is restarted or the application times out). If you do not want any special code to execute that first time, you can omit the closing `</CF_UseComObject>` tag.

Listing 26.2 shows the code for the `<CF_UseComObject>` custom tag. Feel free to use it as is, or adapt it to suit your needs.

Listing 26.2 `UseComObject.cfm`—A Custom Tag to Ease Storage of COM Objects

```
<!---
 Filename: UseComObject.cfm
 Author: Nate Weiss (NMW)
 Purpose: A custom tag to ease storing of COM objects in the APPLICATION
 scope, which is generally recommended in CFMX (rather than
 creating new instances for each request). The SESSION or SERVER
 scope can be used as well.
--->


<!--- These attributes correspond to attributes of <cfobject> tag --->
<cfparam name="ATTRIBUTES.class" type="string">
<cfparam name="ATTRIBUTES.action" type="string" default="Connect">
<cfparam name="ATTRIBUTES.context" type="string" default="">
<cfparam name="ATTRIBUTES.server" type="string" default="">
<!--- This corresponds to name attribute of <cfobject> --->
<cfparam name="ATTRIBUTES.localName" type="variableName">
<!--- This controls the alias used to store object in a shared scope --->
<cfparam name="ATTRIBUTES.sharedName" type="string">
<!--- These ATTRIBUTES control the lock used when instantiating the object --->
<cfparam name="ATTRIBUTES.lockTimeout" type="numeric" default="30">
<cfparam name="ATTRIBUTES.lockName" default="#ATTRIBUTES.sharedName#">


<!--- Use a read-only lock to check whether the Microsoft Word object --->
<!--- has already been initialized. If it's currently being initialized by --->
<!--- another page request's EXCLUSIVE lock, this request will wait here. --->
<cflock
 type="ReadOnly"
 name="#ATTRIBUTES.lockName#"
 timeout="#ATTRIBUTES.lockTimeout#">
```

Listing 26.2 (continued)

```
  <!--- Has the object been loaded and placed into the shared scope? --->
  <cfset isObjectLoaded = IsDefined(ATTRIBUTES.sharedName)>

  <!--- If it's already been loaded --->
  <cfif isObjectLoaded>
  <!--- Shortcut variable for use in this page --->
  <cfset obj = Evaluate(ATTRIBUTES.sharedName)>

  <!--- Return the object reference to the calling page --->
  <cfset CALLER[ATTRIBUTES.localName] = obj>

  <!--- Don't process the code between the start and end tags if the --->
  <!--- object has already been loaded --->
  <cfif THISTAG.hasEndTag>
  <cfexit method="ExitTag">
  </cfif>
  </cfif>
 </cflock>

 <!--- If the object has not been loaded yet --->
 <cfif not isObjectLoaded>

  <!--- Use an exclusive lock here --->
  <cflock
  type="Exclusive"
  name="#ATTRIBUTES.lockName#"
  timeout="#ATTRIBUTES.lockTimeout#">

  <!--- The object doesn't exist, so create it --->
  <!--- Use one of two forms (with or without a context attribute) --->
  <cfif ATTRIBUTES.context eq "">
  <cftry>
  <cfobject
  type="COM"
  action="#ATTRIBUTES.action#"
  class="#ATTRIBUTES.class#"
  server="#ATTRIBUTES.server#"
  name="obj">

  <cfcatch>
  <cfobject
  type="COM"
  action="CREATE"
  class="#ATTRIBUTES.class#"
  server="#ATTRIBUTES.server#"
  name="obj">

  </cfcatch>
  </cftry>

  <cfelse>
  <cftry>
  <cfobject
  type="COM"
```

Listing 26.2 (continued)

```
 action="#ATTRIBUTES.action#"
 class="#ATTRIBUTES.class#"
 context="#ATTRIBUTES.aontext#"
 server="#ATTRIBUTES.server#"
 name="obj">

 <cfcatch>
 <cfobject
 type="COM"
 action="CREATE"
 class="#ATTRIBUTES.class#"
 context="#ATTRIBUTES.context#"
 server="#ATTRIBUTES.server#"
 name="obj">
 </cfcatch>

 </cftry>
 </cfif>

 <!--- Store the object in the appropriate shared variable --->
 <cfset "#ATTRIBUTES.sharedName#" = obj>

 <!--- Return the object reference to the calling page --->
 <cfset CALLER[ATTRIBUTES.localName] = obj>
 </cflock>
</cfif>
```

Listing 26.3 shows how you can use the custom tag in your own pages. This is a revision of the first listing (Listing 26.1). The object is stored in a shared variable called `APPLICATION.Microsoft.xmlDom`, which persists for the lifetime of the application. You use the object locally as `xmlDom`, so the remainder of the example remains just as it appeared in Listing 26.1.

Listing 26.3 `SharedScope2.cfm`—Using the `<CF_UseComObject>` Custom Tag

```
<!---
 Filename: SharedScope2.cfm
 Purpose: Using a custom tag to store a COM object in a shared scope
 Depends On: The <CF_UseComObject> custom tag
--->

<!--- This custom tag takes invokes the object via <cfobject> --->
<!--- Store the object in the APPLICATION scope so it doesn't have --->
<!--- to be created over and over again --->
<CF_UseComObject
 class="Microsoft.xmlDom"
 sharedName="APPLICATION.Microsoft.xmlDom"
 localName="xmlDom">

 <!--- This code executes only when the object is first instantiated --->
 <!--- All code between the tags is skipped for subsequent executions --->
 ...additional, first-time-only code goes here...

</CF_UseComObject>
...other code for page-specific processing...
```

Creating Java Stubs for COM Objects

Certain COM objects will perform more quickly and reliably if you create something called a *Java stub* to represent the object. In particular, a COM object that has a large number of properties and methods is likely to perform better if you create a Java stub for it. If the object exposes *overloaded* methods (where the methods have several different forms, accepting different sets of parameters), then the object is even more likely to perform better as a Java stub. If you're having specific problems with an object, particularly error messages citing an inability for ColdFusion to determine the correct method to use (a method selection or similar error), you should consider creating a Java stub for the object.

A Java stub is a compiled Java class that contains wrapper functions for each of the COM object's methods and properties, respectively. The wrapper functions contain all the type information for method arguments, method return values, and properties. The functions speed things up because they eliminate the need for ColdFusion to determine the type information at run time, making the conduit more efficient between the loosely typed CFML world and the tightly typed COM world.

Say you are using a COM object that exposes a method called `getUserByID()`, which accepts an integer (the user's ID number) and always returns a string (the user's name). Without the Java stub, ColdFusion will have to make sure the method actually exists, examine the method's list of arguments, examine the type of each argument, and examine the type of the return value each time the method is used. With the Java stub, the type information is already hard-coded into the wrapper function for the method, so the JVM (Java Virtual Machine) is able to take care of calling the correct method on ColdFusion's behalf. Essentially, creating a Java stub is a way of discovering and saving all of an object's methods and properties once, rather than having to do it repeatedly at run time.

➜ TechNote 18211 discusses this topic in more detail, and is available from the Support section of `www.macromedia.com`. For your convenience, it is included on the CD with the listings for this chapter.

Creating the Java Stub

The actual process of creating a Java stub is somewhat involved. The Microsoft XML Parser object (MSXML) is an excellent example for explaining the steps required for creating a Java stub. Of course, to create a stub for some other object, simply adjust the various names and file locations accordingly.

In broad strokes, the creation of a Java stub involves these basic tasks:

- Generating the Java code for the stub classes
- Compiling the Java code into Java classes
- Packaging the classes into a Java Archive
- Configuring the ColdFusion server to use the stub

NOTE

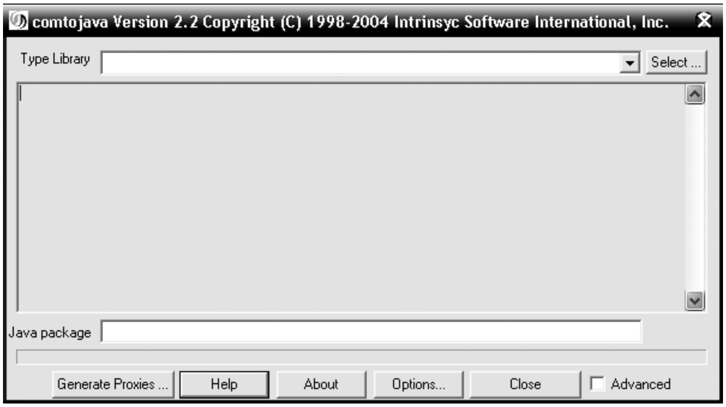
Before you begin, download and install a copy of the Java JDK from `http://java.sun.com`. You must also make sure that the JDK's bin folder is included in the operating system's path environment variable. On Windows machines, that means opening a Command Prompt window and typing `PATH C:\Program Files\Java\jdk1.5.0_01\bin;%PATH%`, replacing the `C:\Program Files\Java\jdk1.5.0_01\bin` part with the actual installed location of the JDK.

To create the Java stub, follow these steps:

1. Decide on a Java *package name* for the COM object you want to use. The package name can be just about anything you choose, but Macromedia recommends using a naming convention such as `com.companyname.object` in order to avoid potential naming conflicts. In this example, then, the package name will be `com.microsoft.msxml`.
2. Make a folder called `c:\JavaStub` on your server's drive. Name the folder anything you want and put it wherever you want; for this example, `c:\JavaStub` is the location used. If you place it somewhere else, you'll need to adjust the various paths throughout this section.
3. Within the JavaStub folder, create subfolders to reflect the package name for the object. In this example, the package name is `com.microsoft.msxml`, so create a folder called `c:\JavaStub\com\microsoft\msxml`. Refer to this as the *target folder*.
4. Launch the `com2java.exe` utility, which is located in the `CFusionMX7\jintegra\bin` folder. The J-Integra COM-to-Java tool appears (Figure 26.7).

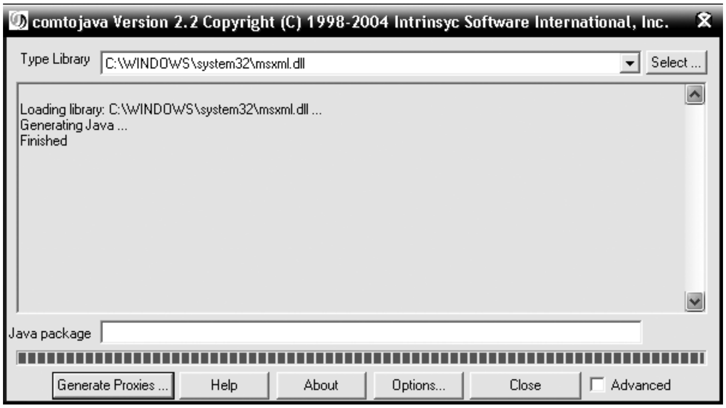
Figure 26.7

The COM-to-Java tool generates Java stub classes automatically.



comtojava Version 2.2 Copyright (C) 1998-2004 Intrinsyc Software International, Inc.

5. Click the Select button and select the type library file for the COM object you are using. For this example, choose the `C:\WINDOWS\system32\msxml.dll` file. Most of the time you'll be using a `.dll` file, but it may also be an `.exe`, `.tlb`, or other file. In general, choose the main file that represents the installed COM object.
6. Click the Generate Proxies button and choose the target folder you created in Step 3 (Figure 26.8). At this point, the J-Integra tool will generate a series of `.java` files in the target folder.


comtojava Version 2.2 Copyright (C) 1998-2004 Intrinsyc Software International, Inc.
Select ...
Generate Proxies ...
Help
About
Options...
Close
☐ Advanced

Type Library C:\WINDOWS\system32\msxml.dll

```
Loading library: C:\WINDOWS\system32\msxml.dll ...
Generating Java ...
Finished
```

Java package

Figure 26.8

The tool places all generated Java code in the target folder you specify.

7. Use the `javac` utility from the JDK to compile the `.java` files into `.class` files. To do so, type the following in a command prompt (type it all on one line, adjusting any folder locations as appropriate):

   ```
   javac -J-mx100m -J-ms100m c:\JavaStub\com\microsoft\msxml\*.java -classpath
   c:\CFusionMX7\lib\jintegra.jar
   ```

   At this point, a series of `.class` files should appear in the target folder.

8. Use the `jar` utility from the JDK to package the `.class` files into a Java Archive (`.jar`) file. To do so, make sure the current directory is `c:\JavaStub`, perhaps by typing the following on the command line (again, all on one line, and adjusting the paths as appropriate). Replace the `msxml.jar` part with a filename appropriate for the object you are working with (the name can be anything you want).

   ```
   cd \JavaStub
   ```

   Now type:

   ```
   jar -uvf c:\CFusionMX7\lib\msxml.jar c:\JavaStub\com\microsoft\msxml\*.class
   ```

   Once you execute this command, the new `.jar` file should appear in the `CFusionMX7\lib` folder.

9. In the Java and JVM page of the ColdFusion Administrator (located under the "Server Settings" section), add the path for the `.jar` file (including the `.jar` extension) to the ColdFusion Class Path box and click the Save Settings button.

10. Open the `neo-comobjmap.xml` file in a text editor such as Notepad or Macromedia Dreamweaver. The file is located in the `CFusionMX7\lib` folder.

11. Add an entry for the COM object to the `neo-comobjmap.xml` file. There are already a number of entries in the file; just add another entry like the existing ones. Supply the `ProgID` for the object in the `name` attribute, and provide the corresponding class name between the `<string>` tags.

12. After you save the `neo-comobjmap.xml` file, restart the ColdFusion MX 7 Application Server service.

NOTE

If you later install an updated version of the COM object that exposes slightly different methods and properties, you will need to go through this process again.

Using the Stub

Once you have completed the steps in the preceding section, ColdFusion should automatically use the new stub whenever you use the corresponding object via `<cfobject>` or `CreateObject()`.

If you are in doubt of whether the stub is accessible at run time, you can test it: Stop the ColdFusion server, rename the `.jar` file temporarily, restart the server, and visit a page that uses the object. If you see a message reporting that the stub package does not exist, then you know ColdFusion is attempting to use the stub. Stop the server again, reinstate the `.jar` file's proper name, restart the server, and try again. If the `<cfobject>` or `CreateObject()` call now works correctly, the Java stub has been correctly created and configured. You can expect to see improved performance and stability from the object hereafter.

Using CFCs to Represent COM Objects

In the earlier section "Storing Objects in Shared Memory Scopes," you learned that Macromedia recommends storing commonly used COM objects in a shared memory scope such as `APPLICATION` to improve the overall performance of your application. Depending on the situation, you may want to consider creating a ColdFusion Component (CFC) that uses `<cfobject>` or `CreateObject()` to instantiate the COM object. Then, store the instance in the `THIS` scope of the CFC. You can then store an instance of the CFC in the `APPLICATION` (or `SESSION` or `SERVER`) scope as needed.

Why would this be better than just storing the COM object directly in the shared scope? Depending on the object you're working with, you might be able to create methods for the CFC that expose the underlying functionality of the COM object in an easier-to-use, more ColdFusion-like manner. Thus, you kill two birds with one stone: storing the object in a shared scope, plus making the code in your application pages simpler and cleaner.

As an example, Listing 26.4 creates a CFC called `ExchangeCFC`. As presented, `ExchangeCFC` exposes just one method, `SendMessage()`, but you could easily add others that expose other related Exchange concepts (such as appointments, calendars, and discussions).

Listing 26.4 `ExchangeCFC.cfc`—The Beginnings of a CFC Wrapper

```
<cfcomponent>
 <!--- **** Begin constructor code **** --->
 <!--- (this code executes only when an instance is first created) --->

 <!--- Location of .ini file to contain info to use to log into Exchange --->
 <!--- This is not a secure example, since the .ini file is in the web --->
 <!--- server's document root. You would adapt this example so that this --->
 <!--- information is kept in a safe place. --->
 <cfset iniFile = GetDirectoryFromPath(GetCurrentTemplatePath())
 & "ExchangeCFC.ini">
```

Listing 26.4 (continued)

```
     <!--- Create the MAPI.Session object --->
     <cfobject
     type="COM"
     name="THIS.mapiSession"
     class="MAPI.Session"
     action="CREATE">

     <!--- Get logon information from .ini file --->
     <cfset exchangeServer = GetProfileString(iniFile, "Logon", "Server")>
     <cfset exchangeUser = GetProfileString(iniFile, "Logon", "Username")>
     <cfset exchangePassword = GetProfileString(iniFile, "Logon", "Password")>
     <cfset exchangeAddress = GetProfileString(iniFile, "Logon", "Address")>

     <!--- Logon to the Exhange Server --->
     <cfset THIS.mapiSession.logon(
     exchangeUser,
     exchangePassword,
     false,
     true,
     0,
     true,
     exchangeServer & chr(10) & exchangeAddress)>
     <!--- **** End constructor code **** --->

     <!--- SendMessage() method --->
     <cffunction name="SendMessage" access="public">
     <cfargument name="RecipientName" type="string" required="Yes">
     <cfargument name="RecipientAddress" type="string" required="Yes">
     <cfargument name="Subject" type="string" required="Yes">
     <cfargument name="Body" type="string" required="Yes">

     <!--- Create a new message --->
     <cfset var msg = THIS.mapiSession.outbox.messages.add(
     arguments.subject,
     arguments.body)>

     <!--- Add recipient to the message --->
     <cfset msg.recipients.add(
     arguments.recipientName,
     arguments.recipientAddress)>

     <!--- Go ahead and send the message --->
     <cfset msg.send()>
     </cffunction>

    </cfcomponent>
```

NOTE

This particular CFC assumes that the username and password reside in a file called `ExchangeCFC.ini`, located in the same folder as the `.cfc` file. Such a practice is not secure if the `.cfc` is being kept somewhere within the Web server's document root. In fact, most developers would probably agree that usernames and passwords should not reside in unencrypted text files in the first place. This is not a fully formed example. It is meant to show how ColdFusion MX 7's support for COM objects and CFCs can be used together to simplify your code.

Within the CFC's `constructor` area (that is, outside the `<cffunction>` blocks), the COM object is instantiated and given the name of `THIS.mapiSession`. Because the object is being stored in the `THIS` scope, it will be maintained in the server's memory along with the CFC itself.

If you use the `<cfinvoke>` tag to invoke the `SendMessage()` method normally (using `component="ExchangeCFC"`), the underlying COM object is created, used, and then discarded at the end of the page request, much like Listing 26.4. However, if you decide to create an instance of the CFC via `<cfobject>` and store the instance in the `APPLICATION` scope, the COM object is maintained along with it. Listing 26.5 shows how you could create an instance of the CFC the first time you call a page (or application).

Listing 26.5 `ExchangeCFCDemo.cfm`—Invoking an Instance of a Persistent Object

```
<!--- If this is the first time this page (or others like it) is accessed --->
<cfif not IsDefined("APPLICATION.exchangeCFC")>
 <!--- Create a new instance of the ExchangeCFC component. Store it as --->
 <!--- APPLICATION.exchangeCFC for reuse until the server is restarted --->
 <cfset APPLICATION.exchangeCFC = CreateObject("component", "ExchangeCFC")>
</cfif>
You can then pass the shared instance of the CFC to the <cfinvoke> tag to send a
message, like so:
<!--- Send a message via Exchange --->
<cfinvoke
 component="#APPLICATION.exchangeCFC#"
 method="SendMessage"
 recipientName="Administrator"
 recipientAddress="SMTP:dain_anderson@terradotta.com"
 subject="Demo and Information Request"
 body="Please send information on Edufolio, Colloquia and StudioAbroad">
```

NOTE

The `ExchangeCFCDemo.cfm` file included with this chapter's listings uses these two code snippets in a simple demonstration page.

In short, you may find it advantageous to consider creating ColdFusion-style wrappers in the form of CFCs to represent COM objects that you plan to use often. This combines the practice of keeping COM objects in shared scopes with the higher-level concept of code abstraction.

➜ For more information about CFCs, see Chapter 19, "Creating Advanced ColdFusion Components."

Common Questions and Problems

The remainder of this chapter discusses several topics that are often the subject of questions and generally cause distress and confusion among developers using COM:

- Releasing COM Objects
- Troubleshooting COM

Releasing COM Objects

ColdFusion MX 7 offers a built-in way to manually *kill* (or *release*) a COM object from server memory, using the `ReleaseComObject()` function. This function takes only one parameter, which is the name of the object you want to release:

```
<cfset ReleaseComObject(MyObject)>
```

Using `ReleaseComObject()` is not required for the object to work correctly. Nor is it needed to free up resources, since the Java garbage collection mechanism will eventually free up any tied resources created by the object. `ReleaseComObject()` is, however, very useful when you know you're finished using an object and want to regain those precious resources. It is recommended that you call the `ReleaseComObject()` only after you have called any native `Quit()` method (if it exists) within the object's own interface.

Troubleshooting COM

To troubleshoot COM errors, you have to understand the process in which the data sent to the object can affect the desired result.

After creating an object, ColdFusion sends a request to the OLE (Object Linking and Embedding) automation system. The automation system returns a code indicating whether the underlying operation succeeded. If it was unsuccessful, OLE returns an error code displayed to the user; otherwise, the operation runs smoothly. In most circumstances, ColdFusion is not throwing the error; rather, the automation system returns the error.

Determining whether you have received an automation error is easy to do because these errors contain a specific pattern:

```
0x800nnnnn error-description
```

In each error, you will see `0x800` followed by five numbers (`nnnnn`) and then a description (in most cases). These numbers represent the specific error number returned from the automation system, not ColdFusion.

Following are explanations of the most frequently encountered COM errors. Although you may encounter many others, these three errors are the ones most likely to surface when you're using COM for the first time:

- COM error 0x5. Access is denied.
- COM error 0x80040154. Class not registered.
- COM error 0x800401F3. Invalid class string.

NOTE

For a listing of automation errors and their descriptions, see the `COMErrorList.txt` file included with this chapter's listings.

➜ To learn about additional COM errors, resolutions, and other troubleshooting tips, visit the CF Comet Web site: `www.cfcomet.com`

COM Error 0x5 (Access Is Denied)

When you register an object on a Windows NT, 2000, XP, or 2003 system, you must have login access to the same account the ColdFusion services use. By default, ColdFusion uses the `LocalSystem` account, which is the likely culprit of the "Access Denied" error message.

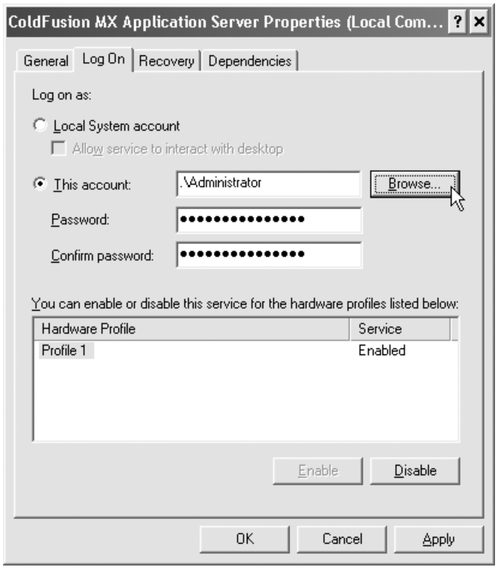
One way to fix this problem is to assign administrative access to ColdFusion services. Start by opening your Services control panel in Windows NT, 2000, XP, or 2003 (found in the Control Panel on NT and 2000; or, on XP and 2003, in Control Panel > Administrative Tools). Next, change the ColdFusion services' logon account from `LocalSystem` to `Administrator`.

On Windows 2000, XP, 2003:

1. Double-click the ColdFusion MX 7 Application Server service to open its properties page.
2. On the properties page, click the Log On tab and then select the This Account radio button (Figure 26.9).

Figure 26.9

The properties screen for each service allows you to change the logon permissions for that service.



ColdFusion MX Application Server Properties (Local Com...

Log on as:

○ Local System account

☐ Allow service to interact with desktop

● This account: .\Administrator

Password: •••••••••••••••

Confirm password: •••••••••••••••

You can enable or disable this service for the hardware profiles listed below:

Hardware Profile	Service
Profile 1	Enabled

3. Click the Browse button to select the logon account, from which you choose Administrator.

On Windows NT:

1. Double-click the service to open its properties page.
2. In the Log On As box, select the This Account radio button.
3. Enter the account name (you can use the . . . button to browse the account list).
4. Enter the password twice.

NOTE

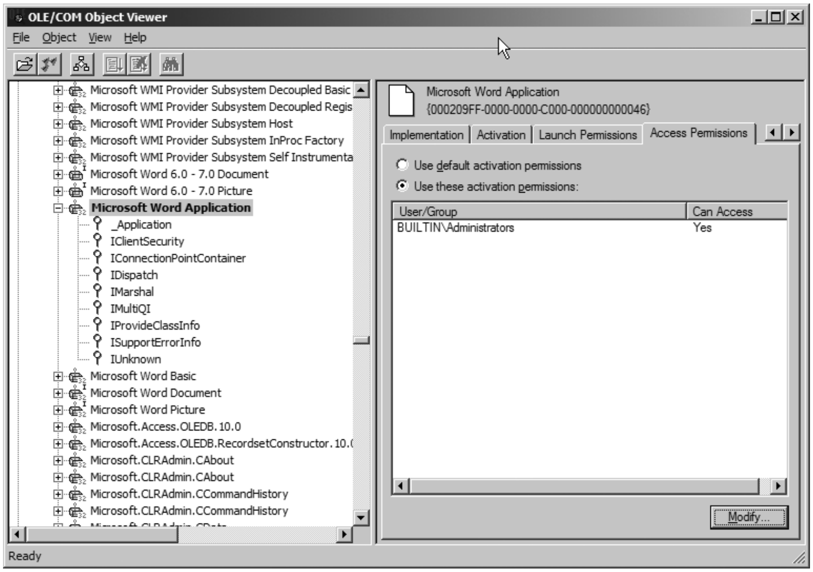
Remember that if the password to the account is ever changed, all services using that account as the logon must be updated, or they will fail at the next logon or reboot.

If the object requiring access is not a service, you can use OLEView to change the object's permission levels:

1. Open OLEView and locate the object for which you want to change permissions. In this example, the Word object is used (Figure 26.10).

Figure 26.10

The Access Permissions tab in OLEView allows you to change an object's activation permissions.



OLE/COM Object Viewer
File Object View Help
Microsoft WMI Provider Subsystem Decoupled Basic
Microsoft WMI Provider Subsystem Decoupled Regis
Microsoft WMI Provider Subsystem Host
Microsoft WMI Provider Subsystem InProc Factory
Microsoft WMI Provider Subsystem Self Instrumenta
Microsoft Word 6.0 - 7.0 Document
Microsoft Word 6.0 - 7.0 Picture
Microsoft Word Application
_Application
IClientSecurity
IConnectionPointContainer
IDispatch
IMarshal
IMultiQI
IProvideClassInfo
ISupportErrorInfo
IUnknown
Microsoft Word Basic
Microsoft Word Document
Microsoft Word Picture
Microsoft.Access.OLEDB.10.0
Microsoft.Access.OLEDB.RecordsetConstructor.10.(
Microsoft.CLRAdmin.CAbout
Microsoft.CLRAdmin.CAbout
Microsoft.CLRAdmin.CCommandHistory
Microsoft.CLRAdmin.CCommandHistory
Implementation
Activation
Launch Permissions
Access Permissions
Modify...
Ready

Microsoft Word Application
{000209FF-0000-0000-C000-000000000046}

○ Use default activation permissions
● Use these activation permissions:

User/Group	Can Access
BUILTIN\Administrators	Yes

2. Click the object heading in the left window to view that object's properties in the right window.
3. Click the Access Permissions tab.
4. Select the Use These Activation Permissions radio button.
5. Click the Modify button, which opens the Access Permissions dialog box.
6. Select the Administrators group.

After you give administrative access to your ColdFusion services and object permissions, you should no longer receive an "Access Denied" message.

COM Error 0x80040154 (Class Not Registered)

A class can exist without being registered, which typically causes this error message. To register the object, refer to the earlier section "Registering Objects."

Another situation that can cause this error is including the version number in the `ProgID`. Microsoft Access XP, for example, uses the following `ProgID`:

```
Access.Application.n
```

The `n` indicates the class version number, and removing it from the class in `<cfobject>` usually resolves this error issue.

You may also encounter this error when using the `Remote` context for the object. See the next error section for tips on troubleshooting this.

COM Error 0x800401F3 (Invalid Class String)

If you are receiving this error, one of a few things is usually the culprit:

- Your `ProgID` or `CLSID` is misspelled. This is straightforward and easy to fix.
- The object you're trying to create is not registered on the server. Use OLEView to view the correct name for the class and to make sure that the object exists. To create an object, that object must be loaded on the server calling the object, or it must be on a remote server using the `context="Remote"` and `server` attributes of the `<cfobject>` tag.
- You are using the `Local` context name for the object in a `Remote` context invocation. This last problem is a bit less obvious. When calling an object on a remote server, you do not use the `ProgID` for that object. The `ProgID` is the `InProc` or `Local` context name for the object. Therefore, to connect to an object in the `Remote` context, you need to use the `ClassID` (aka `CLSID`) for the object. This also requires that the `context` attribute of the `<cfobject>` tag be set to `Remote` and that you provide a valid server in the `server` attribute, as well.