COM Collections

William Pantoja

Writers
William Pantoja

HTML Coding
William Pantoja

Quality Assurance
William Pantoja
Mike Prestwood

Technical Advisors/Source Code
William Pantoja

Editing
Mike Prestwood

Additional Material
Some of the material in this guide is taken from the following sources and is copyrighted by their respective holders:

?? Borland Delphi 4 VCL source code
?? Borland Delphi 5 VCL source code
?? Borland Delphi 6 VCL source code
?? Microsoft Developers Network (MSDN) Platform SDK

Table of Contents

Introduction

The Component Object Model (COM) specification provides a tool to allow developers to create reusable components that are accessible in many different programming languages. COM objects may be developed in a variety of languages including Delphi and C++. This guide covers one portion of COM object development: the creation of COM collections.

It is beyond the scope of this guide to discuss the basics of COM object development. Several resources are available which describe the basics of developing COM objects in detail.

What is a COM collection?

A collection is a set of similar entities that can be manipulated as a group using a single interface. An excellent example of a collection is the fields collection of an ADO RecordSet. A COM collection is simply the implementation of a collection by a COM server that follows certain interface guidelines to allow for a standard way of manipulating the items within the collection.

What is a COM collection used for?

A COM collection is an excellent tool for the developer who wishes to allow a developer to use a standard way of manipulating a set of similar entities. Visual Basic[1] programmers have a simple mechanism for using COM collections: the **For...Each...Next** statement.

Whenever a **For...Each...Next** statement is used to iterate through a collection, Visual Basic uses standard properties and the **IEnumVARIANT** interface defined in the COM collection to enumerate through the entities contained within the collection.

Why would you want to implement a COM collection?

Although the are several methods to implementing a collection in COM, by following the specification set forth by Microsoft you make it easier for developers to manipulate your collection and allow them to utilize the **For...Each...Next** statement to enumerate the entities within your collection.

Consider the two following code fragments in Visual Basic which enumerate through the entities within a collection:

Code Fragment 1:

```
For I = 1 To objMyCollection.Count
   objMyEntity = objMyCollection.Item(I)
   'Use objMyEntity
Next
```

Code Fragment 2:

```
For Each objMyEntity In objMyCollection
   'Use objMyEntity
Next
```

There are several potential problems with the first code fragment. In order for the code to work, a single important assumption was made: the collection's first index is 1. Though the standard initial index of a COM collection should always be 1 relative and not 0 relative, it assumes that the developer of the collection followed this standard.

The first code fragment also assumes that the index of the collection is an ordinal value. It is possible (and sometimes beneficial) to create a collection whose unique index is a string rather then a number.

The second code fragment uses the **For...Each...Next** statement to enumerate through the entities contained within the COM collection. It makes no assumptions about, and indeed does not need to know, how an element is indexed. In addition, the developer saves a line of code.

[1] References to Visual Basic include Visual Basic, VBScript, and Visual Basic for Applications (VBA).

Building a COM Collection Interface

Our first task in building a COM collection is creating the collection interface in the type library. When implementing a collection, there are two properties and a method that are required to be implemented. Two additional methods are optional and you may also choose to add additional methods and properties that are specific to your own design requirements.

The _NewEnum Property

The first property that is required is the **_NewEnum** property. The **_NewEnum** property is read-only and returns an **IUnknown** interface to an object which implements the **IEnumVARIANT** interface and must have a dispatch ID of -4. The **_NewEnum** property should also have the **hidden** attribute to prevent it from showing up Visual Basic's IntelliSense™.

The Microsoft Interface Development Language (MIDL) declaration is as follows:

```
[propget, id(-4), hidden] HRESULT get__NewEnum([out, retval] IUnknown **pVal);
```

Visual Basic uses the **_NewEnum** property to enumerate the collection when a **For...Each...Next** is used.

The Count Property

The second property that is required is the **Count** property. The **Count** property is read-only returns the number of elements in the collection.

The MIDL declaration is as follows:

```
[propget, id(1)] HRESULT get_Count([out, retval] int **pVal);
```

The Item Method

The **Item** method is the only required method. The **Item** method takes one or more parameters that indicate an item in the collection to return. The value returned by the **Item** method is dependent on the type of items within the collection. If the collection is a collection of objects, an **IDispatch** interface should be returned. There should be a way to indicate a unique item in the collection. The dispatch ID should be 0 and have the **uidefault** attribute to allow this method to be the default method of the object.

The template for the MIDL declaration is as follows:

```
[id(0), uidefault] HRESULT Item([in] datatype varname, [out, retval] returntype *pVal);
```

Where *datatype varname* is one or more parameters used to indicate an item in the collection. *returntype* is an OLE-safe data type that is the item returned by the **Item** method.

For example, if you created a collection of objects that contain name-value pairs, you could declare the Item method as follows:

```
[id(0), uidefault] HRESULT Item([in] VARIANT Index, [out, retval] IDispatch **pVal);
```

Where the *Index* parameter takes a variant that could be either the ordinal index of an item in the collection (much like an array) or a string that refers to one of the name-value pairs. If duplicate names are allowed in this collection, you can reference a single item in the collection by specifying the ordinal index of an item.

The Add Method

The **Add** method is an optional method that allows the user to add items to the collection. The parameters (if any) are up to the developer depending on the design requirements of the collection. However, if a value is returned it should be the same data type returned by the **Item** method.

The Remove Method

The **remove** method is an optional method that allows the user to remove items from the collection. The parameters (if any) are up to the developer depending on the design requirements of the collection. However, if a value is returned it should be the same data type returned by the **Item** method.

The IEnumVARIANT Interface

The **IEnumVARIANT** interface provides a method for enumerating a collection of variants, including heterogeneous collections of objects and intrinsic types. Callers of this interface do not need to know the specific type (or types) of the items in the collection.

The MIDL declaration of the **IEnumVARIANT** interface is as follows:

```
[
  odl,
  uuid(00020404-0000-0000-C000-000000000046),
  hidden
]
interface IEnumVARIANT : IUnknown {
    HRESULT Next([in] unsigned long celt, [in] VARIANT* rgvar, [out] unsigned long* pceltFetched);
    HRESULT Skip([in] unsigned long celt);
    HRESULT Reset();
    HRESULT Clone([out] IEnumVARIANT** ppenum);
};
```

The Next Method

The **Next** method attempts to get the next *celt* items in the collection returning them through the array of variants pointed to by *rgvar*. It is the responsibility of the caller to allocate enough memory to hold *celt* variants in the array. The number of items returned in *rgvar* is returned in *pceltFetched* if it is not a NULL pointer. If the number of elements returned is less then *celt*, the **Next** method must return S_FALSE.

The Skip Method

The **Skip** method attempts to skip over the next *celt* items in the collection. If the end of the collection is reached before *celt* items have been skipped, then the **Skip** method must return S_FALSE.

The Reset Method

The **Reset** method resets the enumeration sequence back to the beginning. If possible, the items return by subsequent calls to **Next** should return the same items as before. However, sometimes this is impractical for collections whose items are dynamic (such as a collection of files is a folder).

The Clone Method

The **Clone** method creates a copy of the current state of the enumeration. Using this method, a particular point in the enumeration sequence can be recorded, and then returned to at a later time. The returned enumerator is of the same actual interface as the one that is being cloned.

There is no guarantee that exactly the same set of variants will be enumerated the second time as was enumerated the first. Although an exact duplicate is desirable, the outcome depends on the collection being enumerated. You may find that it is impractical for some collections to maintain this condition (for example, an enumeration of the files in a directory).

Implementing a COM Collection

Create a New Project

The first thing we must do is create a new project. We are going to create an automation server that is compatible with ASP.

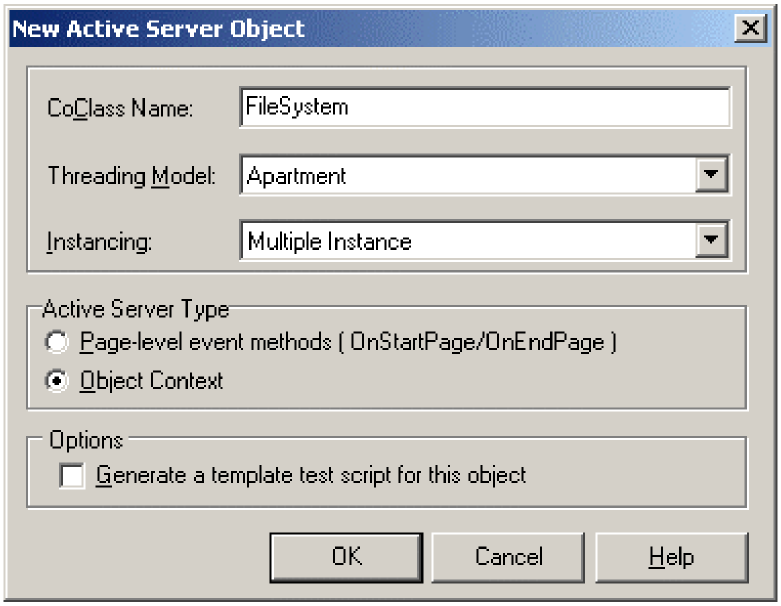
1. Select **Tools|Environment Options...** from the main menu.
2. Select the **Type Library** tab.
3. Under **Language**, select **Pascal**.
4. Click the **OK** button.
5. Select **File|New|Other...** from the main menu.
6. Select the **ActiveX** tab.
7. Double-click the **ActiveX Library** icon.
8. Select **View|Type Library** from the main menu.
9. Select the ❖ library in the tree.
10. In the **Name** field enter "COMExample".
11. Save your project. When prompted for a project filename, enter "COMExample.dpr".

Create the Objects

Using the following table, create all the objects listed following the steps below:

CoClass	Interface	Filename	CanCreate
FileSystem	IFileSystem	objFileSystem.pas	Yes
Folder	IFolder	objFolder.pas	No
Files	IFiles	objFiles.pas	No
Folders	IFolders	objFolders.pas	No

1. Select **File|New|Other...** from the main menu.
2. Select the **ActiveX** tab.
3. Double-click the **Automation Object** icon.
4. Type the coclass name (refer to the table above) in the **CoClass name** box.



New Active Server Object

CoClass Name: FileSystem

Threading Model: Apartment

Instancing: Multiple Instance

Active Server Type

○ Page-level event methods (OnStartPage/OnEndPage)

● Object Context

Options

☐ Generate a template test script for this object

5. Click the **OK** button.
6. Select **View|Type Library** from the main menu.

7. Select the coclass.
8. Select the **Flags** tab.
9. Set the **Can Create** box according to the table above.
10. Select the interface.
11. Check the **Hidden** box.
12. Save your project. When prompted for a filename, use the filename from the table above.
13. Repeat for all objects.

We have now created all the objects we will need for this example.

Implement the FileSystem Object

The FileSystem object will be our root object. Users of your automation server will create and instance of this object and, using it's properties and methods, retrieve folder and file information. For simplicity's sake, we will not allow the user to modify folders or files.

Add Methods

The FileSystem object will have three methods.

Method	Description
FolderExists	Returns true if the specified folder exists.
FileExists	Returns true if the specified file exists.
GetFolder	Returns a Folder object represented the specified folder or Null if the folder does not exist.

1. Select **View|Type Library** from the main menu.
2. Select the "IFileSystem" interface.

3. Click the **New Method** button in the toolbar.
4. Select the **Attributes** tab.
5. In the Name field type "FileExists".
6. Select the **Parameters** tab.
7. Change **Return Type** to "WordBool".
8. Click the **Add** button to add a new parameter.
9. In the **Name** column for the parameter type "Name".
10. In the **Type** column select "WideString".

11. Click the **New Method** button in the toolbar.
12. Select the **Attributes** tab.
13. In the **Name** field type "FolderExists".
14. Select the **Parameters** tab.
15. Change **Return Type** to "WordBool".
16. Click the **Add** button to add a new parameter.
17. In the **Name** column for the parameter type "Name".
18. In the **Type** column select "WideString".

19. Click the **New Method** button in the toolbar.
20. Select the **Attributes** tab.
21. In the **Name** field type "GetFolder".
22. Select the **Parameters** tab.
23. Change **Return Type** to "IDispatch".
24. Click the **Add** button to add a new parameter.
25. In the **Name** column for the parameter type "Name".
26. In the **Type** column select "WideString".

27. Save your project.

Implement the Methods and Properties

The first step is to modify the class declaration of TFiles.

```
   :
   :
   :
implementation

uses ComServ, FileCtrl, SysUtils, objFolder;

{-----------------------------------------------------------------------------}

function TFileSystem.FileExists(const Name: WideString): WordBool;

begin
   Result := FileExists(Name);
end;

{-----------------------------------------------------------------------------}

function TFileSystem.FolderExists(const Name: WideString): WordBool;

begin
   Result := DirectoryExists(Name);
end;

{-----------------------------------------------------------------------------}

function TFileSystem.GetFolder(const Name: WideString): IDispatch;

var
   Folder : objFolder.TFolder;

begin
   if DirectoryExists(Name) then
   begin
      Folder := objFolder.TFolder.Create;
      Folder.Name := ExpandFilename(Name);
      Result := Folder;
   end
   else begin
      Result := nil;
   end;
end;
   :
   :
   :
```

Implement Folder Object

The Folder object will be the second object we implement. The folder object itself contains two collections: Folders and Files.

You will note that whenever an object is returned by a method, it is always returned as IDispatch. As you recall from the beginner's guide, the reason you return objects as IDispatch (or IUnknown) is that you are giving the calling application an interface it knows how to work with (refer to the beginner's guide for more details).

Add Methods and Properties

The Folder object will have two methods and two read-only properties.

Method	Description
Folders	Returns a Folders object.
Files	Returns Files object.

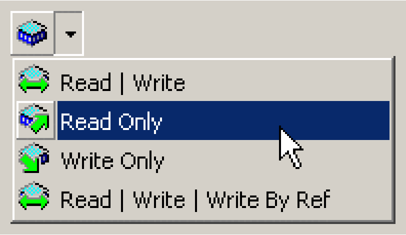
Property	Description
Name	*Read only*. Returns the name of the folder.
Path	*Read only*. Returns the path of this folder.

1. Select **View|Type Library** from the main menu.
2. Select the "IFolder" interface.

3. Click the **New Method** button in the toolbar.
4. Select the **Attributes** tab.
5. In the **Name** field type "Folders".
6. Select the **Parameters** tab.
7. Change **Return Type** to "IDispatch".

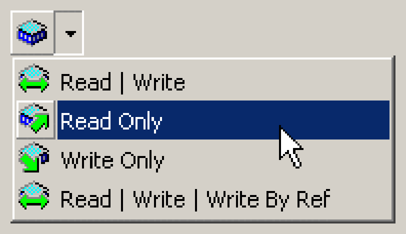
8. Click the **New Method** button in the toolbar.
9. Select the **Attributes** tab.
10. In the **Name** field type "Files".
11. Select the **Parameters** tab.
12. Change **Return Type** to "IDispatch".

13. Click the **New Property** dropdown button in the toolbar.
14. Select **Read Only** from the menu.



15. Select the **Attributes** tab.
16. In the **Name** field type "Name".
17. In the **ID** field type "0".
18. Select "WideString" in the **Type** field.
19. Select the **Flags** tab.
20. Check **UI Default**.

21. Click the **New Property** dropdown button in the toolbar.
22. Select **Read Only** from the menu.



23. Select the **Attributes** tab.
24. In the **Name** field type "Path".
25. Select "WideString" in the **Type** field.

26. Save your project.

Implement the Methods and Properties

Like the FileSystem object, we will not go into detail with the Folder object. Enter the entire implementation below to complete the Folder object:

```
   :
type
  TFolder = class(TAutoObject, IFolder)
  private
    FName : string;
    procedure SetName (Value : string);
  protected
    function Get_Name: WideString; safecall;
    function Files: IDispatch; safecall;
    function Folders: IDispatch; safecall;
    function Get_Path: WideString; safecall;
  public
    property Name : string read FName write SetName;
  end;

{-------------------------------------------------------------------------------}

implementation

uses SysUtils, ComServ, objFolders, objFiles;

{-------------------------------------------------------------------------------}

function ExtractLastFolder (Path : string) : string;

var
   I : Integer;

begin
   if (Length(Path) > 0) and (Path[Length(Path)] = '\') then
   begin
      Delete(Path,Length(Path),1);
   end;
   I := LastDelimiter('\',Path);
   Result := Copy(Path,I+1,Length(Path));
   if (Length(Result) > 0) and (Result[Length(Result)] = ':') then
   begin
      Result := Result+'\';
   end;
end;

{---------------------------------------------------------------------------}

procedure TFolder.SetName (Value : string);

begin
   FName := ExpandFilename(Value);
   if (Length(FName) > 0) and (FName[Length(FName)] <> '\') then
   begin
      FName := FName+'\';
   end;
end;

{---------------------------------------------------------------------------}

function TFolder.Get_Name: WideString;

begin
   Result := ExtractLastFolder(FName);
end;

{---------------------------------------------------------------------------}

function TFolder.Files: IDispatch;

var
   Files : objFiles.TFiles;

begin
   Files := objFiles.TFiles.Create;
   Files.Path := FName;
   Result := Files;
end;

{---------------------------------------------------------------------------}

function TFolder.Folders: IDispatch;

var
   Folders : objFolders.TFolders;

begin
   Folders := objFolders.TFolders.Create;
   Folders.Path := FName;
   Result := Folders;
end;

{---------------------------------------------------------------------------}

function TFolder.Get_Path: WideString;

var
   I : Integer;

begin
   I := Length(ExtractLastFolder(FName));
   Result := Copy(FName,1,Length(FName)-I-1);
end;
   :
   :
```

Implement Files Collection

The Files collection is the easiest collection we will create. It will be a collection of strings that represent all the files within a folder.

Add Methods and Properties

The Files collection will have one method and two read-only properties.

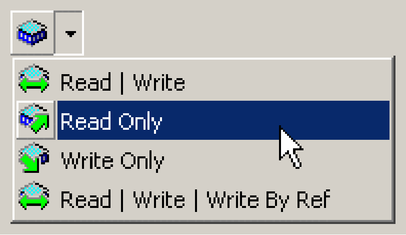
Method	Description
Item	Returns a single file in the collection.

Property	Description
_NewEnum	*Read only*. Returns an IEnumVARIANT interface.
Count	*Read only*. Returns the number of files in the collection.

1. Select **View|Type Library** from the main menu.
2. Select the "IFiles" interface.

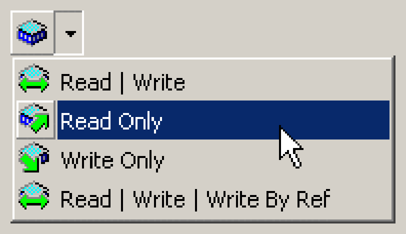
3. Click the **New Method** button in the toolbar.
4. Select the **Attributes** tab.
5. In the **Name** field type "Item".
6. In the **ID** field type "0".
7. Select the **Parameters** tab.
8. Change **Return Type** to "WideString".
9. Click the **Add** button to add a new parameter.
10. In the **Name** column for the parameter type "Index".
11. In the **Type** column select "Integer".
12. Select the **Flags** tab.
13. Check **UI Default**.

14. Click the **New Property** dropdown button in the toolbar.
15. Select **Read Only** from the menu.



16. Select the **Attributes** tab.
17. In the **Name** field type "_NewEnum".
18. In the **ID** field type "-4".
19. Select "IUnknown" in the **Type** field.
20. Select the **Flags** tab.
21. Check **Hidden**.

22. Click the **New Property** dropdown button in the toolbar.
23. Select **Read Only** from the menu.



24. Select the **Attributes** tab.
25. In the **Name** field type "Count".
26. Select "Integer" in the **Type** field.

27. Save your project.

Modify the Type Library

Before we can fully implement the collection, we need to indicate that in addition to implementing IFiles interface, the Files coclass also implements the IEnumVARIANT interface.

1. Select **View|Type Library** from the main menu.
2. Select the "Files" coclass.
3. Select the **Implements** tab.
4. Right-click in the box and select **Insert Interface** from the popup menu.
5. Select "IEnumVARIANT" from the list.
6. Click the **OK** button.
7. Save your project.

Implement the Class Prototype

The first step to implementing the coclass is to implement the class prototype.

```
   :
   :
   :
uses Classes, ComObj, ActiveX, COMExample_TLB, StdVcl;

type
  TFiles = class(TAutoObject, IFiles, IEnumVARIANT)
  private
    FPath  : string;
    FItems : TStringList;
    FIndex : Integer;
    procedure SetPath (Value : string);
  protected
    function Get__NewEnum : IUnknown; safecall;
    function Get_Count : Integer; safecall;
    function Item (Index : Integer) : WideString; safecall;

    { IEnumVARIANT }
    function Next (celt : LongWord; out rgVar : OleVariant; out pCeltFetched : LongWord) : HRESULT;
stdcall;
    function Skip (celt : LongWord) : HRESULT; stdcall;
    function Reset : HRESULT; stdcall;
    function Clone (out Enum : IEnumVARIANT) : HRESULT; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    property Path : string read FPath write SetPath;
  end;
```

```
   :
   :
   :
```

Implement the SetPath, AfterConstruction, and BeforeDestruction methods

The first methods we'll implement on serve a support role in this class. We'll get these methods out of the way first.

```
   :
   :
   :
implementation

uses ComServ, SysUtils;

{-------------------------------------------------------------------------------}

procedure TFiles.SetPath (Value : string);

var
   SearchRec : TSearchRec;

begin
   FItems.Clear;
   FIndex := 0;
   FPath := Value;
   if FindFirst(FPath+'*.*',faAnyFile,SearchRec) = 0 then
   begin
      repeat
         if not (SearchRec.Attr and faDirectory = faDirectory) then
         begin
            FItems.Add(SearchRec.Name);
         end;
      until FindNext(SearchRec) <> 0;
   end;
   FindClose(SearchRec);
end;

{-------------------------------------------------------------------------------}

procedure TFiles.AfterConstruction;

begin
   inherited;
   FItems := TStringList.Create;
end;

{-------------------------------------------------------------------------------}

procedure TFiles.BeforeDestruction;

begin
   FItems.Free;
   inherited;
end;
   :
   :
   :
```

Implement the _NewEnum property

The _NewEnum property must return an interface pointer to an object that implements IEnumVARIANT. Because the Files collection also implements IEnumVARIANT, all we need to do is return a reference to the same object.

```
   :
   :
   :
function TFiles.Get__NewEnum: IUnknown;

begin
   Result := Self as IEnumVARIANT;
end;
   :
   :
   :
```

Implement the Count property

The Count property simply returns the number of items in the collection.

```
   :
   :
   :
function TFiles.Get_Count : Integer;

begin
   Result := FItems.Count;
end;
   :
   :
   :
```

Implement the Item method

The Item method returns a filename based on the index passed. For simplicity's sake, we're going to let the handling of any exceptions raised when the index is out of bounds fall to the calling process.

```
   :
   :
   :
function TFiles.Item (Index : Integer) : WideString;

begin
   Result := FItems[Index-1];
end;
   :
   :
   :
```

Implement the Next method

The Next method returns the next *celt* items in the collection.

```
   :
   :
   :
function TFiles.Next (celt : LongWord; var rgVar : OleVariant; out pCeltFetched : LongWord) : HRESULT;

type
```

```
   TVariantList = array [0..0] of OleVariant;

var
   I : LongWord;

begin
   I := 0;

   while (I < celt) and (FIndex < FItems.Count) do
   begin
      TVariantList(rgVar)[I] := FItems[Integer(I)+FIndex];
      Inc(I);
      Inc(FIndex);
   end;

   if (@pCeltFetched <> nil) then
   begin
      pCeltFetched := I;
   end;

   if (I = celt) then
   begin
      Result := S_OK;
   end
   else begin
      Result := S_FALSE;
   end;
end;
   :
   :
   :
```

Implement the Skip method

The Skip method attempts to skip the next *celt* items in the collection.

```
   :
   :
   :
function TFiles.Skip (celt : LongWord) : HRESULT;

begin
   if (FIndex+Integer(celt)) <= FItems.Count then
   begin
      Inc(FIndex,celt);
      Result := S_OK;
   end
   else begin
      FIndex := FItems.Count;
      Result := S_FALSE;
   end;
end;
   :
   :
   :
```

Implement the Reset method

The Reset method sets the current position to the first item in the collection.

```
   :
   :
   :
function TFiles.Reset : HRESULT;

begin
   FIndex := 0;
   Result := S_OK;
end;
   :
   :
   :
```

Implement the Clone method

The Clone method attempts to make a copy of the collection's current state.

```
   :
   :
   :
function TFiles.Clone (out ppEnum : IEnumVARIANT) : HRESULT;

var
   Files : TFiles;

begin
   Files := TFiles.Create;
   Files.Path := FPath;
   ppEnum := Files as IEnumVARIANT;
   Result := S_OK;
end;
   :
   :
   :
```

Implement Folders Collection

The Folders collection is the last collection we will create. It will be a collection of objects that represent all the folders within a folder.

Add Methods and Properties

The Folders collection will have one method and two read-only properties.

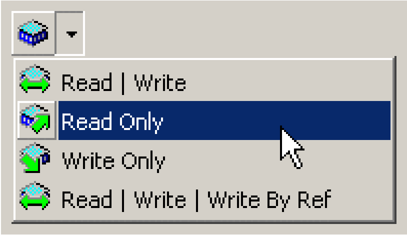
Method	Description
Item	Returns a single Folder object in the collection.

Property	Description
_NewEnum	*Read only*. Returns an IEnumVARIANT interface.
Count	*Read only*. Returns the number of folders in the collection.

1. Select **View|Type Library** from the main menu.
2. Select the "IFolders" interface.

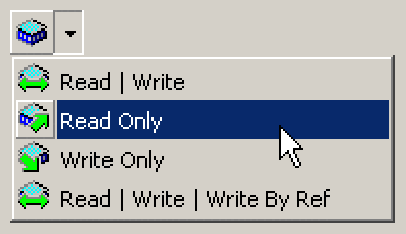
3. Click the **New Method** button in the toolbar.
4. Select the **Attributes** tab.
5. In the **Name** field type "Item".
6. In the **ID** field type "0".
7. Select the **Parameters** tab.
8. Change **Return Type** to "IDispatch".
9. Click the **Add** button to add a new parameter.
10. In the **Name** column for the parameter type "Index".
11. In the **Type** column select "Integer".
12. Select the **Flags** tab.
13. Check **UI Default**.

14. Click the **New Property** dropdown button in the toolbar.
15. Select **Read Only** from the menu.



16. Select the **Attributes** tab.
17. In the **Name** field type "_NewEnum".
18. In the **ID** field type "-4".
19. Select "IUnknown" in the **Type** field.
20. Select the **Flags** tab.
21. Check **Hidden**.

22. Click the **New Property** dropdown button in the toolbar.
23. Select **Read Only** from the menu.



24. Select the **Attributes** tab.
25. In the **Name** field type "Count".
26. Select "Integer" in the **Type** field.

27. Save your project.

Modify the Type Library

Before we can fully implement the collection, we need to indicate that in addition to implementing IFolders interface, the Folders coclass also implements the IEnumVARIANT interface.

1. Select **View|Type Library** from the main menu.
2. Select the "Folders" coclass.
3. Select the **Implements** tab.
4. Right-click in the box and select **Insert Interface** from the popup menu.
5. Select "IEnumVARIANT" from the list.
6. Click the **OK** button.
7. Save your project.

Implement the Class Prototype

The first step to implementing the coclass is to implement the class prototype.

```
   :
   :
   :
uses Classes, ComObj, ActiveX, COMExample_TLB, StdVcl;

type
  TFolders = class(TAutoObject, IFolders, IEnumVARIANT)
  private
    FPath  : string;
    FItems : TStringList;
    FIndex : Integer;
    procedure SetPath (Value : string);
  protected
    function Get__NewEnum : IUnknown; safecall;
    function Get_Count : Integer; safecall;
    function Item (Index : Integer) : WideString; safecall;

    { IEnumVARIANT }
    function Next (celt : LongWord; out rgVar : OleVariant; out pCeltFetched : LongWord) : HRESULT;
stdcall;
    function Skip (celt : LongWord) : HRESULT; stdcall;
    function Reset : HRESULT; stdcall;
    function Clone (out Enum : IEnumVARIANT) : HRESULT; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    property Path : string read FPath write SetPath;
  end;
```

```
   :
   :
   :
```

Implement the SetPath, AfterConstruction, and BeforeDestruction methods

The first methods we'll implement on serve a support role in this class. We'll get these methods out of the way first.

```
   :
   :
   :
implementation

uses ComServ, SysUtils, objFolder;

{------------------------------------------------------------------------------}

procedure TFolders.SetPath (Value : string);

var
   SearchRec : TSearchRec;

begin
   FItems.Clear;
   FIndex := 0;
   FPath := Value;
   if FindFirst(FPath+'*.*',faAnyFile,SearchRec) = 0 then
   begin
      repeat
         if (SearchRec.Attr and faDirectory = faDirectory) then
         begin
            if (SearchRec.Name <> '.') and (SearchRec.Name <> '..') then
            begin
               FItems.Add(SearchRec.Name);
            end;
         end;
      until FindNext(SearchRec) <> 0;
   end;
   FindClose(SearchRec);
end;

{------------------------------------------------------------------------------}

procedure TFolders.AfterConstruction;

begin
   inherited;
   FItems := TStringList.Create;
end;

{------------------------------------------------------------------------------}

procedure TFolders.BeforeDestruction;

begin
   FItems.Free;
   inherited;
end;
   :
   :
```

Implement the _NewEnum property

The _NewEnum property must return an interface pointer to an object that implements IEnumVARIANT. Because the Folders collection also implements IEnumVARIANT, all we need to do is return a reference to the same object.

```
   :
   :
   :
function TFolders.Get__NewEnum: IUnknown;

begin
   Result := Self as IEnumVARIANT;
end;
   :
   :
   :
```

Implement the Count property

The Count property simply returns the number of items in the collection.

```
   :
   :
   :
function TFolders.Get_Count : Integer;

begin
   Result := FItems.Count;
end;
   :
   :
   :
```

Implement the Item method

The Item method returns a filename based on the index passed. For simplicity's sake, we're going to let the handling of any exceptions raised when the index is out of bounds fall to the calling process.

```
   :
   :
   :
function TFolders.Item (Index : Integer) : IDispatch;

var
   Folder : objFolder.TFolder;

begin
   Folder := objFolder.TFolder.Create;
   Folder.Name := FPath+FItems[Index-1];
   Result := Folder as IDispatch;
end;
   :
   :
   :
```

Implement the Next method

The Next method returns the next *celt* items in the collection.

```
   :
```

```
   :
   :
function TFolders.Next (celt : LongWord; var rgVar : OleVariant; out pCeltFetched : LongWord) : HRESULT;

type
   TVariantList = array [0..0] of OleVariant;

var
   Folder : objFolder.TFolder;
   I      : LongWord;

begin
   I := 0;

   while (I < celt) and (FIndex < FItems.Count) do
   begin
      Folder := objFolder.TFolder.Create;
      Folder.Name := FPath+FItems[Integer(I)+FIndex];
      TVariantList(rgVar)[I] := Folder as IDispatch;
      Inc(I);
      Inc(FIndex);
   end;

   if (@pCeltFetched <> nil) then
   begin
      pCeltFetched := I;
   end;

   if (I = celt) then
   begin
      Result := S_OK;
   end
   else begin
      Result := S_FALSE;
   end;
end;
   :
   :
   :
```

Implement the Skip method

The Skip method attempts to skip the next *celt* items in the collection.

```
   :
   :
   :
function TFolders.Skip (celt : LongWord) : HRESULT;

begin
   if (FIndex+Integer(celt)) <= FItems.Count then
   begin
      Inc(FIndex,celt);
      Result := S_OK;
   end
   else begin
      FIndex := FItems.Count;
      Result := S_FALSE;
   end;
end;
   :
```

Implement the Reset method

The Reset method sets the current position to the first item in the collection.

```
   :
   :
   :
function TFolders.Reset : HRESULT;

begin
   FIndex := 0;
   Result := S_OK;
end;
   :
   :
   :
```

Implement the Clone method

The Clone method attempts to make a copy of the collection's current state.

```
   :
   :
   :
function TFolders.Clone (out ppEnum : IEnumVARIANT) : HRESULT;

var
   Folders : TFolders;

begin
   Folders := TFolders.Create;
   Folders.Path := FPath;
   ppEnum := Folders as IEnumVARIANT;
   Result := S_OK;
end;
   :
   :
   :
```

Testing the Automation Server

There are several ways to test you automation server. The easiest way is to use either ASP or Visual Basic. For the purposes of this guide, we will limit our testing to ASP.

Once you have compiled your project, register your automation server either with your development tool (if supported) or using REGSRVR32.EXE located in your system (Windows 95/98/98se/Me) or system32 (Windows NT/2000) directory.

Create a new ASP file on your web server and type in the code at the end of this page. The images referenced in the ASP file are included in the source code that was packaged with this guide.

Once you have set up your web server, open Internet Explorer and enter the URL for the asp file. You should see a list of files and folders on the root directory of your C drive.

```
<%@ Language=VBScript %>
<%

Option Explicit

%>

<html>
<head>
<title>COMExample</title>
<style>
BODY, P, TABLE, TH, TD, DL, DT, DD, LI, UL
{
font-family: verdana;
font-size: 8pt;
font-weight: normal;
color: black;
}
.Border
{
BORDER-TOP: 1px solid black;
BORDER-BOTTOM: 1px solid black;
BORDER-LEFT: 1px solid black;
BORDER-RIGHT: 1px solid black;
}
.BorderBottom
{
BORDER-BOTTOM: 1px solid black;
}
TH
{
color: white;
background-color: black;
font-weight: bold;
}
.Shaded
{
background-color: #cccccc;
}
.LightShaded
{
background-color: #dddddd;
}
.White, .TableHidden
{
background-color: white;
}
.Black
{
background-color: black;
}
A
{
color: blue;
}
.TableHidden
{
display: none;
}
</style>
</head>
<body>

<%

Dim objFileSystem
Dim objFolder
Dim objSubFolder
Dim objFile

Set objFileSystem = Server.CreateObject("COMExample.FileSystem")

If Len(Trim(Request("Folder"))) > 0 Then
   Set objFolder = objFileSystem.GetFolder(Request("Folder"))
Else
   Set objFolder = objFileSystem.GetFolder("c:\")
End If

'Folder

Response.Write("<table border=0 cellspacing=1 cellpadding=0 width='100%'>" & vbCrLf)
Response.Write("<tr>" & vbCrLf)
Response.Write("<td class=border>" & vbCrLf)
Response.Write("<table border=0 cellspacing=0 cellpadding=2 width='100%' class=black>" & vbCrLf)
Response.Write("<tr>" & vbCrLf)
Response.Write("<th colspan=2 valign=top align=left>" & Server.URLEncode(objFolder.Path &
objFolder.Name) & "</th>" & vbCrLf)
Response.Write("</tr>" & vbCrLf)
Response.Write("</table>" & vbCrLf)
Response.Write("<table border=0 cellspacing=1 cellpadding=2 width='100%' class=white>" & vbCrLf)

'Folders

For Each objSubFolder In objFolder.Folders
   Response.Write("<tr>" & vbCrLf)
   Response.Write("<td width='1%' align=right valign=center class=lightshaded nowrap><a
href='index.asp?folder=" & Server.URLEncode(objSubFolder.Path & objSubFolder.Name) & "'><img
src='folder.gif' border=0 width=16 height=16></a></td>" & vbCrLf)
   Response.Write("<td valign=center class=lightshaded nowrap><a href='index.asp?folder=" &
Server.URLEncode(objSubFolder.Path & objSubFolder.Name) & "'>" & objSubFolder.Name & "</a></td>" &
vbCrLf)
   Response.Write("</tr>" & vbCrLf)
Next

'Files

For Each objFile In objFolder.Files
   Response.Write("<tr>" & vbCrLf)
   Response.Write("<td width='1%' align=right valign=center class=lightshaded nowrap><img src='file.gif'
border=0 width=16 height=16></td>" & vbCrLf)
   Response.Write("<td valign=center class=lightshaded nowrap>" & Server.URLEncode(objFile) & "</td>" &
vbCrLf)
   Response.Write("</tr>" & vbCrLf)
```

```
Next

Response.Write("</table>" & vbCrLf)
Response.Write("</td>" & vbCrLf)
Response.Write("</tr>" & vbCrLf)
Response.Write("</table>" & vbCrLf)
Response.Write("</td>" & vbCrLf)
Response.Write("</tr>" & vbCrLf)

Set objFolder = Nothing
Set objFileSystem = Nothing

%>

</body>
</html>
```

Resources

COM References

1. Microsoft Corporation, **Microsoft Developer Network Library, Component Object Model (COM) SDK**, http://msdn.microsoft.com/library/en-us/com/hh/com/comportal_3qn9.asp

2. George Reilly et al, **Beginning ATL COM Programming**, *Wrox*, ISBN: 1861000111

3. Sing Li et al, **Professional Visual C++ 5: ActiveX/COM Control Programming**, *Wrox*, ISBN: 1861000375

4. Richard Grimes, **Professional ATL COM Programming**, *Wrox*, ISBN: 1861001401

5. Richard Grimes, **ATL COM Programmer's Reference**, *Wrox*, ISBN: 1861002491

6. Richard Grimes et al, **Beginning ATL 3 COM Programming**, *Wrox*, ISBN: 1861001207

7. Alex Homer et al, **Beginning Components for ASP**, *Wrox*, ISBN: 1861002882

8. Eric Harmon, **Delphi COM Programming**, *Macmillan Technical Publishing*, ISBN: 1578702216

IDL References

1. Microsoft Corporation, **Microsoft Developer Network Library, Microsoft Interface Definition Language SDK**, http://msdn.microsoft.com/library/en-us/midl/hh/midl/midlstart_4ox1.asp

2. Martin Gudgin, **Essential IDL: Interface Design for COM (The DevelopMentor Series)**, *Addison-Wesley Pub Co*, ISBN: 0201615959

3. Al Major, **COM IDL and Interface Design**, *Wrox*, ISBN: 1861002254