.NET Matters
IFileOperation in Windows Vista
Stephen Toub

Code download available at: NetMatters2007_12.exe (160 KB)
Browse the Code Online

Q I have a bunch of file operations I'd like to execute as a batch, and I'd like to get the standard Windows® progress UI for my operations. I know I could use classes from the System.IO namespace to do all of the file operations one after the other, but then I would need to create my own progress UI, which is way more work than I wish to undertake. I noticed that Windows Vista® includes a new IFileOperations interface, but none of the samples I've seen demonstrate how to use this from managed code. How can it be done?
Q I have a bunch of file operations I'd like to execute as a batch, and I'd like to get the standard Windows® progress UI for my operations. I know I could use classes from the System.IO namespace to do all of the file operations one after the other, but then I would need to create my own progress UI, which is way more work than I wish to undertake. I noticed that Windows Vista® includes a new IFileOperations interface, but none of the samples I've seen demonstrate how to use this from managed code. How can it be done?
A Windows Vista does indeed include a new copy engine that supports exactly what you're looking to do. However, it's possible that previously existing functionality may meet your needs. For example, if you want to copy, move, rename, or delete an individual file or directory, you can take advantage of SHFileOperation (exposed from shell32.dll), which is already wrapped by the Visual Basic® runtime. If you're using Visual Basic 2005, you can simply use functionality from the My namespace, for example:
A Windows Vista does indeed include a new copy engine that supports exactly what you're looking to do. However, it's possible that previously existing functionality may meet your needs. For example, if you want to copy, move, rename, or delete an individual file or directory, you can take advantage of SHFileOperation (exposed from shell32.dll), which is already wrapped by the Visual Basic® runtime. If you're using Visual Basic 2005, you can simply use functionality from the My namespace, for example:

```
My.Computer.FileSystem.CopyDirectory(
    sourcePath, destinationPath, UIOption.AllDialogs)
```

Accomplishing the same thing in C# involves only a little more work, adding a reference to Microsoft.VisualBasic.dll (from the Microsoft® .NET Framework installation directory) and using code such as the following:

```
using Microsoft.VisualBasic.FileIO;
...
FileSystem.CopyDirectory(
    sourcePath, destinationPath, UIOption.AllDialogs);
```

When run, this will result in the same progress UI you'd see if you were doing the same file operations from Windows Explorer. In fact, when running on Windows Vista, you automatically get the new Window Vista progress UI, as shown in **Figure 1**.

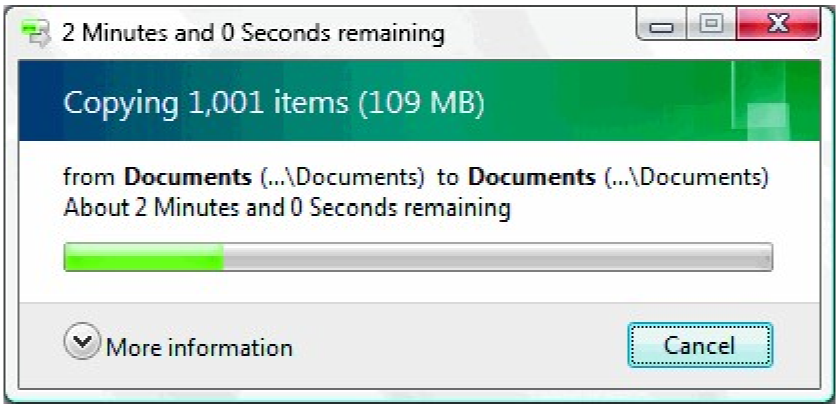
2 Minutes and 0 Seconds remaining
Copying 1,001 items (109 MB)
from **Documents** (...\Documents) to **Documents** (...\Documents)
About 2 Minutes and 0 Seconds remaining
More information
Cancel

Figure 1 **Windows Vista Progress Dialog**

However, SHFileOperation and the wrappers provided by .NET are limited to operating on one file system object (which could be a directory) and one operation (copy, move, rename, or delete) per call; you can't mix and match or operate on disparate files within the same operation. That's where IFileOperation comes in. IFileOperation is the successor to SHFileOperation and provides a wealth of new functionality, including the ability to batch a bunch of operations, which can be a mix of copies, renames, moves, deletes, and even the creation of new items. It can also operate on any set of files, not just those in the same directory. These are just a few of the cool features provided through IFileOperation; for a more in-depth look, see shellrevealed.com/blogs/shellblog/archive/2007/04/16/IFileOperation-_1320_-Part-1_3A00_-Introduction.aspx.

Unfortunately, as you point out, at the time of this writing there are currently no managed wrappers provided by Microsoft to expose this functionality from managed code. As such, I've built one. You can download the code from the MSDN® Magazine Web site, and I'll spend the rest of this answer walking through how to use it and how it was implemented. Note that my wrapper is not meant to expose all of IFileOperation's functionality, nor is it meant to completely abstract away all of the underlying bits and bytes and provide a perfect .NET wrapper that follows all of the standard .NET design guidelines; it's simply meant to easily expose IFileOperation's functionality so you can get up and running with the basics in your Windows Vista-targeting applications today.

Let's begin with the IFileOperation interface itself, for which I've shown my managed definition in **Figure 2** (this is based on the IDL from the ShObjIdl.idl file in the Windows SDK). The interface is designed such that you schedule with it all of the operations you want performed, and then when you're ready, you tell it to execute them all. At that point, it'll begin execution, and if it senses that the operations may take a noticeable amount of time, it'll present the standard Windows Vista progress dialog, complete with descriptions about everything it is doing. To schedule actions, you use the IFileOperation methods CopyItem, MoveItem, RenameItem, DeleteItem, and NewItem. (The interface also exposes plural versions of those methods, but since their behavior can be simulated as necessary with the singular versions and because they're more difficult to work with from managed code, I won't be discussing them further.) Each of these methods merely expresses your intent for the operation to happen, but the actions aren't actually executed at that time. Once all of the operations have been scheduled, you use the PerformOperations method to run all of the operations previously scheduled. Details about how the operations are performed can be configured with several methods (such as SetOwnerWindow and SetOperationFlags) on IFileOperation before calling PerformOperations. IFileOperation also supports a nice callback notification system, whereby you can register an IFileOperationProgressSink and receive callbacks for all sorts of events, including pre- and post-operation notification, progress updates, and the like. You can register a sink using IFileOperation's Advise method (unregistering it using Unadvise when you're done) that will cause the sink to be used for all of the operations performed, or you can register a sink for individual operations using the last parameter to each of the singular operation methods. My managed definition for IFileOperationProgressSink is shown in **Figure 3**.(This is also based on the ShObjIdl.idl file from the Windows SDK.)

⊞ **Figure 3 IFileOperationProgressSink**

```
[ComImport]
[Guid("04b0f1a7-9490-44bc-96e1-4296a31252e2")]
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
public interface IFileOperationProgressSink
{
    void StartOperations();
    void FinishOperations(uint hrResult);

    void PreRenameItem(uint dwFlags, IShellItem psiItem,
        [MarshalAs(UnmanagedType.LPWStr)] string pszNewName);
    void PostRenameItem(uint dwFlags, IShellItem psiItem,
        [MarshalAs(UnmanagedType.LPWStr)] string pszNewName,
        uint hrRename, IShellItem psiNewlyCreated);

    void PreMoveItem(uint dwFlags, IShellItem psiItem, IShellItem
        psiDestinationFolder,
        [MarshalAs(UnmanagedType.LPWStr)] string pszNewName);
    void PostMoveItem(uint dwFlags, IShellItem psiItem,
        IShellItem psiDestinationFolder,
        [MarshalAs(UnmanagedType.LPWStr)] string pszNewName,
        uint hrMove, IShellItem psiNewlyCreated);

    void PreCopyItem(uint dwFlags, IShellItem psiItem,
        IShellItem psiDestinationFolder,
        [MarshalAs(UnmanagedType.LPWStr)] string pszNewName);
    void PostCopyItem(uint dwFlags, IShellItem psiItem,
        IShellItem psiDestinationFolder,
        [MarshalAs(UnmanagedType.LPWStr)] string pszNewName,
        uint hrCopy, IShellItem psiNewlyCreated);

    void PreDeleteItem(uint dwFlags, IShellItem psiItem);
    void PostDeleteItem(uint dwFlags, IShellItem psiItem, uint hrDelete,
        IShellItem psiNewlyCreated);

    void PreNewItem(uint dwFlags, IShellItem psiDestinationFolder,
        [MarshalAs(UnmanagedType.LPWStr)] string pszNewName);
    void PostNewItem(uint dwFlags, IShellItem psiDestinationFolder,
        [MarshalAs(UnmanagedType.LPWStr)] string pszNewName,
        [MarshalAs(UnmanagedType.LPWStr)] string pszTemplateName,
        uint dwFileAttributes, uint hrNew, IShellItem psiNewItem);

    void UpdateProgress(uint iWorkTotal, uint iWorkSoFar);

    void ResetTimer();
    void PauseTimer();
    void ResumeTimer();
}
```

⊞ Figure 2 IFileOperation

```
[ComImport]
[Guid("947aab5f-0a5c-4c13-b4d6-4bf7836fc9f8")]
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
internal interface IFileOperation
{
    uint Advise(IFileOperationProgressSink pfops);
    void Unadvise(uint dwCookie);

    void SetOperationFlags(FileOperationFlags dwOperationFlags);
    void SetProgressMessage(
        [MarshalAs(UnmanagedType.LPWStr)] string pszMessage);
    void SetProgressDialog(
        [MarshalAs(UnmanagedType.Interface)] object popd);
    void SetProperties(
        [MarshalAs(UnmanagedType.Interface)] object pproparray);
    void SetOwnerWindow(uint hwndParent);

    void ApplyPropertiesToItem(IShellItem psiItem);
    void ApplyPropertiesToItems(
        [MarshalAs(UnmanagedType.Interface)] object punkItems);

    void RenameItem(IShellItem psiItem,
        [MarshalAs(UnmanagedType.LPWStr)] string pszNewName,
        IFileOperationProgressSink pfopsItem);
    void RenameItems(
        [MarshalAs(UnmanagedType.Interface)] object pUnkItems,
        [MarshalAs(UnmanagedType.LPWStr)] string pszNewName);

    void MoveItem(
        IShellItem psiItem,
        IShellItem psiDestinationFolder,
        [MarshalAs(UnmanagedType.LPWStr)] string pszNewName,
        IFileOperationProgressSink pfopsItem);
    void MoveItems(
        [MarshalAs(UnmanagedType.Interface)] object punkItems,
        IShellItem psiDestinationFolder);

    void CopyItem(
        IShellItem psiItem,
        IShellItem psiDestinationFolder,
        [MarshalAs(UnmanagedType.LPWStr)] string pszCopyName,
        IFileOperationProgressSink pfopsItem);
    void CopyItems(
        [MarshalAs(UnmanagedType.Interface)] object punkItems,
        IShellItem psiDestinationFolder);

    void DeleteItem(
        IShellItem psiItem,
        IFileOperationProgressSink pfopsItem);
    void DeleteItems(
        [MarshalAs(UnmanagedType.Interface)] object punkItems);

    uint NewItem(
        IShellItem psiDestinationFolder,
        FileAttributes dwFileAttributes,
        [MarshalAs(UnmanagedType.LPWStr)] string pszName,
        [MarshalAs(UnmanagedType.LPWStr)] string pszTemplateName,
        IFileOperationProgressSink pfopsItem);

    void PerformOperations();

    [return: MarshalAs(UnmanagedType.Bool)]
    bool GetAnyOperationsAborted();
}
```

Before I dive into how to use these interfaces directly from managed code, I'll first show you the wrapper I've implemented around them. The outline for my FileOperation class is shown in **Figure 4**. As an example of using this class, let's say I have a directory C:\test that contains two files and a directory: test1.txt, test2.txt, and SampleDir. I want to copy the first file to copy.text, delete the second file, and move the directory to NewDir; I can accomplish that with code like the following:

⊞ **Figure 4 Managed FileOperation Wrapper**

```
class FileOperation : IDisposable
{
    public FileOperation() : this(null);
    public FileOperation(FileOperationProgressSink callbackSink) :
        this(callbackSink, null);
    public FileOperation(
        FileOperationProgressSink callbackSink, IWin32Window owner);

    public void CopyItem(
        string source, string destination, string newName);

    public void MoveItem(
        string source, string destination, string newName);

    public void RenameItem(string source, string newName);

    public void DeleteItem(string source);

    public void NewItem(
        string folderName, string name, FileAttributes attrs);

    public void PerformOperations();

    public void Dispose();
}
```

```
using(FileOperation fileOp = new FileOperation())
{
    fileOp.CopyItem(@"C:\test\test1.txt", @"C:\test", @"copy.txt");
    fileOp.DeleteItem(@"C:\test\test2.txt");
    fileOp.MoveItem(@"C:\test\SampleDir", @"C:\test", @"NewDir");
    fileOp.PerformOperations();
}
```

If I want to receive callback notifications for various actions, I can register a progress sink through FileOperation's constructor. (The FileOperationProgressSink class used here is a default implementation of IFileOperationProgressSink I've provided that traces all events, but more on that shortly.)

```
using(FileOperation fileOp =
    new FileOperation(new FileOperationProgressSink()))
{
    ...
}
```

And if this is running in a Windows Forms application and I want to associate the progress window with my current Form, I can use code like this:

```
Form form = this;
using(FileOperation fileOp = new FileOperation(null, form))
{
    ...
}
```

Easy, right? Let's take a look at how this is implemented. IFileOperation is a COM interface, implemented in Windows Vista by the COM component identified by CLSID_FileOperation, and we can create an instance of one with code like the following:

```
private static readonly Guid CLSID_FileOperation =
    new Guid("3ad05575-8857-4850-9277-11b85bdb8e09");
private static readonly Type FileOperationType =
    Type.GetTypeFromCLSID(CLSID_FileOperation);
```

```
public FileOperation(
    FileOperationProgressSink callbackSink, IWin32Window owner)
{
    _fileOperation = (IFileOperation)
        Activator.CreateInstance(FileOperationType);
    ...
}
```

The created IFileOperation stored in the _fileOperation member variable can then be used for all of the operations in the class. For example, the CopyItem method is implemented as follows:

```
public void CopyItem(string source, string destination, string newName)
{
    ThrowIfDisposed();
    using (ComReleaser<IShellItem> sourceItem =
        CreateShellItem(source))
    using (ComReleaser<IShellItem> destinationItem =
        CreateShellItem(destination))
    {
        _fileOperation.CopyItem(
            sourceItem.Item, destinationItem.Item, newName, null);
    }
}
```

Ignoring ThrowIfDisposed and ComReleaser for a moment, CopyItem first creates an IShellItem instance for the source path and an IShellItem for the destination folder, and then passes these (along with the new name for the item) to the IFileOperation's CopyItem method. All of the operation methods on FileOperation are implemented just like this, and PerformOperations simply delegates to the same method on the underlying IFileOperation:

```
public void PerformOperations()
{
    ThrowIfDisposed();
    _fileOperation.PerformOperations();
}
```

The ComReleaser class shown in the CopyItem implementation is a little helper class that makes it easy to dispose of a COM object using Marshal.FinalReleaseComObject (the objects would be cleaned up eventually by the garbage collector, but, given the frequency with which these may be created, I've opted to ensure they're disposed of as soon as they're no longer being used). The type's constructor accepts a COM object and caches it in a private member variable; its Dispose method then releases that object. It also exposes an Item property that returns the object, making it easy to use in situations such as the one demonstrated. ComReleaser is shown in **Figure 5**.

⊞ **Figure 5 Helper Class for Releasing COM Objects**

```
class ComReleaser<T> : IDisposable where T : class
{
    private T _obj;

    public ComReleaser(T obj)
    {
        if (obj == null) throw new ArgumentNullException("obj");
        if (!obj.GetType().IsCOMObject)
            throw new ArgumentOutOfRangeException("obj");
        _obj = obj;
    }

    public T Item { get { return _obj; } }

    public void Dispose()
    {
        if (_obj != null)
        {
            Marshal.FinalReleaseComObject(_obj);
            _obj = null;
        }
    }
}
```

In order to get a ComReleaser<IShellItem> for a given file or directory path, I take advantage of the SHCreateItemFromParsingName function exported in Windows Vista from shell32.dll. You can provide this function with the path to the file system object and have it provide you back the appropriate IShellItem; my CreateShellItem method then simply wraps this IShellItem with a ComReleaser<IShellItem>:

```
private static ComReleaser<IShellItem> CreateShellItem(string path)
{
    return new ComReleaser<IShellItem>((IShellItem)
        SHCreateItemFromParsingName(path, null, ref _shellItemGuid));
}

[DllImport("shell32.dll", SetLastError=true,
           CharSet=CharSet.Unicode, PreserveSig=false)]
[return: MarshalAs(UnmanagedType.Interface)]
private static extern object SHCreateItemFromParsingName(
    [MarshalAs(UnmanagedType.LPWStr)] string pszPath,
    IBindCtx pbc, ref Guid riid);
```

All that leaves now is the FileOperationProgressSink implementation. The FileOperationProgressSink class implements the IFileOperationProgressSink interface shown in **Figure 3**:

```
public class FileOperationProgressSink : IFileOperationProgressSink
{
   ...
}
```

It does so by providing virtual method implementations for each of the interface's methods. That way, if you want to provide custom behavior to respond to events, you only need to derive from FileOperationProgressSink. As an example, PreRenameItem is the method called before an item is renamed:

```
public virtual void PreRenameItem(
    uint dwFlags, IShellItem psiItem, string pszNewName)
{
#if DEBUG
    string message = string.Format("Renaming {0} to {1}",
        item.GetDisplayName(SIGDN.SIGDN_NORMALDISPLAY), pszNewName);
    Debug.WriteLine(message);
#endif
}
```

However, you can override it in your own progress sink in whatever manner you choose:

```
public class MyProgressSink : FileOperationProgressSink
{
    public virtual void PreRenameItem(
        uint dwFlags, IShellItem psiItem, string pszNewName)
    {
        Console.WriteLine("Goodbye, {0}", item.GetDisplayName(0));
    }
}
```

Then, when you instantiate your FileOperation class, you just provide an instance of your sink:

```
using(FileOperation fileOp = new FileOperation(new MyProgressSink()))
{
    ...
}
```

All in all, the Windows Vista team did a very nice job of designing the IFileOperation API in a way that makes it easily usable from managed code. Note, however, that there is a lot of functionality I haven't exposed here. For example, IFileOperation in Windows Vista not only supports per operation progress sinks (which I haven't exposed), but it also supports providing multiple sinks for all operations by calling IFileOperation::Advise multiple times, once per sink to be registered. Moreover, IFileOperation supports a bunch of operation flags that I haven't exposed for controlling things like directory recursion, what and how dialogs are displayed, how collisions are handled, and so on. It also exposes additional operations I didn't provide wrappers for, such as ones to modify item properties (through the SetProperties, ApplyPropertiesToItem, and ApplyPropertiesToItems methods). Additionally, IFileOperation provides detailed error information about each operation performed. If you need any of that functionality, it should be a straightforward process for you to download the code I've provided from the *MSDN Magazine* Web site and modify it to suit your needs.

One final note about using FileOperation: IFileOperation and its supporting types (and thus my managed FileOperation class) can only be used in a single-threaded apartment (STA). If you're using this in a Windows Forms application, this shouldn't be an issue, since much of Windows Forms itself requires threads to be STA. If you must run in a multithreaded apartment (MTA) situation, you can still use SHFileOperation. Alternatively, you can spin up a new thread with the appropriate threading model to execute the IFileOperation code; for an example of that, see the ApartmentStateSwitcher class from my December 2004 .NET Matters column at msdn.microsoft.com/msdnmag/issues/04/12/NETMatters.

Send your questions and comments for Stephen to netqa@microsoft.com.

Stephen Toub is a Senior Program Manager on the Parallel Computing Platform team at Microsoft. He is also a Contributing Editor for *MSDN Magazine*.