How do I accept files to be opened via IDropTarget instead of on the command line?

devblogs.microsoft.com/oldnewthing/20100503-00

May 3, 2010

Raymond Chen

Commenter Yaron wants to know how to use the new IDropTarget mechanism for receiving a list of files to open. (Also asked by Anthony Wieser as a comment to an article.) The MSDN documentation on Verbs and File Assocations mentions that DDE has been deprecated as a way of launching documents and that you should use the DropTarget method instead. But what is the DropTarget method? (Note that the word *method* here is in the sense of *technique* and not in the C++ sense of *function that belongs to a class*.)

The documentation in MSDN tells you what to do, but it does so very tersely. It says to create a `DropTarget` key under the verb key and create a `Clsid` string value whose data is the string version of the CLSID for your drop target. The documentation tells you to be careful in your `IDropTarget::Drop`, so it stands to reason that `IDropTarget` is the interface that will be used. From context, therefore, you should expect that the shell is going to simulate a drop on your drop target.

You can implement your drop target either as an in-process server or a local server. The in-process case is well-known; nearly all shell extensions are in-process. But using an in-process server for the DropTarget technique only solves half the problem: Sure, the `IDropTarget::Drop` will take place and you will get your `IDataObject`, but you still have to transfer the file list from your shell extension running inside the context menu host to your application. May as well let COM do the heavy lifting of marshalling the data. (Well, okay, maybe using COM is overkill. You might have a lighter weight way of getting the data across, but since that's out of scope for today's exercise, I'll leave it for you to figure out.)

Okay, let's roll up our sleeves and get to it! It turns out that nearly all the work is just creating a COM local server. If you know how to do that already, then I apologize in advance for the oppressive boredom about to fall upon you. I'll try to remember to wake you up when something interesting is about to happen. Note also that I am not an expert on COM local servers, so if you find a discrepancy between what I write and information from people who actually know what they're doing, go with the people who know what they're doing. (Actually, that sentence pretty much applies in general to everything I write.) Indeed, I had never

written a COM local server before now, so all of what you see here is the result of a crash course in COM local servers from reading the documentation. (Translation: You could've done this too.)

Start by adding some header files and a forward reference.

```
#include <shlobj.h>
#include <shellapi.h>
#include <new> // for new(nothrow)
void OpenFilesFromDataObject(IDataObject *pdto);
```

Next, I'm going to steal the `ProcessReference` class which I had created some time ago. It's not the most efficient solution to the problem, but it works well enough, and it's a nice preparatory step in case a shell extension loaded into our process needs to take a process reference. We use the process reference object to keep track of our outstanding objects and locks.

```
ProcessReference *g_ppr;
```

Of course our custom drop target needs a class ID:

```
const CLSID CLSID_Scratch = { ... };
```

I leave it to you to fill in the `CLSID` structure from the output of `uuidgen -s`.

Next, our simple drop target. COM servers need to keep track of the number of objects that have been created, so we'll piggyback off our existing process reference.

```
class SimpleDropTarget : public IDropTarget
{
public:
 SimpleDropTarget() : m_cRef(1) { g_ppr->AddRef(); }
 ~SimpleDropTarget() { g_ppr->Release(); }
 // *** IUnknown ***
 STDMETHODIMP QueryInterface(REFIID riid, void **ppv)
 {
  if (riid == IID_IUnknown || riid == IID_IDropTarget) {
    *ppv = static_cast<IUnknown*>(this);
    AddRef();
    return S_OK;
  }
  *ppv = NULL;
  return E_NOINTERFACE;
 }
 STDMETHODIMP_(ULONG) AddRef()
 {
  return InterlockedIncrement(&m_cRef);
 }
 STDMETHODIMP_(ULONG) Release()
 {
  LONG cRef = InterlockedDecrement(&m_cRef);
  if (cRef == 0) delete this;
  return cRef;
 }
```

Next come the methods of `IDropTarget`, none of which are particularly interesting. We just say that we are going to copy the data.

```
 // *** IDropTarget ***
 STDMETHODIMP DragEnter(IDataObject *pdto,
    DWORD grfKeyState, POINTL ptl, DWORD *pdwEffect)
 {
  *pdwEffect &= DROPEFFECT_COPY;
  return S_OK;
 }
 STDMETHODIMP DragOver(DWORD grfKeyState,
   POINTL ptl, DWORD *pdwEffect)
 {
  *pdwEffect &= DROPEFFECT_COPY;
  return S_OK;
 }
 STDMETHODIMP DragLeave()
 {
  return S_OK;
 }
 STDMETHODIMP Drop(IDataObject *pdto, DWORD grfKeyState,
    POINTL ptl, DWORD *pdwEffect)
 {
  OpenFilesFromDataObject(pdto);
  *pdwEffect &= DROPEFFECT_COPY;
  return S_OK;
 }
private:
 LONG m_cRef;
};
```

People who know how COM servers work wake up: When something is dropped on our drop target, we call `OpenFilesFromDataObject`. That's actually not all that interesting, but at least it's nontrivial. **People who know how COM servers work can go back to sleep now**.

The next part of the code is just setting up our class factory.

```
class SimpleClassFactory : public IClassFactory
{
public:
 // *** IUnknown ***
 STDMETHODIMP QueryInterface(REFIID riid, void **ppv)
 {
  if (riid == IID_IUnknown || riid == IID_IClassFactory) {
    *ppv = static_cast<IUnknown*>(this);
    AddRef();
    return S_OK;
  }
  *ppv = NULL;
  return E_NOINTERFACE;
 }
 STDMETHODIMP_(ULONG) AddRef()
 {
  return 2;
 }
 STDMETHODIMP_(ULONG) Release()
 {
  return 1;
 }
 // *** IClassFactory ***
 STDMETHODIMP CreateInstance(IUnknown *punkOuter, REFIID riid, void **ppv)
 {
    *ppv = NULL;
    if (punkOuter) return CLASS_E_NOAGGREGATION;
    SimpleDropTarget *pdt = new(nothrow) SimpleDropTarget();
    if (!pdt) return E_OUTOFMEMORY;
    HRESULT hr = pdt->QueryInterface(riid, ppv);
    pdt->Release();
    return hr;
 }
 STDMETHODIMP LockServer(BOOL fLock)
 {
  if (!g_ppr) return E_FAIL; // server shutting down
  if (fLock) g_ppr->AddRef(); else g_ppr->Release();
  return S_OK;
 }
};
SimpleClassFactory s_scf;
```

The `LockServer` call takes advantage of our process reference object by forwarding lock and unlock calls into the reference count of the process reference object. This keeps our process running until the server is unlocked.

Remember that COM rules specify that the class factory itself does not count as an outstanding COM object, so we don't use the same `m_punkProcess` trick that we did with our drop target. Instead, we just use a static object.

People who know how COM servers work wake up: The COM server code is pretty much done. Now we're back to user interface programming.

The next part of the code is just copied from our scratch program, with the following changes:

```
BOOL
OnCreate(HWND hwnd, LPCREATESTRUCT lpcs)
{
    g_hwndChild = CreateWindow(
        TEXT("listbox"), NULL, WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        0, 0, 0,0, hwnd, (HMENU)1, g_hinst, 0);
    return TRUE;
}
```

The list box is not an important part of the program. We'll just fill it with data to prove that we actually did something.

```
void OpenFilesFromDataObject(IDataObject *pdto)
{
 if (!g_hwndChild) { /* need to create a new main window */ }
 FORMATETC fmte = { CF_HDROP, NULL, DVASPECT_CONTENT,
                    -1, TYMED_HGLOBAL };
 STGMEDIUM stgm;
 if (SUCCEEDED(pdto->GetData(&fmte, &stgm))) {
  HDROP hdrop = reinterpret_cast<HDROP>(stgm.hGlobal);
  UINT cFiles = DragQueryFile(hdrop, 0xFFFFFFFF, NULL, 0);
  for (UINT i = 0; i < cFiles; i++) {
   TCHAR szFile[MAX_PATH];
   UINT cch = DragQueryFile(hdrop, i, szFile, MAX_PATH);
   if (cch > 0 && cch < MAX_PATH) {
    ListBox_AddString(g_hwndChild, szFile);
   }
  }
  ReleaseStgMedium(&stgm);
 }
}
```

The `OpenFilesFromDataObject` function does only enough work to prove that it actually got the list of file names. When we receive a data object from the simulated drop, we retrieve the `HDROP` and enumerate the files in it. For each file, we add it to the list box.

There's some code I've not bothered to write: Namely, if a request to open some files comes in after the user closed our main window, we need to open a new main window. (Exercise: How can this happen?)

Another difference between this program and real life is that in real life, your `OpenFiles-FromDataObject` would do some real work. *But wait*, if your function does any actual work, you should just `AddRef` the data object and return, so that the shell can return to

interacting with the user. If you stop to do a lot of work before returning, the shell will lock up because it's waiting for your drop to complete.

```
// Version of OpenFilesFromDataObject that is more
// appropriate for real life.
void OpenFilesFromDataObject(IDataObject *pdto)
{
 if (!g_hwndChild) { /* need to create a new main window */ }
 pdto->AddRef();
 PostMessage(GetParent(g_hwndChild), WM_OPENFILES, 0,
             reinterpret_cast<LPARAM>(pdto));
}
case WM_OPENFILES:
 IDataObject *pdto = reinterpret_cast<IDataObject*>(lParam);
 ... rest of code from the original OpenFilesFromDataObject ...
 pdto->Release();
 break;
```

In real life, you just `AddRef` the data object and then post a message to finish processing it later. The aim here is to release the shell thread as quickly as possible. When the posted message is received, we can extract the information from the data object at our leisure. **People who know how COM servers work can go back to sleep now**.

Finally, we hook up our class factories to the main program:

```
int WINAPI WinMain(HINSTANCE hinst, HINSTANCE hinstPrev,
                   LPSTR lpCmdLine, int nShowCmd)
{
    MSG msg;
    HWND hwnd;
    g_hinst = hinst;
    if (!InitApp()) return 0;
    if (SUCCEEDED(CoInitialize(NULL))) {/* In case we use COM */
        HRESULT hrRegister;
        DWORD dwRegisterCookie;
        {
            ProcessReference ref;
            g_ppr = &ref;
            hrRegister = CoRegisterClassObject(CLSID_Scratch, &s_scf,
                  CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE,
                  &dwRegisterCookie);
            hwnd = CreateWindow(
                TEXT("Scratch"),               /* Class Name */
                TEXT("Scratch"),               /* Title */
                WS_OVERLAPPEDWINDOW,           /* Style */
                CW_USEDEFAULT, CW_USEDEFAULT,  /* Position */
                CW_USEDEFAULT, CW_USEDEFAULT,  /* Size */
                NULL,                          /* Parent */
                NULL,                          /* No menu */
                hinst,                         /* Instance */
                0);                            /* No special parameters */
            if (CompareStringA(LOCALE_INVARIANT, NORM_IGNORECASE,
                 lpCmdLine, -1, "-Embedding", -1) != CSTR_EQUAL &&
                CompareStringA(LOCALE_INVARIANT, NORM_IGNORECASE,
                 lpCmdLine, -1, "/Embedding", -1) != CSTR_EQUAL) {
                /* OpenFilesFromCommandLine(); */
            }
            ShowWindow(hwnd, nShowCmd);
            while (GetMessage(&msg, NULL, 0, 0)) {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
            g_hwndChild = NULL;
        } // wait for process references to die
        g_ppr = NULL;
        if (SUCCEEDED(hrRegister)) {
            CoRevokeClassObject(dwRegisterCookie);
        }
        CoUninitialize();
    }
    return 0;
}
```

After creating our process reference, we register our class factory by calling `CoRegisterClassObject`. We do this even if not invoked by COM, because we want COM to be able to find us once we're up and running: If the user runs the application manually and then

double-clicks an associated document, we want that document to be handed to us rather than having COM launch another copy of the program.

After creating the window, we check if the command line is `-Embedding` or `/Embedding`. This is the magic command line switch which COM gives us when we are being launched as a local server. If we don't have that switch, then we're being launched with a file name on our command line, so proceed with "old school" command line parsing. (I didn't bother writing the `OpenFilesFromCommandLine` function since it is irrelevant to the topic.)

After our message loop exits, we clear the `g_hwndChild` so `OpenFilesFromDataObject` knows that there is no main window any more. In real life, we'd have to create a new main window and restart the message loop.

Once all outstanding COM objects and server locks and process references are gone, we can tear down the process. We unregister the COM server (if we registered it) so that COM won't try to ask us to open any more documents. (COM will instead launch a new copy of the program.)

And that's it.

Oh wait, we also have to register this program so COM and the shell can find us.

Registering the COM server is just a matter of setting the registry key

```
[HKCR\CLSID\{...}\LocalServer32]
@="C:\\Path\\To\\scratch.exe"
```

You probably should also set a friendly name into `HKCR\CLSID\{...}` so people will have a clue what your server is for.

People who know how COM servers work wake up: After we register our class with COM, we can register it with the shell. For demonstration purposes, we'll make our command a secondary verb on text files.

```
[HKCR\txtfile\shell\scratch\DropTarget]
"Clsid"="{...}"
```

Wow, all we had to do was set two registry values and boom, we can now accept files via drop target. Multiselect a whole bunch of text files, right-click them, and then select "scratch". The shell sees that the verb is registered as a drop target, so it calls `CoCreateInstance` on the `CLSID` you provided. COM looks up the `CLSID` in the registry and finds the path to your program. Your program runs with the `-Embedding` flag, registers its class factory, and awaits further instructions. COM asks your class factory to create a drop target, which it returns to the shell. The shell then performs the simulated drop, and when you get the `IDropTarget::Drop`, your code springs into action and extracts all the files in the data object.

Now that we have all this working, it's just one more tiny step to register your application's drop target so that it is invoked when the user drops a group of files on the EXE itself (or on a shortcut to the EXE):

```
[HKLM\Software\Microsoft\Windows\CurrentVersion\App Paths\scratch.exe]
"DropTarget"="{...}"
```

With this additional registration, grab that bunch of files and drop it on the `scratch.exe` icon. Instead of passing all those files on the command line (and possibly overflowing the command line limit), the shell goes through the same procedure as it did with the context menu to hand the list of files to your program via the data object.

Nearly all of the work here was just managing the COM local server. The parts that had to do with the shell were actually quite small.

Raymond Chen

Follow