Access COM Clients From .NET Objects

Access COM clients from .NET components, program against the thread pool, and use the system's File Properties dialog.

by Juval Löwy and Karl E. Peterson

Technology Toolbox

- ❑ VB.NET
- ☑ C#
- ❑ SQL Server 2000
- ❑ ASP.NET
- ❑ XML
- ☑ VB6
- ☑ **Note:** Karl E. Peterson's solution also works with VB5.

Go Online!

Use these Locator+ codes at **www.visualstudiomagazine.com** to go directly to these related resources.

Download

VS0212QA Download the code for this article, including the ThreadPool project, which allows the user to queue requests for the thread pool; and a drop-in–ready module containing code that brings up any file's Properties dialog.

Discuss

VS0212QA_D Discuss this article in the .NET forum.

Read More

VS0212QA_T Read this article online.

VSEP011204RH_T "Boost Web Power With ASP.NET" by Rob Howard

VS0209BB_T Black Belt, "Sync Threads Automatically," by Juval Löwy

VS0205RT_T "Invoke Asynchronous Magic" by Robert Teixeira

Q: Use .NET Components With COM Clients

Does .NET use COM apartments? If not, why do I see the single-threaded apartment (STA) attribute on new Windows Forms applications?

A:

.NET does not have an equivalent to COM's apartments. Unlike COM, every .NET component resides in a free-threaded environment, and it's up to you to provide proper synchronization (see Figure 1). The question is, what threading model should .NET components present to COM when interoperating with COM components as a client? COM needs to take the client's threading model into account when deciding on the exact apartment of the COM server object. The Thread class has a property called ApartmentState of the enum type ApartmentState:

```
public enum ApartmentState
{
   STA,
   MTA,
   Unknown
}
```

By default, the Thread class's ApartmentState property is set to ApartmentState.Unknown. You can instruct .NET programmatically which apartment to present to COM. Simply set the value of the thread's ApartmentState property to either ApartmentState.STA or ApartmentState.MTA (but not to ApartmentState.Unknown):

```
Thread currentThread;
currentThread = Thread.CurrentThread;
currentThread.ApartmentState =
ApartmentState.STA;
```

You can even set the threading model before the thread starts to run:

```
//Some thread method
void ThreadMethod(){...}

ThreadStart threadStart;
threadStart = new
ThreadStart(ThreadMethod);
Thread workerThread = new
Thread(threadStart);
currentThread.ApartmentState =
ApartmentState.STA;
workerThread.Start();
```

You can also use either the STAThread or the MTAThread method attributes to set the apartment state declaratively. Although the compiler doesn't enforce that, you should only apply these attributes to the Main() method:

```
[STAThread]
static void Main()
{...}
```

Use programmatic settings for your worker threads. The Windows Forms application wizard applies the [STAThread] attribute automatically to the Main() method of a Windows Forms application. This in done for two reasons. First, you need the attribute applied in the event of the application hosting ActiveX controls, which are STA objects by definition. Second, you need the attribute applied for when the Windows Forms application interacts with the

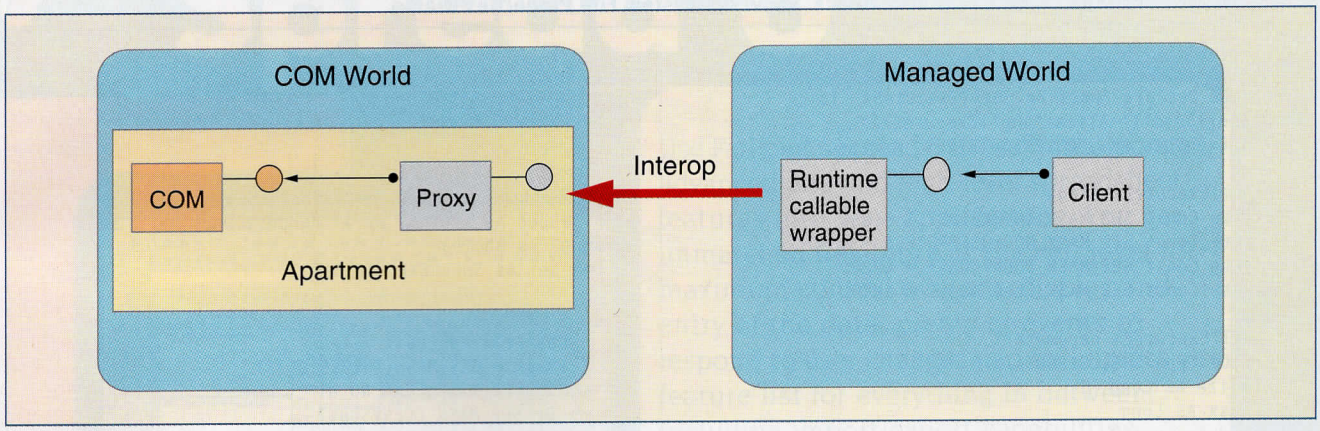

Figure 1 Masquerade COM Apartments. COM objects require apartments to manage concurrency and activation. However, .NET has no use for them. The interop layer converts a managed call to a COM call, and in the process, conveys a synthetic apartment model to COM that you can set, just as if your .NET client were a COM client.

clipboard, which still uses COM interop.

When you apply the [STAThread] attribute, the underlying physical thread uses OleInitialize() instead of CoInitializeEx() to set up the apartment model. OleInitialize() automatically does additional initialization required for enabling drag-and-drop.

You'll experience one side effect when you select an apartment-threading model: You cannot call WaitHandle.WaitAll() from a thread whose apartment state is set to ApartmentState.STA. If you do, .NET throws an exception of type NotSupportedException. The underlying implementation of WaitHandle.WaitAll() uses the Win32 call WaitForMultipleObjects(), and that call blocks the STA thread from pumping COM calls in the form of messages to the COM objects. —*J.L.*

Q: Use the System's File Properties Dialog

My application offers a display of filenames as part of its user interface. I'd like to give users the ability to bring up the system File Properties dialog as one of the options when they right-click on a filename. How can I do that?

A:

This task involves only a single quick call to the ShellExecuteEx API (see Listing 1). Unlike ShellExecute, ShellExecuteEx enables you to call any context menu item related to a given shell object—"properties," in this case.

Set up the call by creating a SHELLEXECUTEINFO structure and filling its first element (cbSize) with the structure's overall size so the system knows that you know what you're doing. Create the flag mask by combining the constant values for SEE_MASK_INVOKEIDLIST (which allows the use of shortcut menu extension verbs, rather than only Registry verbs), SEE_MASK_FLAG_NO_UI (which prevents the system from popping message boxes on errors), and SEE_MASK_DOENVSUBST (which expands environment variables in the passed filespec).

It's good form to stuff the structure's hWnd element with the handle of a window in your app. This window serves as the parent of any error-related message boxes the system pops up, although you've instructed the system already not to do this. The lpVerb element is the key to bringing up the desired dialog—assign "properties" to this element. Finally, assign the desired filename to the lpFile element, and make the call to ShellExecuteEx.

You can gauge your call's success by examining the SHELLEXECUTEINFO structure's hInstApp element for a value greater than 32. Values less than 32 indicate an error occurred, and you can interpret the exact cause from the specific value. —*K.E.P.*

Q: Program Against the Thread Pool

I read that .NET uses a thread pool under the hood for tasks such as asynchronous method calls. Is there a way I can program against that pool directly? This will save me the trouble of managing my own threads.

A:

You can program directly against the thread pool. Creating worker threads and managing their lifecycle gives you ultimate control over these threads. It also increases your application's overall complexity. If you only need to dispatch a unit of work to a worker thread, then you can take advantage of a .NET-provided thread from the thread pool instead of creating a thread. .NET manages the thread pool, and the pool has a set of threads ready to serve application requests. .NET makes extensive use of the thread pool itself, not only for asynchronous calls, but also for timers and remote calls. You access the .NET thread pool through the ThreadPool class's static methods. Using the thread pool is simple. First, create a delegate of type WaitCallback, targeting a method with a matching signature:

```
public delegate void WaitCallback
   (object state);
```

Then, provide the delegate to one of the ThreadPool class' static methods—typically, QueueUserWorkItem():

```
public sealed class Threading.ThreadPool
{
   public static bool
   QueueUserWorkItem(WaitCallback
   callBack);
   /* Other methods */
}
```

VB5, VB6 • Show the System File Properties Dialog

```
Option Explicit

Private Declare Function ShellExecuteEx Lib _
  "shell32.dll" Alias "ShellExecuteExA" _
  (lpExecInfo As SHELLEXECUTEINFO) As Long

' ShellExecuteEx flags
Private Const SEE_MASK_INVOKEIDLIST = &HC
Private Const SEE_MASK_NOCLOSEPROCESS = &H40
Private Const SEE_MASK_DOENVSUBST = &H200
Private Const SEE_MASK_FLAG_NO_UI = &H400

' ShellExecuteEx parameters
Private Type SHELLEXECUTEINFO
  cbSize As Long
  fMask As Long
  hWnd As Long
  lpVerb As String
  lpFile As String
  lpParameters As String
  lpDirectory As String
  nShow As Long
  hInstApp As Long
  '  Optional fields
  lpIDList As Long
  lpClass As String
  hkeyClass As Long
  dwHotKey As Long
  hIcon As Long
  hProcess As Long
End Type

Public Function ShowFilePropertiesDialog(ByVal _
  FileSpec As String, ByVal hWndMsgOwner As _
  Long) As Boolean
  Dim sei As SHELLEXECUTEINFO

  With sei
    .cbSize = Len(sei)
    .fMask = SEE_MASK_INVOKEIDLIST _
      Or SEE_MASK_FLAG_NO_UI _
      Or SEE_MASK_DOENVSUBST
    .hWnd = hWndMsgOwner
    .lpVerb = "properties"
    .lpFile = FileSpec
  End With
  Call ShellExecuteEx(sei)
  ' An "instance handle" is always greater than
  ' 32.  An error value from this call is always
  ' less than 32
  ShowFilePropertiesDialog = (sei.hInstApp > 32)
End Function
```

Listing 1 Enter this code into a standalone BAS module, and you have a drop-in–ready solution for displaying the system File Properties dialog. Simply pass the ShowFilePropertiesDialog procedure the filename you'd like to have displayed, and the handle for an owner form should the system decide it needs to display a message box (which is unlikely).

As the method name implies, dispatching a work unit to the thread pool is subject to pool limitations. This means that if no available threads exist in the pool, .NET queues the work unit and serves it only when a worker thread returns to the pool. .NET serves pending requests in order. Use the thread pool like this:

```
void ThreadPoolCallback(object state)
{
  Thread currentThread =
    Thread.CurrentThread;
  Debug.Assert(currentThread.
    IsThreadPoolThread);
  int threadID =
    currentThread.GetHashCode();
  Trace.WriteLine("Called on thread "
    "with ID :" +
    threadID.ToString());
}

WaitCallback callBack = new
  WaitCallback(ThreadPoolCallback);
ThreadPool.QueueUserWorkItem(callBack);
```

For diagnostic purposes, you can find out whether the thread your code runs on originated from the thread pool using the Thread class's IsThreadPoolThread property.

A second overloaded version of QueueUserWorkItem() allows you to pass in an identifier to the callback method in the form of a generic object:

```
public static bool QueueUserWorkItem(
WaitCallback callBack,object state);
```

You pass in the identifier as a single parameter to the callback method. If you don't provide such a parameter, .NET passes in null. The identifier enables the same callback method to handle multiple posted requests, while at the same time being able to distinguish between them.

The ThreadPool class supports several other useful ways of queuing a work unit. The RegisterWaitForSingleObject() method allows you to provide a waitable handle as a parameter. The thread from the thread pool waits on the handle, and only calls the callback once the handle is signaled. You can also specify a timeout to wait for. The GetAvailableThreads() method allows you to find out how many threads are available in the pool, and the GetMaxThreads() returns the pool's maximum size. —*J.L.*

Juval Löwy is a software architect and principal of IDesign, a consulting and training company focused on .NET design and migration. Juval is a Microsoft Regional Director for Silicon Valley, the author of *Programming .NET Components* (O'Reilly & Associates), and he speaks at software development conferences. Contact him at www.idesign.net.

Karl E. Peterson is a GIS analyst with a regional transportation planning agency and serves as a member of the *VSM* Technical Review and Editorial Advisory Boards. Online, he's a Microsoft MVP and a section leader on several DevX forums. Find more of Karl's VB samples at www.mvps.org/vb.

Additional Resources

"HOWTO: Set the COM Apartment Type in Managed Threads": http://support.microsoft.com/default.aspx?scid=kb;en-us;q318402&