Prepare for .NET 1.1 and Beyond

Maintain application compatibility and make your apps cope with multiple .NET versions.

by Juval Löwy

Technology Toolbox

- ☑ VB.NET
- ☑ C#
- ❑ SQL Server 2000
- ❑ ASP.NET
- ❑ XML
- ❑ VB6

.NET simplifies application deployment and versioning. Before .NET, you were probably often caught in DLL Hell: Deploying a new version of an application with its class libraries caused previous versions (and sometimes other applications) to stop functioning. The .NET architects solved this problem with a clear, deterministic set of ways an application binds to a particular version of an assembly it uses. However, if the other assemblies your app uses are the .NET assemblies, different versions of .NET itself can affect your apps. I'll give you some ground rules you should know and the actions you need to take to maintain application compatibility and, if possible, take advantage of new .NET versions (see the sidebar, ".NET Versions Include Security Policies").

.NET provides two kinds of assemblies—friendly-named assemblies and strongly named assemblies. Friendly-named class library assemblies typically reside in the directory of the application using them. Strongly named assemblies contain their producer's digital signature. The digital signature is the product of public and private encryption keys. Strongly named assemblies usually reside in the Global Assembly Cache (GAC). The GAC can contain multiple versions of the same class library assembly. Strongly named assemblies can also reside in the application directory. Each assembly has an assembly version number, provided as an assembly attribute, such as:

```
[VB.NET]
<Assembly: AssemblyVersion("1.2.3.4")>
```

When the MyApp assembly references another assembly—MyClassLibrary—that reference is recorded in the MyApp assembly's manifest. Which version of MyClassLibrary .NET

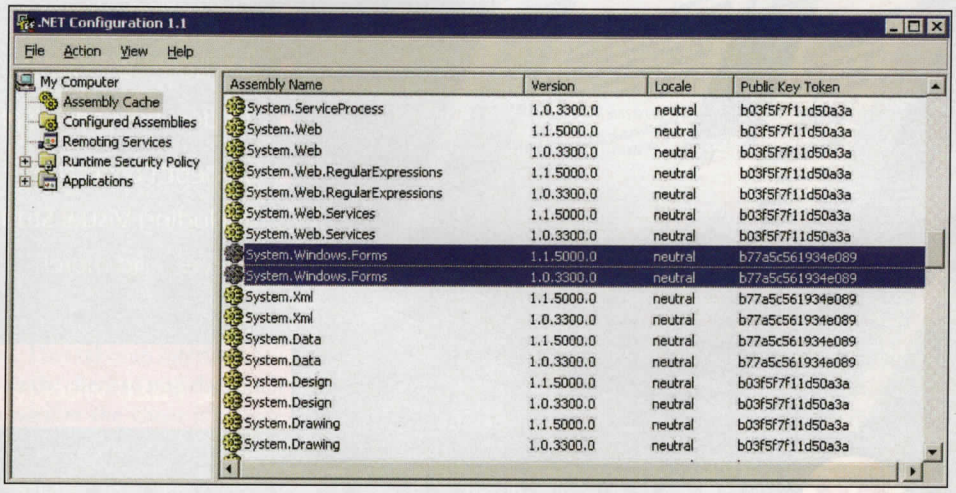

Figure 1 The GAC Supports Side-By-Side Execution. Every .NET version is deployed to the Global Assembly Cache (GAC), which can contain multiple .NET versions. Here the GAC contains both version 1.0 and version 1.1 of .NET, with version number 1.0.3300.0 and 1.1.5000.0, respectively, of the System.Windows.Forms class library assembly.

Go Online!

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

Discuss

VS0212DT_D Discuss this article in the .NET Framework/IDE forum.

Read More

VS0212DT_T Read this article online.

VS0206QA_T Q&A, "Develop Rich-UI Apps," by Karl E. Peterson and Juval Löwy

VS0203DT_T Desktop Developer, "Implement Versioning in .NET," by Bill Wagner

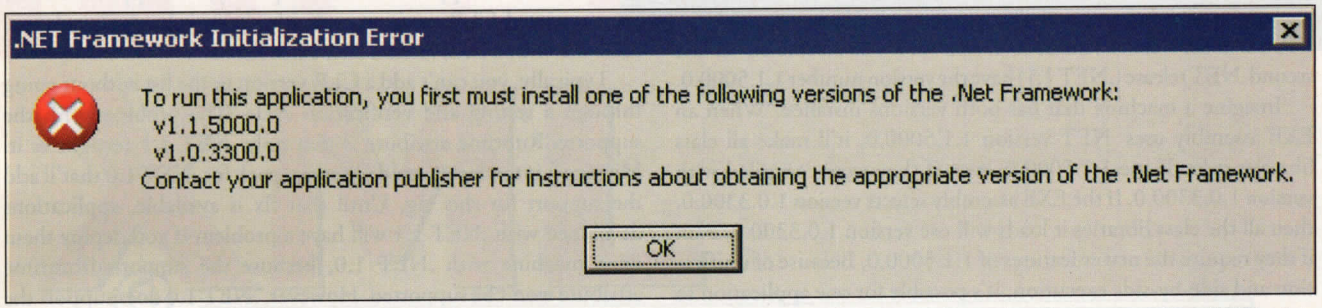

Figure 2 Install a Supported CLR Version. If an application specifies in its config file one or more CLR versions it supports, then .NET insists only these versions be used. If none of the specified versions is installed on the machine, .NET refuses to load the application and informs the user to install at least one supported CLR version.

uses depends on the kind of assembly MyClassLibrary is. If MyClassLibrary has a friendly name only, .NET will use whatever version of MyClassLibrary is found in the application directory, regardless of its version. .NET also supports custom assembly resolving options. In any case, when using friendly names only, application administrators can deploy new versions of MyClassLibrary without recompiling MyApp, as long as the new version of MyClassLibrary is backward compatible. .NET insists on a perfect version match if MyClassLibrary has a strong name. .NET looks in the GAC for a matching version of MyClassLibrary. If it finds a matching version, .NET loads it. If it finds no compatible version in the GAC, .NET looks in the application directory. Under no circumstances will .NET load a strongly named assembly with a different version number.

.NET's rigorous enforcement of version compatibility raises an interesting problem: If an application is built against version *n* of .NET, the application won't be able to take advantage of version *n*+1 of .NET (when it's available). The reason is that the application's manifest contains the version numbers of all assemblies it relies on, including the Common Language Runtime (CLR) and application frameworks. The .NET assemblies are strongly named, so the assembly resolver (the .NET entity responsible for finding and loading a compatible assembly) insists on a perfect version match. To overcome the issue of version compatibility with its own assemblies, .NET provides a different set of ground rules from those it uses with any other assembly. The issues involved are intricate. The exact CLR version that components in a class library or an EXE use can vary, depending on what they were compiled with, the available .NET versions, and the application-versioning policy.

Even though the CLR and the various .NET application frameworks consist of many assemblies, all are treated as a single versioning unit. Multiple versions of these units can coexist on any given machine. This is called CLR side-by-side execution.

Side-by-Side Execution Lets You Choose

Side-by-side execution is possible because .NET is deployed in the GAC, and the GAC supports side-by-side execution of different versions of the same assembly (see Figure 1). Because of CLR side-by-side execution, different .NET applications can use different versions of .NET simultaneously. It's also possible to install new versions of .NET or remove existing versions. Side-by-side coexistence reduces the likelihood of impacting one application when you install another because the old application can still use the older .NET version (provided you take certain steps I'll describe next). Nonetheless, CLR side-by-side execution allows you to choose when you'll upgrade to the next .NET version, rather than have the latest installed version ordain the upgrade.

When you choose to take advantage of features available in a newer .NET version but not in older ones, your components will no longer be compatible with older versions. As a result, you must test and certify your components against each .NET version and state clearly in your product documentation which .NET versions are supported. Next, you need to understand CLR *version unification*. All .NET applications are hosted in an unmanaged process that loads the CLR DLLs. That unmanaged process can use exactly one version of the CLR assemblies. Not only that, but when you get a particular version of the CLR, that also dictates which version of the .NET application frameworks is used, because both the CLR and the application framework's assemblies are treated as a single versioning unit. The fact that .NET always runs a unified stack of framework assemblies is called version unification.

Unification is required because the CLR and the .NET application frameworks aren't designed for mix and match, with some assemblies coming from version *n* and some from version *n*+1. A .NET application usually contains a single EXE application assembly, and potentially multiple class library assemblies. Unification means that in a process containing a managed application, the EXE application assembly, and the class libraries it loads, use the same .NET version. It's up to the EXE to select the CLR and application

.NET Versions Include Security Policies

Changes to the CLR version number aren't the only differences in new versions of .NET. Together with each version of the CLR assemblies, .NET also ships access security policy configuration files. These files contain the Enterprise, Machine, and User policies and are stored in version-specific directories. For example, the Enterprise policy file resides at <*Windows Directory*>\Microsoft.NET\Framework\<*Version*>\config\enterprisesec.config, and the Machine policy file at <*Windows Directory*>\Microsoft.NET\Framework\<*Version*>\config\security.config.

New versions of .NET might ship with different default security configurations. These security configuration changes mean that when you consider compatibility with a new .NET version, you must take its security policy into account. You need to decide whether that security policy is adequate, and if you think it isn't, you need to provide your own security policy.

framework version to use. The class libraries have no say in the matter. For example, all assemblies in the first .NET release (.NET 1.0) have the version number 1.0.3300.0. All assemblies in the second .NET release (.NET 1.1) have the version number 1.1.5000.0.

Imagine a machine that has both versions installed. When an EXE assembly uses .NET version 1.1.5000.0, it'll make all class libraries it loads use 1.1.5000.0, even if they were compiled with version 1.0.3300.0. If the EXE assembly selects version 1.0.3300.0, then all the class libraries it loads will use version 1.0.3300.0, even if they require the newer features of 1.1.5000.0. Because of unification and side-by-side execution, it's possible for one application to use version 1.0.3300.0 and another application to use 1.1.5000.0 at the same time, even if they interact with each other.

Any combination of CLR versions can reside on a given machine. Applications can rely implicitly on a default CLR version-resolution policy, or they can provide explicit configuration indicating the supported CLR versions.

Specify Supported CLR Versions

If the application doesn't indicate to .NET which CLR versions it requires, then the application is actually telling .NET that any compatible CLR version is allowed. In that case, .NET detects the CLR version the application was compiled with, and uses the latest compatible CLR version on the machine. To that end, .NET is aware of which CLR version is backward compatible with other versions. (Currently, all newer versions are backward compatible). The compatibility list is maintained in the Registry. Applications that rely on this default policy are typically mainstream applications that use the subset of types and services all the CLR versions support. Applications that take advantage of new features or types can't use the default policy, because they might be installed on machines with only older versions of the CLR. Similarly, applications that use features that are no longer supported can't use the default policy.

The default version might cause applications to run against CLR versions they weren't tested for, resulting in undetermined behavior. Applications that don't wish to rely on the default version binding policy, and would like to have deterministic behavior, can provide explicit version configuration. In such cases, the application must indicate in its configuration file which versions of the CLR it supports, using the startup tag with the supportedRuntime attribute:

```
<?xml version="1.0"?>
<configuration>
   <startup>
      <supportedRuntime version="v1.1.5000.0"/>
      <supportedRuntime version="v1.0.3300.0"/>
   </startup>
</configuration>
```

The order in which the CLR versions are listed indicates priority. .NET tries to provide the first CLR version to the application. If that version isn't available on the machine, .NET tries to use the next version down the list, and so on. If none of the specified versions is available, .NET refuses to load the application. .NET presents a message box, asking the user to install at least one of the supported versions specified in the configuration file (see Figure 2). Note that the startup directive overrides any default behavior .NET can provide, meaning that even if another compatible version is available on the machine (but not listed in the configuration file), .NET refuses to run the application. Consequently, if an application lists the supported CLR versions explicitly, it can't be deployed on a machine with a new version that's not listed.

Typically, you can't add a CLR version to the list without going through a testing and verification cycle. The problem with the supportedRuntime attribute is that only .NET 1.1 recognizes it. Microsoft intends to provide a service pack for .NET 1.0 that'll add the support for this tag. Until that fix is available, applications developed with .NET 1.1 will have a problem if you deploy them on a machine with .NET 1.0, because the supportedRuntime attribute won't be supported. However, .NET 1.0 does support the requiredRuntime attribute under the startup tag:

```
<startup>
   <requiredRuntime version="v1.0.3300.0"/>
</startup>
```

When the requiredRuntime attribute is specified, .NET uses the specified CLR version number instead of the one the EXE was built with. If the specified CLR version number isn't available on the machine, then .NET looks in the Registry for the newest available compatible version and uses that one. You (or a system administrator) can even instruct .NET not to look in the Registry by setting the safemode attribute to true:

```
<startup>
   <requiredRuntime version="v1.0.3300.0" safemode="true"/>
</startup>
```

In this case, .NET displays an error message and refuses to load the application if the required CLR version isn't available—even if a compatible version is available. The safemode's default value is false. Once .NET 1.0 supports the supportedRuntime attribute, then the requiredRuntime attribute will be considered deprecated and shouldn't be used.

The .NET architects tried to strike a balance between allowing innovation and new versions on one hand, and supporting existing applications on the other. Ultimately, it's up to you to decide whether you want your applications and components to support a particular CLR version. This marks a change of philosophy—Microsoft no longer guarantees absolute backward and forward compatibility, because that would be impractical. Instead, it pledges to make every effort to have newer .NET versions be backward compatible and to point out incompatibilities. VSM

Juval Löwy is a software architect and principal of IDesign, a .NET consulting and training company. He's also a Microsoft Regional Director, the .NET California Bay Area User Group Program committee chairman, and a conference speaker. This article derives from his upcoming book on .NET components (O'Reilly). Contact Juval at www.idesign.net.

Additional Resources

- *Programming .NET Components* by Juval Löwy [O'Reilly & Associates, 2002, ISBN: 0596003471]
- "End DLL Hell with .NET Version Control and Code Sharing" by Juval Löwy: www.devx.com/codemag/articles/2002/julyaug/enddllhell/codemag-1.asp

Q&A

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 is 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

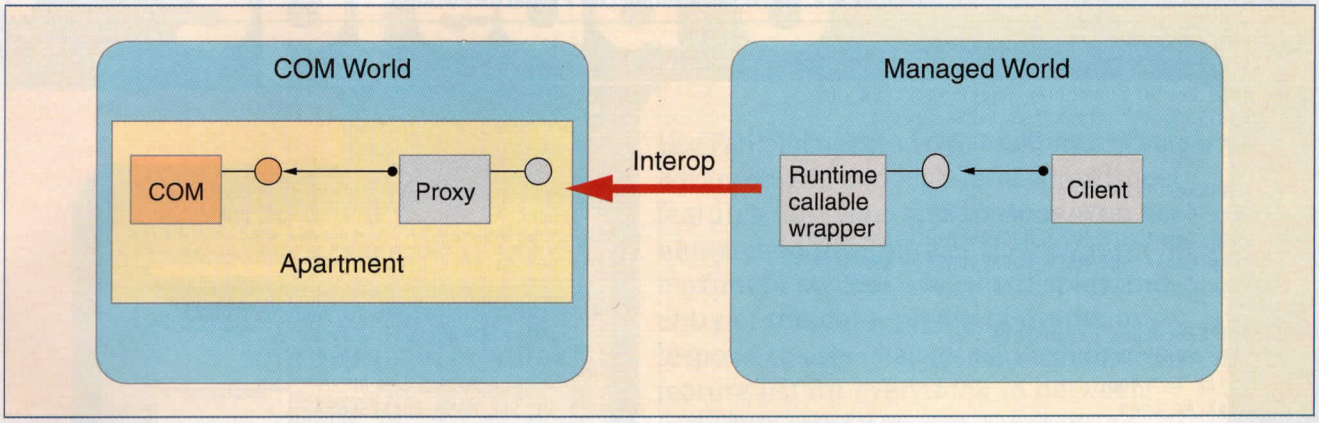

Figure 1 Masquerade COM Apartments. COM objects require apartments to manage concurrency and activation. However, .NET has no use for them. The interop layer converts a managed call to a COM call, and in the process, conveys a synthetic apartment model to COM that you can set, just as if your .NET client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