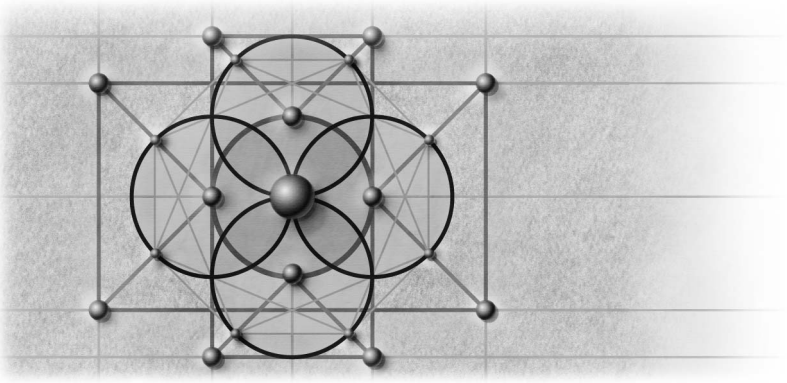

13

Upgrading ActiveX Controls and Components

If you're on old hand at Microsoft Visual Basic programming, you likely have vivid memories of the last great migration—the one between Visual Basic 3 and 4. It involved the great leap forward from 16 bits to 32 bits and the introduction of OLE components now known as ActiveX components.

The transition from Visual Basic 6 to Visual Basic .NET is reminiscent of those days, but on a much grander scale. Once again Microsoft is introducing a new component model to open doors to a whole new world of applications: Web applications and XML Web services. This new component model is the foundation for the .NET Framework.

This is all fine and wonderful, you might say, but are you kidding me? Do I need to update all of my ActiveX controls to .NET controls, just as I was required to replace all of my VBXs in Visual Basic 4 32 bit? Fortunately, the answer is no to both questions. You can still use your existing ActiveX components with Visual Basic .NET.

ActiveX Controls Are Still Supported—Yes!

One difference between the transition from Visual Basic 6 to .NET and that from Visual Basic 3 to 4 is that Visual Basic .NET still supports the old component model. This was not true of Visual Basic 4. Migrating your VBX components to ActiveX was an all-or-nothing proposition. If you wanted to continue to use VBXs, you had to keep your application in the 16-bit world. If the vendor for your favorite control did not have a 32-bit ActiveX control replacement for your favorite VBX control, you were out of luck. You had to either wait it out and

hope the vendor released a compatible control or use an ActiveX component provided by a different vendor. In any case, you were likely to run into issues when upgrading your application, whether the ActiveX component claimed full compatibility or not.

In Visual Basic .NET, you have the choice of using either an ActiveX component or an equivalent .NET component in its place. Unlike Visual Basic 4, which had a control migration feature that automatically upgraded your VBX controls to ActiveX controls, Visual Basic .NET has no such feature. Furthermore, Microsoft is not introducing any of the new .NET controls and components as direct replacements of similarly featured ActiveX controls. For example, even though there is a .NET TreeView control, it is not fully compatible with the ActiveX TreeView control. In most cases, you will need to make modifications to your code to use the new .NET controls. We'll discuss how to replace ActiveX controls with Windows Forms controls in Chapter 19. For now, let's concentrate on using ActiveX controls in Visual Basic .NET.

ActiveX Upgrade Strategy

Since one of the overriding goals of the Upgrade Wizard is to ensure compatible behavior after you upgrade your application to Visual Basic .NET, the Visual Basic team decided to allow your project to use the same ActiveX controls and components as before. There are a couple of exceptions to this rule, but in general your Visual Basic .NET application will run against the same set of ActiveX components after the upgrade. This strategy greatly increases the chances that your application will perform exactly as it did in Visual Basic 6. Once you are assured that everything is behaving the way you expect it to, you can replace ActiveX controls and components with equivalent .NET components as you see fit.

Limitations of ActiveX Control Hosting

If you have ever used an ActiveX control in an environment outside Visual Basic, such as Microsoft Internet Explorer or Microsoft Office, you might have encountered differences or limitations in behavior for that control. Likewise, a .NET Windows form is a unique control host environment with its own capabilities and limitations. Since a .NET Windows form is primarily the host container for .NET components, some interesting magic is cast on an ActiveX control to make it act as if it were a .NET component. Unfortunately, this magic has limited power and cannot be applied to all types of ActiveX controls. Therefore the Windows Forms environment offers no support for the following types of ActiveX controls and components:

- Container controls
- Windowless controls
- DAO-based data-bound controls
- Controls that make use of internal Visual Basic 6 interfaces
- Components that hook into the Visual Basic 6 extensibility model
- ActiveX designers

We discuss each of these in turn in the sections that follow.

Container Controls

The SSTab ActiveX control is an example of a container control. Although you can place an SSTab control on a Windows form, you cannot place other controls within the SSTab. Part of the problem is that the SSTab control needs to communicate with the controls placed inside it. The communication happens using ActiveX interfaces that are not supported by a .NET Windows form. The end result is that the SSTab cannot find the child controls. Moreover, it cannot associate each child control with the tab to which it belongs. The ability to display contained controls on separate tabs—an essential feature of the SSTab control—is lost.

> **Note** The Visual Basic .NET Upgrade Wizard automatically replaces each instance of the SSTab control with a .NET Windows TabControl. This is one of the few cases in which the Upgrade Wizard replaces an ActiveX control with an equivalent .NET control. Since the SSTab control is dead weight on a Windows form, the Visual Basic team decided to give you a head start. You will find that all of the child controls on each SSTab tab are moved to the appropriate TabControl tab. You are left to focus on changing top-level properties of the TabControl and tweaking some code. Otherwise, you would be spending hours meticulously re-creating and placing each child control on each TabControl tab exactly as it appeared in the SSTab control.

Windowless Controls

The Windows Forms environment provides support only for controls that display within a standard window. To support windowless controls, a host needs to support painting the control directly within the form window. One nice feature is that windowless controls can be made transparent, since the host manages the painting for the control and the form on the same window surface. A

side effect of Windows Forms not supporting windowless controls is that you cannot create transparent labels, as you could in Visual Basic 6.

In most cases the lack of support for windowless controls will not prohibit you from placing a windowless control on a Windows form. Most windowless controls have a split personality of being either windowed or windowless, depending on the capabilities of the host. If the host supports windowless controls, the control will take on its preferred windowless form. If the host does not support windowless controls, the control will take on its less desirable, but workable, windowed appearance. In the case of Windows Forms, all windowless controls will be asked to create a window and render as a windowed control.

DAO-Based Data-Bound Controls

DAO-based data-bound controls include the Visual Basic 5 DBGrid and RDO Data control. Visual Basic 6 links these controls to their data source using an internal data binding manager object. Visual Basic .NET contains no such internal data binding manager object and so provides no such support for DAO-based controls. Therefore, you will find that you cannot bind a DBGrid control to an RDO Data control in Visual Basic .NET.

ADO data binding works just fine. The data binding manager object for ADO is contained in an ActiveX component called *MSBind*. This component enables you to bind ADO components such as the Visual Basic 6 *DataGrid* to ADO data sources such as an upgraded *DataEnvironment* class.

Controls That Make Use of Internal Visual Basic 6 Interfaces

A limitation of the ActiveX architecture is that it does not provide a standard mechanism permitting controls to find one another on the same form or document. It is left up to the host to define interfaces that allow controls to hook up with one another. Visual Basic 6 defines interfaces such as *IVBGetControl* that allow ActiveX controls to find their parent, children, or siblings. Visual Basic .NET does not provide full support for these interfaces. As a result, controls that depend on other controls will not work properly. For example, the UpDown ActiveX control becomes more or less a paperweight in the Visual Basic .NET environment. You can attempt to buddy the control to another control, but you will find that the values for the buddy control will not update when you click the UpDown control. In this case, it is recommended that you use the .NET Framework UpDown control instead.

Components That Hook into the Visual Basic 6 Extensibility Model

Don't expect to be able to use a Visual Basic 6 add-in component in the Visual Basic .NET environment. Add-in components such as wizards generally interact with the development environment to create controls, change code, and update project settings. The component interacts with the environment through a set of

extensibility interfaces. The problem is that Visual Basic .NET supports a completely different set of extensibility interfaces than Visual Basic 6 does. Visual Basic 6 add-in components simply will not work in the new environment.

ActiveX Designers

ActiveX designers such as the DataEnvironment, WebClass, DHTML, and DataReport Writer are not supported in Visual Basic .NET. The Microsoft Visual Studio .NET environment supports a completely different approach to designers known as packages. None of these designers are provided as Visual Studio .NET packages. In the case of the DataEnvironment and WebClass designers, the Upgrade Wizard will upgrade your Visual Basic 6 DataEnvironment and WebClass projects to take advantage of run-time support classes for these designers. This means that your projects based on DataEnvironment and WebClass can be made to work at run time, but you will not get a fancy designer that you can use to change settings at design time. All settings must be changed by writing code.

ActiveX .NET Controls: Best of Both Worlds

When you place an ActiveX control on a Windows form, something interesting happens. The control becomes part ActiveX, part .NET—something we call an ActiveX .NET control. The ActiveX portion is the core control that defines the behavior of the control you know and love. The .NET part is a translation layer that is wrapped around the control to help make it fit in with other .NET controls and objects. This section describes how ActiveX controls are supported in .NET.

ActiveX Interop Ax Wrapper: The Windows Forms Wrapper

For Windows Forms to be able to position and display your ActiveX control, additional properties, events, and methods are added to the control. You can think of these as extended properties, methods, and events (PMEs) of your control. Figure 13-1 illustrates how an ActiveX control is extended to include properties such as *Location*, *Tag*, and *Visible*. These extended PMEs are combined with the public PMEs of your ActiveX control to form a new wrapper class. When you write code against the control, you are actually writing it against the wrapper class. The wrapper class in turn delegates to the ActiveX control. If you are setting a public property that is available on the ActiveX control, the wrapper class simply passes the property setting straight through to the control. If you are setting an extended property not found on the ActiveX control, the wrapper takes care of performing the operation on the control, such as toggling visibility or changing its size.

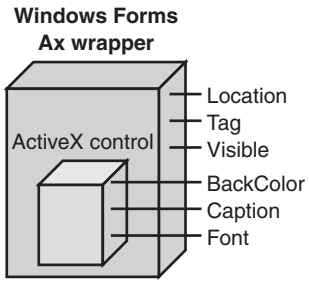

Figure 13-1 An Ax wrapper encapsulates and extends an ActiveX control.

The Ax (short for ActiveX) wrapper is officially known as the Windows Forms Wrapper (WFW). It is called this because the wrapper is specifically generated when the control is placed on a Windows form. If you place an ActiveX control on any other type of .NET form, such as a Web form, the Ax wrapper is not generated.

The concept of a wrapper class is not new. Visual Basic 6 wraps ActiveX controls in a similar fashion. However, the way controls are wrapped in the two environments is quite different. This leads to a number of interesting issues when the code is hosted on a Visual Basic .NET form.

> **Note** If you create a new Visual Basic .NET Windows application project and place an ActiveX control on the form, the control name is prefixed with *Ax*. For example, if you place an ActiveX TreeView control on the form, the resulting name of the control is AxTreeView. As the name implies, when you write code against AxTreeView you are actually programming against the Ax wrapper class for the control, not the control itself.

Property and Parameter Type Mappings

One purpose of the Windows Forms ActiveX control wrapper is to expose ActiveX properties in such a way that they can be naturally assigned to other .NET component properties. For example, in Visual Basic 6, the *Picture* property is based on the *StdPicture* object, which implements the *IPicture* and *IPictureDisp* interfaces. In Visual Basic .NET, however, the *Picture* property for .NET components is type *System.Drawing.Image*. The problem is that you cannot directly assign a property of type *StdPicture* to a *System.Drawing.Image*

property. To help alleviate this problem and avoid nasty-looking conversion code around every property assignment, the wrapper automatically exposes your ActiveX control's *Picture* property as *System.Drawing.Image*. Figure 13-2 illustrates how the Ax wrapper exposes an ActiveX property such as *Picture* as a .NET type such as *System.Drawing.Image*.

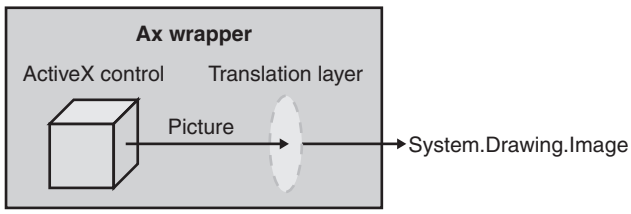

Figure 13-2 Picture property translated to *System.Drawing.Image*.

In a similar manner, the Windows Forms Ax wrapper exposes other common ActiveX properties, such as *Font* and *Color*, as the respective .NET type. Table 16-1 lists the ActiveX property types that the Windows Forms Ax wrapper exposes as the appropriate .NET type.

Table 16-1 Ax Wrapper Mapping of ActiveX Types to .NET Types

ActiveX Control Property Type	Maps to Ax Wrapper Property Type
OLE_COLOR	System.Drawing.Color
Font	System.Drawing.Font
Picture (bitmap)	System.Drawing.Image

Take, for example, the following Visual Basic 6 code, which assigns a *Picture* to the *Picture* property of a ListView ActiveX control:

```
ListView1.BackColor = vbRed
Set ListView1.Picture = _
   LoadPicture(Environ("WINDIR") & "\Prairie Wind.bmp")
Set Picture1.Picture = ListView1.Picture
```

The code is upgraded to the following Visual Basic .NET code:

```
ListView1.BackColor = System.Drawing.Color.Red
ListView1.Picture = _
   System.Drawing.Image.FromFile(Environ("WINDIR") & _
   "\Prairie Wind.bmp")
Picture1.Image = ListView1.Picture
```

Variant and *Object* Types: Mapping—Not!

As you can see, you can assign native .NET types such as *System.Drawing.Color.Red* or *System.Drawing.Image* to ActiveX control properties such as *BackColor* or *Picture*, respectively. This strategy works great when the ActiveX control property or method parameter is strongly typed. In this case Windows Forms sees that the property type for *BackColor* is *OLE_COLOR* and the type for *Picture* is *IPictureDisp*. Seeing these types, Windows Forms automatically maps the properties to their .NET equivalents when creating the wrapper. What happens when Windows Forms cannot tell the property type? For example, what happens when a property or method parameter is of type *Variant* or *Object*?

If an ActiveX control property or method parameter is not strongly typed—that is, if it is a generic type such as *Variant* or *Object*—the type is mapped to a .NET *Object* type in the wrapper. This commonly occurs when the ActiveX control exposes a property or a method that contains optional parameters. In order for a parameter to be optional it must also be of type *Variant*. If, for example, you are calling an ActiveX control method that takes an optional *Picture*, *Color*, or *Font*, you must explicitly pass the underlying ActiveX type as the parameter value. This means that you will need to convert a .NET type to the equivalent ActiveX type before making the call. Doing so can lead to some ugly-looking, but necessary, code.

Consider the following Visual Basic 6 code, which adds an image to an ActiveX ImageList control:

```
ImageList1.ListImages.Add , , _
   LoadPicture(Environ("WINDIR") & "\Zapotec.bmp")
```

If you right-click the line of code in Visual Basic 6 and choose Object Browser and then navigate to the *ListImages* type, you will see the following declaration:

```
Function Add([Index], [Key], [Picture]) As ListImage
```

Each parameter contained in square brackets is an optional parameter of type *Variant*. Although the third parameter is named *Picture*, it is of type *Variant*, so Visual Basic .NET does not know to map this property to *System.Drawing.Image*. If you upgrade this code to Visual Basic .NET, you end up with the following:

```
ImageList1.ListImages.Add( ,  , _
   VB6.ImageToIPictureDisp(System.Drawing.Image.FromFile( _
   Environ("WINDIR") & "\Zapotec.bmp")))
```

If the *Picture* parameter had been treated as a *System.Drawing.Image* type, the call to *System.Drawing.Image.FromFile* would be sufficient. In this case, however, the *Picture* parameter is mapped to *Object* and requires the underlying

ActiveX type *IPictureDisp*. The compatibility library function *ImageToIPictureDisp* is required to convert the *Image* object to an *IPictureDisp*—the underlying interface type for *Picture*—and the resulting *IPictureDisp* is passed as the *Picture* argument to the *Add* method. Without this conversion you will encounter a run-time COMException returned from the ImageList control, containing the error "Invalid picture."

Fortunately, as in the case just described, the Upgrade Wizard handles most of these conversions for you. In rare cases you will need to call a conversion helper function to convert a .NET type to the equivalent ActiveX type.

Standard Component Wrappers and ActiveX Control Subobjects

ActiveX controls that contain one or more subobjects have a split personality. The top-level set of control properties, methods, and events are wrapped and handled by an Ax wrapper. A simple component wrapper wraps all subobjects and other types defined by the control. A simple component wrapper is also applied to all standard ActiveX components that are not ActiveX controls. For example, if you reference ADO, a simple component wrapper is applied to expose the properties, methods, and events of all ADO objects. The objects are exposed using the .NET types that most closely represent the underlying type contained in the ActiveX component.

In the case of the TreeView control, the Ax wrapper is applied to the TreeView's top-level properties, methods, and events. Properties such as *Appearance*, *Font*, and *Style* are wrapped and handled by the Ax wrapper. The Ax wrapper exposes the *Font* property as a *System.Drawing.Font*, for example. If TreeView had a *BackColor* property, it would be exposed as a *System.Drawing.Color* property.

Now let's take a look at subobjects that the TreeView control exposes, such as *Nodes* and *Node*. The *BackColor* property of a *Node* gets exposed to you as a *System.UInt32* type. But isn't *BackColor* type *OLE_COLOR*? Why *UInt32*? *UInt32* is chosen for two reasons:

- The type most closely matches the type *OLE_COLOR*, which itself is a *UInt32*.
- Neither the simple component wrapper nor the Ax wrapper supports type aliases. In the original COM type library, *OLE_COLOR* is an alias for *UInt32*.

Consider the following Visual Basic 6 example, which sets the *ForeColor* property on both the ListView ActiveX control and one of its *ListItem* subobjects:

```
Dim li As ListItem
Set li = ListView1.ListItems.Add(, "Item1Key", "Item1")
ListView1.ForeColor = vbRed
li.ForeColor = vbRed
```

After upgrade, the Visual Basic .NET code is as follows:

```
ListView1.ForeColor = System.Drawing.Color.Red
li.ForeColor = _
   System.Convert.ToUInt32(System.Drawing.ColorTranslator.ToOle( _
   System.Drawing.Color.Red))
```

Although *ForeColor* is the same exact type for ListView and the *ListItem* object, the code needed to assign the value of Red is radically different. The ListView *ForeColor* property is exposed by the Ax wrapper as type *System.Drawing.Color*, so it's quite natural to assign a *System.Drawing.Color.Red* object to the property. The ListView Ax wrapper takes care of translating the *System.Drawing.Color.Red* object value to the equivalent *UInt32* color value.

In the case of the *ListItem.ForeColor* property, the simple component wrapper exposes the type as *System.UInt32*, leaving it up to you to figure out how to convert a *System.Drawing.Color.Red* object value to a numeric *UInt32* color value. To make this conversion, you first need to convert the color object to a numeric value by using *System.Drawing.ColorTranslator.ToOle*. The *ToOle* function returns a signed long integer or *System.Int32*. You then need to convert the *Int32* value to a *UInt32* value by using the *System.Convert* class.

Fortunately, the Upgrade Wizard handles most of these conversions for you. The downside is that you may end up with some rather unfamiliar-looking conversion code. Table 16-2 provides a list of conversion helper functions so that you'll understand how the conversion works when you see one of these in code. You can also use these functions when writing new code to help make assignments between ActiveX and .NET types.

Table 16-2 Useful Conversion Functions for Common ActiveX Object Types

Conversion Function(s)	Description
System.Convert methods	Allow you to convert from virtually any .NET type to another .NET type. Use *System.Convert* to convert 32-bit signed integers (*System.Int32*) to 32-bit unsigned integers (*System.UInt32*).
System.Drawing.ColorTranslator methods	Allow you to convert between OLE color, Windows color, and .NET color types.
FontToIFont, *IFontToFont*	Located in the *Microsoft.VisualBasic.Compatibility.VB6.Support* class. You use these functions to convert between .NET and ActiveX *Font* types. They are used primarily when you are trying to get or set an ActiveX component property that is exposed as *Object*, *IFont*, *IFontDisp*, or as a component-specific class that implements *IFont*.
CursorToIPicture, *IconToIPicture*	Allow you to convert a *System.Windows.Forms.Cursor* or *System.Drawing.Icon* type directly to an ActiveX *IPicture*.
ImageToIPicture, *ImageToIPictureDisp*, *IPictureToImage*, *IPictureDispToImage*	Allow you to convert between the .NET *System.Drawing.Image* and ActiveX *IPicture* types. These functions are intended for situations in which you are trying to get or set an ActiveX component *Picture* property that is defined as *Object*, *IPicture*, *IPictureDisp*, or as a component-specific class declaration that implements *IPicture*.

Common Exceptions That Require Type Conversions

Certain lines of your upgraded code may contain type mismatch assignments. For example, you may find code in which the *MousePointer* property of an ActiveX control is being assigned to a *System.Windows.Forms.Cursors* property.

The problem is that an ActiveX *MousePointer* property is usually a *MousePointerConstants* enumeration type defined by the control—in other words, a numeric type. Each *System.Windows.Forms.Cursors* property, such as *IBeam*, returns a *Cursor* object. Attempting to assign a *Cursor* object to a numeric type is a recipe for a compiler error. In this case, you receive a descriptive compiler error telling you that you cannot assign a *Cursor* to a numeric type.

Compiler errors are not the type of errors that you need to worry about. Although annoying at times, they are in-your-face error messages that point directly to a problem in your code. You can easily locate the problem and, in most cases, especially with the help of IntelliSense, find a quick fix. A more insidious problem that will give you fits is an exception that occurs at run time. The two most common exceptions that relate to the assignment of incompatible types at run time are InvalidCastException and COMException.

InvalidCastException

The InvalidCastException is a standard .NET exception that occurs when the .NET Framework is unable to cast one type to another at run time. This exception commonly occurs when you attempt to assign a .NET type to an ActiveX type. For example, if you attempt to assign a .NET collection to an ActiveX component method that returns a Visual Basic 6 *Collection* object, this exception will occur.

COMException

A COMException can occur any time a property or method of an ActiveX control or component is called. The exception generally occurs because the Visual Basic .NET code is passing a .NET object when it should be passing an ActiveX object. For example, if you attempt to assign an Ax wrapped ImageList control to the *ImageList* property of an Ax wrapped TreeView control, as in the following line of code, it will bark back at run time with a COMException.

```
AxTreeView1.ImageList = AxImageList1
```

The problem happens in this case because you need to assign the underlying ActiveX control object to the *ImageList* property. You can obtain the underlying ActiveX control object by calling *GetOcx* on the Ax wrapped control. The following code works and does not bark at you:

```
AxTreeView1.ImageList = AxImageList1.GetOcx()
```

Name Collisions

Name collisions happen when the Ax wrapper class includes a property, method, or event that has the same name as the ActiveX control. Take, for example, an ActiveX control that has a property called *Size*. The ActiveX control's *Size* property will be in conflict with the *Size* property that the Ax wrapper adds to the control. To manage this type of conflict, the Ax wrapper will rename the ActiveX control property to *CtlSize*. If you need to set the ActiveX control's size property, you should set *CtlSize*. To set the size property managed by Windows Forms on behalf of the ActiveX control, you would set the *Size* property. Fortunately, name conflicts will not occur for most ActiveX controls. When the Upgrade Wizard detects a name conflict, it will upgrade your code to use the correct property or method even if the property or method has been renamed.

Naming conflicts are not new to Visual Basic .NET. Visual Basic 6 also manages name conflicts for you. As we mentioned earlier in this chapter, Visual Basic 6 creates a wrapper class for ActiveX controls. If there is a conflict with a property or method name, the wrapper hides the ActiveX control property or method in favor of the wrapper's property or method. To access the underlying ActiveX property, you need to use the ActiveX control's *Object* property. For example, the following Visual Basic 6 code:

```
MyFavoriteControl.Object.Tag = "My control's internal tag"
MyFavoriteControl.Tag = "The Tag property given to me by Visual Basic"
```

upgrades to the following Visual Basic .NET code:

```
MyFavoriteControl.CtlTag = "My control's internal tag"
MyFavoriteControl.Tag = "The Tag property given to me by Visual Basic"
```

Event Name Collisions

If an event name conflicts with another property or base class event name, the Upgrade Wizard renames the event. It does so by appending the word *Event* to the end of the event name. One ActiveX control that exhibits this behavior is the Microsoft WinSock control. If you place a WinSock control on a Windows form and view its events, you will see that the *Close* and *Connect* events have been renamed *CloseEvent* and *ConnectEvent*, respectively. Since most controls contain unique event names, you will seldom encounter this issue. Even if you do, you probably will not notice. If a name conflict does occur, the renamed event is easy to find, since the event name is mostly preserved. The Upgrade Wizard automatically upgrades your code to use the renamed event name, so you should not encounter any issues after upgrading your Visual Basic 6 application.

Using ActiveX Components from .NET

This section discusses common issues that you may encounter when using an ActiveX control or component in your Visual Basic application after upgrading to Visual Basic .NET.

When *ByRef* Bites

The Visual Basic .NET compiler introduces a new semantic when passing properties *ByRef* to a function. If the property has write access, it is set to the return value of the *ByRef* parameter. In Visual Basic 6 this is not the case. If you pass a property *ByRef*, the property is never set.

As an example, consider the following Visual Basic 6 code contained in Form1 of a default standard EXE project:

```
Private Sub Form_Load()
   SetString Me.Caption
   MsgBox Me.Caption
End Sub

Private Sub SetString(ByRef str As String)
   str = "String changed by SetString"
End Sub
```

When this code is run, the message box displays "Form1." *Me.Caption* is not changed by calling the function *SetString*.

Consider the following code that has been upgraded to Visual Basic .NET:

```
Private Sub Form1_Load(ByVal eventSender As System.Object, _
                       ByVal eventArgs As System.EventArgs) _
                       Handles MyBase.Load
   SetString(Me.Text)
   MsgBox(Me.Text)
End Sub

Private Sub SetString(ByRef str_Renamed As String)
   str_Renamed = "String changed by SetString"
End Sub
```

If you run the Visual Basic .NET version, you will see that the *Text* property of the form gets set to "String changed by SetString" as a result of calling *SetString*.

This is both good and bad. It's good that Visual Basic .NET offers this new semantic so that properties passed *ByRef* are set the way you would expect. It's bad, however, in that this change could lead to subtle, hard-to-find problems in your upgraded code.

This issue does not end with your own code. You may run into it when attempting to call an ActiveX component that takes *ByRef* parameters. Consider the following Visual Basic .NET code written against a ListView control located on Form1 in a default Windows application project:

```
Dim lvItem As MSComctlLib.ListItem
AxListView1.ListItems.Add(, , "Item1")
lvItem = AxListView1.ListItems(AxListView1.ListItems.Count)
```

Run this code and you will encounter a COMException on the last line, telling you that "Property is read-only." What property is read-only? you might ask. How am I going to figure this one out?

The problem in this case is that the *ListItems* default property *Item* takes a *ByRef Variant* argument. *AxListView1.ListItems.Count* is passed as an argument to this *ByRef* parameter. Shouldn't this be okay? Isn't the *Count* property read-only, so the compiler won't try to set it? Yes and no. The *Count* property is implemented with both a *Get* and a *Let*. Therefore, the compiler thinks the property can be written. When it tries to write to the property, however, the *Let* implementation for the *Count* property throws an exception.

The reason the ListView *ListItems* subobject has a settable *Count* property is a long, twisted story stretching back three versions of Visual Basic. To work around these types of issues, you can include parameters that you don't want to be set in parentheses. Using parentheses tells the compiler to pass the property read-only. For example, if you change the code in the previous example to the following, it will work. Note the extra parentheses around *AxListView1.ListItems.Count*:

```
' Pass AxListView1.ListItems.Count ByVal to avoid Set being called
lvItem = AxListView1.ListItems((AxListView1.ListItems.Count))
```

When a Collection Is Not a Collection

Just because a .NET collection looks like a collection does not mean that it really is a collection—at least not in the Visual Basic 6 sense. A .NET collection is almost the same as a Visual Basic 6 collection. It has the same methods, such as *Add*, *Remove*, and *Item*, but you cannot assign a .NET collection to a Visual Basic 6 collection. Why would you need to use a Visual Basic 6 collection? Isn't this .NET? Don't you want all of your types to be declared in .NET? Isn't the .NET collection a new and improved *Collection* object? Yes on all counts, but there is one case in which you will need to use a Visual Basic 6 collection: when you call a COM object that takes a Visual Basic 6 *Collection* object as a

parameter or that returns a Visual Basic 6 *Collection* object. A .NET collection will not do in this case.

Take, for example, the following Visual Basic 6 code declared in a public class module of an ActiveX DLL called RetCollection.dll:

```
Option Explicit
Private m_Collection As New Collection
Public Function ReturnCollection() As Collection
   Set ReturnCollection = m_Collection
End Function

Private Sub Class_Initialize()
   m_Collection.Add "MyItem1", "MyKey1"
   m_Collection.Add "MyItem2", "MyKey2"
End Sub
```

Now consider the following Visual Basic .NET code, which calls the *ReturnCollection* function:

```
Dim c As Collection
Dim rc As New RetCollectionLib.RetCollection()

c = rc.ReturnCollection
MsgBox(c.Item(1))
```

The code compiles, runs—up to a point, at least—and throws an InvalidCastException on the attempt to assign *c* to *rc.ReturnCollection*. The problem is that a .NET collection—represented by the variable *c*—cannot be assigned to the Visual Basic 6 *Collection* object returned by the function *ReturnCollection*.

To fix this problem, you need to add a reference to the Visual Basic 6 runtime Msvbvm60.dll. Follow these steps to do so:

1. Right-click the References list in Solution Explorer, and choose Add Reference.
2. Select the COM tab.
3. Select Visual Basic For Applications Version 6.0 from the list.
4. Change your Visual Basic .NET code to use the Visual Basic 6 *Collection* object as follows:

```
Dim c As VBA.Collection
```

The variable *c* now matches the type returned by the *ReturnCollection* function, so everything will now work as expected.

Nonzero-Bound Arrays

You may encounter problems when attempting to call a COM object method that returns an array defined with a nonzero-bound lower dimension, such as –10 or 1. Take, for example, the following code defined in a Visual Basic 6 ActiveX server DLL:

```
Option Base 1
Public Function RetStringArray() As String()
   Dim i As Long
   'Array is 1-based. We're using Option Base 1
   Dim s(10) As String

   For i = LBound(s) To UBound(s)
      s(i) = i
   Next

   RetStringArray = s
End Function
```

Suppose that you are trying to call the *RetStringArray* function, using the following Visual Basic .NET code:

```
Dim rs As New RetStringArrayLib.RetStringArray
Dim s() As String

s = rs.RetStringArray
MsgBox(s(1))
```

The code will compile, but you will encounter an InvalidCastException at run time when trying to assign *s* the return value of *rs.RetStringArray*. The problem occurs because you are attempting to assign a nonzero-based string array to a zero-based string array variable. To fix this, you need to change the declaration of *s* from a strongly typed *String* array—always zero based—to a generic *System.Array* type, as follows:

```
Dim s As System.Array
```

The generic *System.Array* type can represent a nonzero-based array but cannot be assigned to a strongly typed array unless it is zero bound.

Alias Types Are Not Supported

Type aliasing is declaring a new type based on an existing type. A common example is the *OLE_* types defined in the Standard OLE Automation type library (StdOle2.tlb). The type *OLE_COLOR*, for example, is an alias for an unsigned 32-bit long integer. When you reference a COM object in .NET, type aliases are

lost. Instead you need to use the base type. For example, you need to use the .NET *System.UInt32* type instead of *OLE_COLOR*.

> **Note** Neither Visual Basic 6 nor Visual Basic .NET allows you to declare a type that is an alias of another type. In the case of Visual Basic 6, however, you can use aliased types from other type libraries. For example, you can create a Visual Basic 6 component with a public *BackColor* property of type *OLE_COLOR*. If you attempt to use the component in .NET, the property type will show up as *System.UInt32*.

Module Methods Are Not Supported

If you have a Visual Basic 6 project that references a COM component that exports module methods, the module methods are not available to be called by a .NET client. For example, the DirectX Visual Basic type library Dx8vb.dll contains a large number of module methods that can be called from Visual Basic 6. When you upgrade Visual Basic 6 code that calls module methods, the calls are left in the code, but the code doesn't compile.

Since module methods delegate to exported functions within a DLL, the trick to solving this problem is finding the DLL-exported function that the module method calls. To do this, you can dump the list of exported functions for the DLL, but first you need to find the DLL containing the module method you want to call. The easiest way to find it is to load the original Visual Basic 6 project in Visual Basic 6 and find the DLL reference within the References list. To view the References list, choose References from the Project menu. Once you have located the reference within the list—for example, DirectX 8 For Visual Basic Type Library—note the location of the DLL for that reference. Open a DOS command window and dump the exported functions contained in the DLL by executing the following command:

```
DumpBin /Exports DX8VB.DLL > Exports.Txt
```

Launch Notepad and open Exports.txt. Look for a function that matches the name of the module method that your Visual Basic 6 code calls. For example, if your Visual Basic 6 code calls the DirectX function *D3DXColorAdd*, you will find the following entry in the DLL exports list:

```
105   15 0002DB8F VB_D3DXColorAdd
```

You can use this information to declare the API entry point in your Visual Basic code. The most useful information in this cryptic entry is the entry ID contained

in the first column and the name of the function. In this case the entry ID is 105 and the function name is *VB_D3DXColorAdd*.

Suppose, for example, that you have upgraded a Visual Basic 6 function that calls the DirectX function *D3DXColorAdd* and you end up with the following Visual Basic .NET code:

```
Dim COut As DxVBLibA.D3DCOLORVALUE
Dim CRed As DxVBLibA.D3DCOLORVALUE
Dim CBlue As DxVBLibA.D3DCOLORVALUE

CRed.r = 255
CBlue.b = 255
'UPGRADE_ISSUE: COM expression not supported: Module methods of
'COM objects. Click for more: 'ms-help://MS.VSCC/commoner/redir/
'redirect.htm?keyword="vbup1060"'
DxVBLibA.D3DXMATH_COLOR.D3DXColorAdd(COut, CRed, CBlue)
```

This code leads to a compiler error, since *D3DXColorAdd*, as suggested by the UPGRADE_ISSUE comment, is not available. Based on the Exports.txt DLL export information obtained earlier, you can declare the function as follows:

```
Private Declare Function D3DXColorAdd Lib "dx8vb.dll" _
   Alias "#105" (ByRef COut As DxVBLibA.D3DCOLORVALUE, _
   ByRef C1 As DxVBLibA.D3DCOLORVALUE, ByRef c2 As _
   DxVBLibA.D3DCOLORVALUE) As Integer
```

In this case we're using the entry ID 105 as the name of the function. You could also declare the function to use the API function name *VB_D3DXColorAdd* in the *Alias* clause. To obtain the full declaration for the function, you can load your original Visual Basic 6 project in Visual Basic 6 and start the Object Browser. With the Object Browser running, you can search for the module method declaration for *D3DXColorAdd*. The module method declaration will match the API declaration you need to create using the *Declare* statement.

Conclusion

The good news is that ActiveX controls and components are supported in Visual Basic .NET. Although there are some limitations in the support that is provided, you should not be hindered in upgrading Visual Basic 6 applications that are good candidates for Visual Basic .NET. For example, if you have a

Visual Basic 6 middle-tier component that makes heavy use of ActiveX-based ADO objects, you can upgrade your component to a Visual Basic .NET class library component. All of your code that talks to the ActiveX ADO components will continue to work as is. You can then quickly take advantage of Visual Basic .NET by exposing some of your methods as WebMethods to be called over the Internet, for example.

If Visual Basic .NET did not allow you to make use of your existing ActiveX components, you would be devoting all of your energy to finding or creating .NET replacements for everything you are using. You would get bogged down just trying to get your application working again in the .NET environment. The ability to use ActiveX components frees you to focus only on those parts of your application that are critical to get working in .NET. This ability enables you to get up and running in .NET quickly.