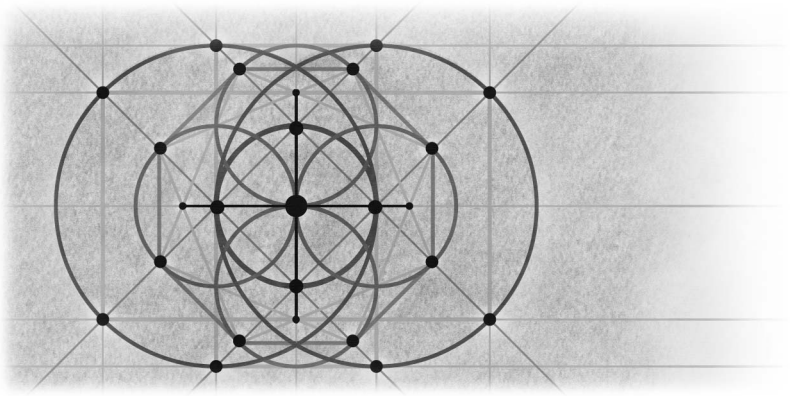

2

Visual Basic 6 and Visual Basic .NET: Differences

More than three years ago, the Microsoft Visual Basic team set out to create Visual Basic .NET. At that time managers would kid the development team by saying that they were making only three "simple" changes to Visual Basic 6: a new runtime system, a new development environment, and a new compiler. The Visual Basic development team spent the next three years working on one of these changes: the new compiler. Two other teams provided the development environment and runtime. As we pointed out in Chapter 1, the end result is not a new version of Visual Basic 6 but an entirely new product: *Microsoft Visual Basic .NET*. The name is important for two reasons. First, Visual Basic is still Visual Basic. Second, Visual Basic .NET is not Visual Basic 7.

This chapter describes the three "simple" changes made to create Visual Basic .NET, including changes to the runtime, the development environment, and the compiler. Microsoft also added other features to Visual Basic .NET along the way, including a new forms package and a new debugger, and these are also discussed in this chapter.

.NET Framework vs. ActiveX

As a Visual Basic developer, you will normally not be concerned with the runtime systems that underlie your Visual Basic applications. Visual Basic 6, for example, makes the details of how ActiveX works largely transparent. The Visual Basic 6 runtime handles all of the messy details that come with implementing an ActiveX-compliant component or application. Licensing, persistable objects, Microsoft Transaction Server (MTS) transaction awareness, and binary

compatibility are exposed as simple settings that you can turn on or off. In the same vein, Visual Basic .NET does a good job of hiding the details of what happens under the hood. For example, you do not need to know that you are creating or using a .NET component. A .NET component is just like any other component. It has properties, methods, and events just as an ActiveX component does. Why should you care about the differences between ActiveX and .NET if everything basically looks the same?

On the surface, it doesn't matter whether you're using ActiveX, .NET, or your best friend's component model—they all look about the same. When you dig into the details, however, you need to understand the machine that lies beneath.

If you have ever created an ActiveX control in Visual Basic 6, you may have found that it behaves slightly differently from other ActiveX controls that you bought off the shelf. For example, if you add a *BackColor* property to your control, you'll notice when you test it that the color picker is not associated with your control. Digging deeper, you'll find that you need to change the type of the property to *OLE_COLOR* and set the *Property ID* attribute on the property to *BackColor*. Only then will the property behave like a *BackColor* property. In solving this problem, you needed to cross over from pure Visual Basic into the world of ActiveX. Although Visual Basic attaches different terminology to options and language statements, you end up being directly or indirectly exposed to ActiveX concepts such as dispatch IDs (DISPIDs), what Visual Basic refers to as property IDs, and OLE types such as *OLE_COLOR*. Visual Basic, as much as it tries, cannot hide this from you. The more properties, events, methods, and property pages you add to your Visual Basic 6 ActiveX control, the more problems you encounter that require an ActiveX-related solution.

Visual Basic .NET works in much the same way. Most of the time, you are just dealing with Visual Basic. However, when you need your application or component to behave consistently with other types of applications, whether they be standard Windows applications or Web service server objects, you will need a detailed understanding of the environment in which you want your application to run. In the case of .NET applications, you will need to understand how .NET works. The more you know about the target environment, the better equipped you are to create a component or application that behaves well in that environment. So let's dig a bit and uncover the machine that will run your upgraded Visual Basic .NET application: the .NET Framework.

.NET Framework

The .NET Framework is composed of two general parts: the common language runtime and the Framework class library. The runtime is the foundation upon which the .NET Framework is based. It provides the basic services on which all .NET applications depend: code execution, memory management, thread management, and code security. The Framework class library provides building blocks for creating a variety of .NET applications, components, and services. For example, the Framework class library contains the base classes for creating ASP.NET Web applications, ASP.NET XML Web services, and Windows Forms. It defines all of the value types, known as *System* types, such as *Byte*, *Integer*, *Long*, and *String*. It gives you complex structure classes such as *Collection* and *HashTable*, as well as the interfaces such as *ICollection* and *IDictionary* so you can define your own custom implementation of a standard *Collection* or *HashTable* class.

The .NET Framework as a whole, since it works across all .NET languages, can be thought of as an expanded version of the Visual Basic 6 runtime. The common language runtime corresponds to the Visual Basic Language Runtime in Visual Basic 6, which includes the byte code interpreter and memory manager. The counterparts of the .NET Framework class library in Visual Basic 6 include the Visual Basic forms package, the *Collection* object, and global objects such as *App*, *Screen*, *Printer*, and *Clipboard*.

The main difference between the two environments is that Visual Basic 6 is a closed environment, meaning that none of the intrinsic Visual Basic types, such as *Collection*, *App*, *Screen*, and so on, can be shared with other language environments, such as C++. Likewise, Microsoft Visual C++ is largely a self-contained language environment that includes its own runtime and class libraries, such as MFC and ATL. The MFC *CString* class, for example, is contained within the MFC runtime and is not shared with other environments such as Visual Basic.

In closed environments such as these, you can share components between environments only when you create them as ActiveX components, and even then there are a number of limitations. ActiveX components need to be designed and tested to work in each target environment. For example, an ActiveX control hosted on a Visual Basic 6 form may work wonderfully, but the same control may not work at all when hosted on an MFC window. You then need to add or modify the interfaces or implementation of your ActiveX component to make it work with both the Visual Basic 6 and MFC environments. As a result, you end up duplicating your effort by writing specialized routines to make your ActiveX component work in all target environments.

The .NET Framework eliminates this duplication by creating an environment in which all languages have equal access to the same broad set of .NET types, base classes, and services. Each language built on the .NET Framework

shares this common base. No matter what your language of choice is—Visual Basic .NET , C#, or COBOL (for .NET)—the compiler for that language generates exactly the same set of .NET runtime instructions, called Microsoft Intermediate Language (MSIL). With each language distilled down to one base instruction set (MSIL), running against the same runtime (the .NET common language runtime), and using one set of .NET Framework classes, sharing and consistency become the norm. The .NET components you create using any .NET language work together seamlessly without any additional effort on your part.

Now that you have seen some of the differences between the Visual Basic 6 ActiveX-based environment and the Visual Basic .NET environment, let's focus on various elements of the .NET Framework and see how each element manifests itself in Visual Basic .NET. The elements we will be looking at are memory management, type identity, and the threading model. Each of these areas will have a profound impact on the way you both create new Visual Basic .NET applications and revise upgraded Visual Basic 6 applications to work with Visual Basic .NET.

Memory Management

Visual Basic .NET relies on the .NET runtime for memory management. This means that the .NET runtime takes care of reserving memory for all Visual Basic strings, arrays, structures, and objects. Likewise, the .NET runtime decides when to free the memory associated with the objects or variables you have allocated. This is not much different from Visual Basic 6, which was also responsible for managing the memory on your behalf. The most significant difference between Visual Basic 6 and Visual Basic .NET in terms of memory management involves determining when an object or variable is freed.

In Visual Basic 6, the memory associated with a variable or object is freed as soon as you set the variable to *Nothing* or the variable falls out of scope. This is not true in Visual Basic .NET. When a variable or object is set to *Nothing* or falls out of scope, Visual Basic .NET tells the .NET runtime that the variable or object is no longer used. The .NET runtime marks the variable or object as needing deletion and relegates the object to the Garbage Collector (GC). The Garbage Collector then deletes the object at some arbitrary time in the future.

Because we can predict when Visual Basic 6 will delete the memory associated with a variable, we refer to the lifespan of a variable in that language as being **deterministic**. In other words, you know the exact moment that a variable comes into existence and the exact moment that it becomes nonexistent. The lifespan of a Visual Basic .NET variable, on the other hand, is **indeterministic**, since you cannot predict exactly when it will become nonexistent. You can tell Visual Basic .NET to stop using the variable, but you cannot tell it when to make the variable nonexistent. The variable could be left dangling for a few

nanoseconds, or it could take minutes for the .NET Framework to decide to make it nonexistent. In the meantime, an indeterminate amount of your Visual Basic code will execute.

In many cases it does not matter whether or not you can predict when a variable or object is going to be nonexistent. For example, a simple variable such as a string or an array that you are no longer using can be cleaned up at any time. It is when you are dealing with objects that things get interesting.

Take, for example, a *File* object that opens a file and locks the file when the *File* object is created. The object closes the file handle and allows the file to be opened by other applications when the object is destroyed. Consider the following Visual Basic .NET code:

```
Dim f As New File
Dim FileContents As String
f.Open("MyFile.dat")
FileContents = f.Read("MyFile.dat")
f = Nothing
FileContents = FileContents & " This better be appended to my file! "
f.Open("MyFile.dat")
f.Write(FileContents)
f = Nothing
```

If you run this application in Visual Basic 6, it will run without error. However, if you run this application in Visual Basic .NET, you will encounter an exception when you attempt to open the file the second time. Why? The file handle associated with MyFile.dat will likely still be open. Setting *f* to *Nothing* tells the .NET Framework that the *File* object needs to be deleted. The runtime relegates the object to the garbage bin, where it will wait until the Garbage Collector comes along to clean it up. The *File* object in effect remains alive and well in the garbage bin. As a result, the MyFile.dat file handle is still open, and the second attempt to open the locked file will lead to an error.

The only way to prevent this type of problem is to call a method on the object to force its handle to be closed. In this example, if the *File* object had a *Close* method, you could use it here before setting the variable to *Nothing*. For example,

```
f.Close
f = Nothing
```

Dispose: Determinism in the Face of Chaos

Despite all of the benefits that a garbage-collected model has to offer, it has one haunting side effect: the lack of determinism. Objects can be allocated and deleted by the hundreds, but you never really know when or in what order

they will actually terminate. Nor do you know what resources are being consumed or locked at any given moment. It's confusing, even chaotic. To add some semblance of order to this system, the .NET Framework offers a mechanism called *Dispose* to ensure that an object releases all its resources exactly when you want it to. Any object that locks resources you need or that otherwise needs to be told to let go should implement the *IDisposable* interface. The *IDisposable* interface has a single method, *Dispose*, that takes no parameters. Any client using the object should call the *Dispose* method when it is finished with the object.

One More Thing to Worry About

If you've been using Visual Basic 6, you're not accustomed to calling *Dispose* explicitly on an object when you write code. Unfortunately, when it comes to Visual Basic .NET, you will have to get accustomed to doing so. Get in the habit now of calling *Dispose* on any object when you are done using it or when the variable referencing it is about to go out of scope. If we change the *File* object shown earlier to use *Dispose*, we end up with the following code:

```
Dim f As New File
Dim FileContents As String
f.Open("MyFile.dat")
FileContents = f.Read("MyFile.dat")
f.Dispose
f = Nothing
FileContents = FileContents & " This better be appended to my file! "
f.Open("MyFile.dat")
f.Write(FileContents)
f.Dispose
f = Nothing
```

> **Note** The Visual Basic Upgrade Wizard does not alert you to cases in which you may need to call *Dispose*. We advise you to review your code after you upgrade to determine when an object reference is no longer used. Add calls to the object's *Dispose* method to force the object to release its resources. If the object—notably ActiveX objects that do not implement *IDisposable*—does not support the *Dispose* method, look for another suitable method to call, such as *Close*. For example, review your code for the use of ActiveX Data Objects (ADO) such as *Recordset* and *Connection*. When you are finished with a *Recordset* object, be sure to call *Close*.

When You Just Want It All to Go Away

While your application runs, objects that have been created and destroyed may wait for the Garbage Collector to come and take them away. At certain points in your application, you may need to ensure that no objects are hanging around locking or consuming a needed resource. To clean up objects that are pending collection, you can call on the Garbage Collector to collect all of the waiting objects immediately. You can force garbage collection with the following two calls:

```
GC.Collect
GC.WaitForPendingFinalizers
```

> **Note** The two calls to *Collect* and to *WaitForPendingFinalizers* are required in the order shown above. The first call to *Collect* kicks off the garbage collection process asynchronously and immediately returns. The call to *WaitForPendingFinalizers* waits for the collection process to complete.

Depending on how many (or few) objects need to be collected, running the Garbage Collector in this manner may not be efficient. Force garbage collection sparingly and only in cases where it's critical that all recently freed objects get collected. Otherwise, opt for using *Dispose* or *Close* on individual objects to free up needed resources as you go.

Type Identity

Mike once played on a volleyball team where everyone on his side of the net, including himself, was named Mike. What a disaster. All the other team had to do was hit the ball somewhere in the middle. Someone would yell, "Get it, Mike!" and they would all go crashing into a big pile. To sort things out, they adopted nicknames, involving some derivation of their full names. After that, the game went much better.

Like names in the real world, types in Visual Basic can have the same name. Instead of giving them nicknames, however, you distinguish them by using their full name. For example, Visual Basic has offered a variety of data access models over the years. Many of these data access models contain objects with the same names. Data Access Objects (DAO) and ActiveX Data Objects (ADO), for instance, both contain types called *Connection* and *Recordset*.

Suppose that, for whatever reason, you decided to reference both DAO and ADO in your Visual Basic 6 project. If you declared a *Recordset* variable, the variable would be either a DAO or an ADO *Recordset* type:

```
Dim rs As Recordset
```

How do you know which type of *Recordset* you are using? One way to tell is to look at the properties, methods, and events that IntelliSense or the event drop-down menu gives you. If the object has an *Open* method, it is an ADO *Recordset*. If instead it has an *OpenRecordset* method, it is a DAO *Recordset*. In Visual Basic 6, the *Recordset* you end up with depends on the order of the references. The reference that appears higher in the list wins. In Figure 2-1, for example, the Microsoft ActiveX Data Objects 2.6 Library reference occurs before the reference to the Microsoft DAO 3.6 Object Library, so ADO wins and the *Recordset* is an ADO *Recordset* type.

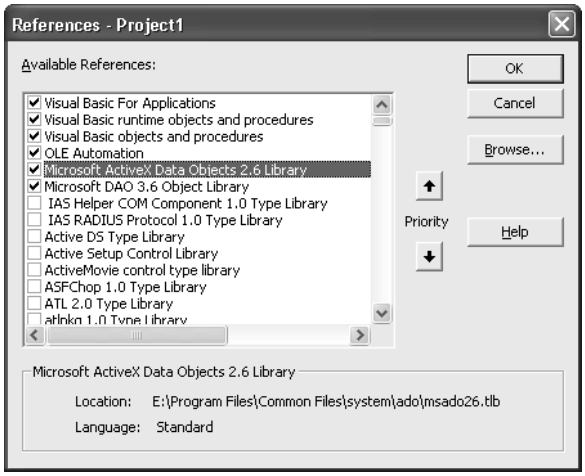

Figure 2-1 ADO 2.6 reference takes precedence over DAO 3.6.

If you change the priority of the ADO reference by selecting it and clicking the down arrow under Priority, the DAO reference will take precedence. Clicking OK to apply the change transforms your *Recordset* type to a DAO *Recordset*.

Suppose you want to use both types of *Recordset* objects in your application. To do so, you need to fully qualify the type name as follows:

```
Dim rsADO As ADODB.Recordset
Dim rsDAO As DAO.Recordset
```

As you can see, Visual Basic 6 is quite flexible when it comes to using types. Indeed, you could argue that it is too flexible, since you could mistakenly change the type for variables in your code simply by changing the order of a reference.

Visual Basic .NET is stricter about the use of the same types in an application. The general rule is that you need to fully qualify every ambiguous type that you are using. If you are referencing both ADO and DAO, for example, you are forced to fully qualify your use of the types just as you would in Visual Basic 6:

```
Dim rsADO As ADODB.Recordset
Dim rsDAO As DAO.Recordset
```

Using Imports

To help you cut down on the number of words and dots that you need to type for each reference, Visual Basic .NET allows you to import a namespace. You can think of it as a global *With* statement that is applied to the namespace. (A namespace is similar to a library or project name in Visual Basic 6.) For example, type references can become quite bothersome when you are dealing with .NET types such as *System.Runtime.Interopservices.UnmanagedType*. To simplify the qualification of this type, you can add an *Imports* statement to the beginning of the file in which it is used:

```
Imports System.Runtime
```

This statement allows you to reference the type as *Interopservices.UnmanagedType*. You can also expand the *Imports* clause to

```
Imports System.Runtime.Interopservices.
```

and then simply refer to *Unmanaged Type* in your code.

Managing Conflicts

Imports works great until there is a conflict. As we indicated earlier, in Visual Basic 6, the rule is that the type library that is higher in the precedence list takes priority. Visual Basic .NET is different in that all conflicts are irreconcilable. You have to either change your *Imports* clause to avoid the conflict or fully qualify each type when it is used. Suppose that you add *Imports* statements for ADO and DAO as follows:

```
Imports ADO
Imports DAO
```

Now suppose that you want to declare a variable of type *Recordset*. As in the volleyball game described earlier, it's as if you yelled out, "Recordset!" Both ADO and DAO jump in. Crash! Big pile. Any attempt to use the unqualified type *Recordset* will lead to an error that states, "The name 'Recordset' is ambiguous, imported from Namespace ADO, DAO." To resolve the problem, you need to either fully qualify the type or remove one of the *Imports* statements.

No More GUIDs

Each ActiveX type, whether it is a class, an interface, an enumerator, or a structure, generally has a unique identifier associated with it. The identifier is a 128-bit, or 16-byte, numeric value referred to as UUID, GUID, LIBID, CLSID, IID, or <whatever>ID. No matter what you call it, it is a 128-bit number.

Rather than make you think in 128-bit numbers, Visual Basic (and other languages) associates human-readable names with each of these types. For example, if you create a Visual Basic 6 class called *Customer*, its type identifier will be something like {456EC035-17C9-433c-B5F2-9F22C29D775D}. You can assign *Customer* to other types, such as *LoyalCustomer*, if *LoyalCustomer* implements the *Customer* type with the same ID value. If the *LoyalCustomer* type instead implements a *Customer* type with a different ID value, the assignment would fail with a "Type Mismatch" error. In ActiveX, at run time, the number is everything; the name means little to nothing.

In .NET, on the other hand, the name is everything. Two types are considered the same if they meet all of the following conditions:

- The types have the same name.
- The types are contained in the same namespace.
- The types are contained in assemblies with the same name.
- The assemblies containing the types are weak named.

Note that the types can be in assemblies that have the same name but a different version number. For example, two types called *Recordset* contained in the namespace ADODB are considered the same type if they live in an assembly such as Microsoft.ADODB.dll with the same name. There could be two Microsoft.ADODB.dll assemblies on your machine with different version numbers, but the *ADODB.Recordset* types would still be considered compatible. If, however, the *Recordset* types lived in different assemblies, such as Microsoft.ADODB_2_6.dll and Microsoft.ADODB_2_7.dll, the types would be considered different. You cannot assign two variables of type *Recordset* to each other if each declaration of *Recordset* comes from an assembly with a different name.

Threading Model

Visual Basic 6 ActiveX DLLs and controls can be either single threaded or apartment threaded; they are apartment threaded by default. **Apartment threading** means that only one thread can access an instance of your Visual Basic 6 ActiveX component at any given time. In fact, the same thread always accesses your component, so other threads never disturb your data, including global data. Visual Basic .NET components, on the other hand, are **multithreaded** by default, meaning that two or more threads can be executing code within your component simultaneously. Each thread has access to your shared data, such as class member and global variables, and the threads can change any data that is shared.

Visual Basic .NET multithreaded components are great news if you want to take advantage of MTS pooling, which requires multithreaded components. They are bad news if your component is not multithread safe and you wind up trying to figure out why member variables are being set to unexpected or random values in your upgraded component.

There is certainly more to cover on this topic, but we'll leave that for the discussion of threading in Chapter 11where we discuss how to make your multithreaded Visual Basic .NET components multithread safe.

Differences in the Development Environment

Although Visual Basic 6 shipped as part of Microsoft Visual Studio 6, it did not share a common infrastructure with its siblings C++, Visual InterDev, and Visual FoxPro. The only sharing came in the form of ActiveX components and in designers such as the DataEnvironment. Although Visual Studio 6 shipped with a common integrated development environment (IDE) called MSDev, Visual Basic 6 did not participate in MSDev and instead came with its own IDE called VB6.exe.

Visual Studio .NET ships with a single IDE that all languages built on the .NET Framework share called Devenv.exe. The Visual Studio .NET IDE is a host for common elements such as the Windows and Web Forms packages, the Property Browser, Solution Explorer (also known as the project system), Server Explorer, Toolbox, Build Manager, add-ins, and wizards. All languages, including Visual Basic .NET and C#, share these common elements.

Although the Visual Studio .NET IDE provides a common environment for different languages, the various languages are not identical or redundant. Each language maintains its own identity in the syntax, expressions, attributes, and

runtime functions you use. When you write code behind a form in a common forms package such as Windows Forms or Web Forms, the code behind the form is represented by the language you are using. If you use Visual Basic, the events for the form are represented using Visual Basic syntax and have event signatures almost identical to those you are accustomed to using in Visual Basic 6. If you use C#, all of the Windows Forms event signatures appear in the syntax of the C# language.

What happened to the common tools that you have grown to love or hate in Visual Basic 6? They have all been rewritten for Visual Studio. NET, as you'll see next.

Menu Editor

Do you really want to keep using the same clunky Menu Editor that has been around since Visual Basic 1, shown in Figure 2-2? We doubt it. So you'll probably be pleased to know that you won't find it in the Visual Studio .NET environment. Instead, you create menus by inserting and editing the menu items directly on a Windows form.

Figure 2-2 Visual Basic 6 Menu Editor.

To insert a new menu in the .NET environment, you drag a *MainMenu* component from the Toolbox and drop it on the form. Then you select the *MainMenu1* component in the component tray, below the form, and type your menu text in the edit box that says "Type Here" just below the title bar for your form. Figure 2-3 shows the Visual Basic .NET menu editor in action.

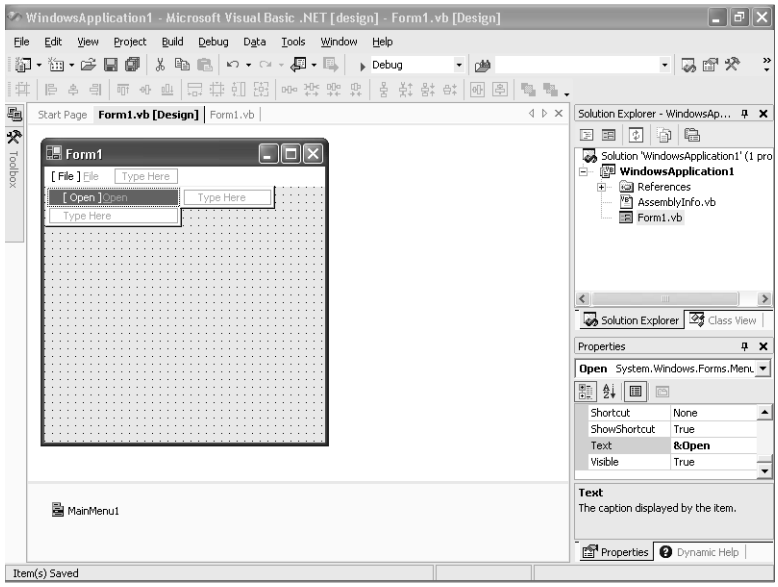

Figure 2-3 Visual Basic .NET's in-place menu editor.

Toolbox

The Visual Studio .NET Toolbox is similar to the Visual Basic 6 Toolbox in appearance and use. A difference you will notice right away is that the Visual Studio .NET Toolbox contains the name of each Toolbox item in addition to the icon. Also, depending on the type of project selected, the Toolbox displays a variety of tabs containing different categories of controls and components that you can add to a form or designer. For example, when you are editing a Windows Forms project, the Toolbox will contain categories titled Data, Components Windows Forms, Clipboard Ring, and General. Each tab contains ADO .NET data components such as *DataSet* and *OleDBAdaptor*; system components such as *MessageQueue* and *EventLog*; and Windows Forms controls and components such as *Button*, *TextBox*, *Label*, and *TreeView*.

A subtle difference between the Visual Basic 6 Toolbox and the Visual Basic .NET Toolbox relates to references. In Visual Basic 6, any ActiveX control you add to the Toolbox is also added as a reference within your project. The reference exists whether you use the ActiveX control on a form or not. In Visual Basic .NET, the items you add to the Toolbox are not referenced by default. It is not until you place the control on a Windows form or designer that a reference to that component is added to your project.

Because a reference to an ActiveX control automatically exists when you place the control on the Toolbox in Visual Basic 6, you can use the reference in code. For example, suppose you add the Masked Edit ActiveX control to the Toolbox but don't add an instance of the control to the form. You can write code to add an instance of the Masked Edit ActiveX control to a form at runtime, as follows:

```
Dim MyMSMaskCtl1 As MSMask.MaskEdBox
Set MyMSMaskCtl1 = Controls.Add("MSMask.MaskEdBox", "MyMSMaskCtl1")
MyMSMaskCtl1.Visible = True
```

If you attempt to place a Masked Edit ActiveX control on a Visual Basic .NET Toolbar, you will find that if you declare a variable of the ActiveX control type, the statement will not compile. For example, if you attempt to declare the Masked Edit control, using Visual Basic .NET equivalent syntax, the statement won't compile, as follows:

```
Dim MyMSMaskCtl1 As AxMSMask.AxMaskEdBox
```

To declare a variable of the ActiveX control type, you need to place the ActiveX control on a form. You will then be able to dimension variables of the ActiveX control type.

> **Note** After you place an ActiveX control on a Visual Basic .NET form, you will find that you can declare variables of the control type. However, you will not be able to use *Controls.Add*, as demonstrated in the Visual Basic 6 code above. *Controls.Add* is not supported in Visual Basic .NET.

Property Browser

The Visual Studio .NET Property Browser is, for the most part, identical in terms of appearance and use to the Visual Basic 6 Property Browser. One minor difference is that the default view for the Property Browser in Visual Studio .NET is Category view, meaning that related properties are grouped under a descriptive category. Alphabetical view is also supported. The Visual Basic 6 Property Browser, on the other hand, defaults to listing properties alphabetically, although it supports a categorized view.

The Visual Studio .NET Property Browser can list all of the properties associated with a control or component. This is not the case when you are using the Visual Basic 6 Property Browser. For example, the Visual Basic 6

Property Browser cannot list object or variant-based properties. It can display properties for a limited number of objects, such as *Picture* or *Font*, but it cannot represent an object property such as the ColumnHeaders collection of a ListView control. Instead the Visual Basic 6 Property Browser relies on an ActiveX control property page to provide editing for object properties such as collections.

The Visual Studio .NET Property Browser allows direct editing of an object property if a custom editor is associated with the property or the property type. For example, the Visual Studio .NET Property Browser provides a standard Collection Editor for any property that implements *ICollection*. In the case of the ColumnHeaders collection for a ListView control, a ColumnHeader Collection Editor, based on the standard Collection Editor, is provided for you to edit the ColumnHeaders collection for the ListView. Figure 2-4 shows an example of editing the *ListView Columns* property.

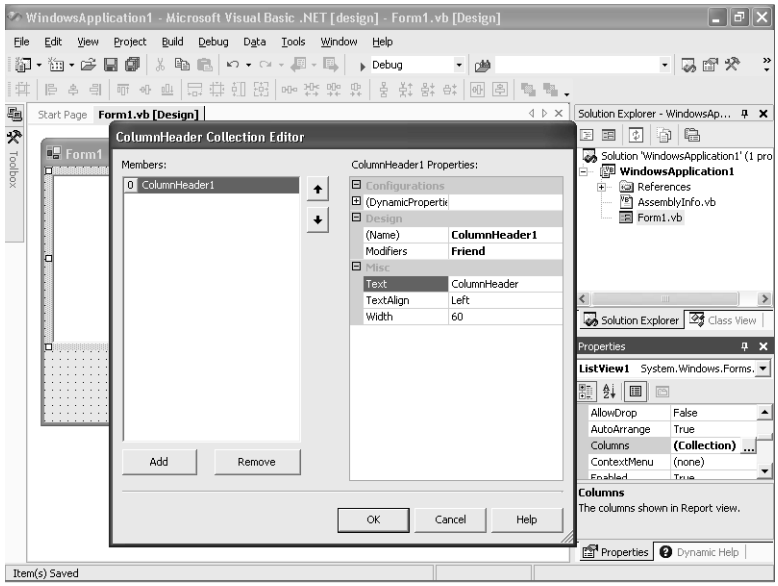

Figure 2-4 Visual Basic .NET ColumnHeader Collection Editor in action.

Tab Layout Editor

Your days of clicking a control, setting the *TabIndex* property, and then repeating the process for the several dozen controls on your form are over. Welcome to the Visual Studio .NET Tab Layout Editor. The Tab Layout Editor allows you to view and edit the tab ordering for all elements on the form at once. To view your tab layout for the current form, select Tab Order from the View menu. A tab index number displays for each control on the form. You can start with the control that you want to be first in the tab order, and then click the remaining

controls in the tab order that you want. The tab index numbers will correspond to the order in which you click the controls. Figure 2-5 illustrates the Tab Layout Editor.

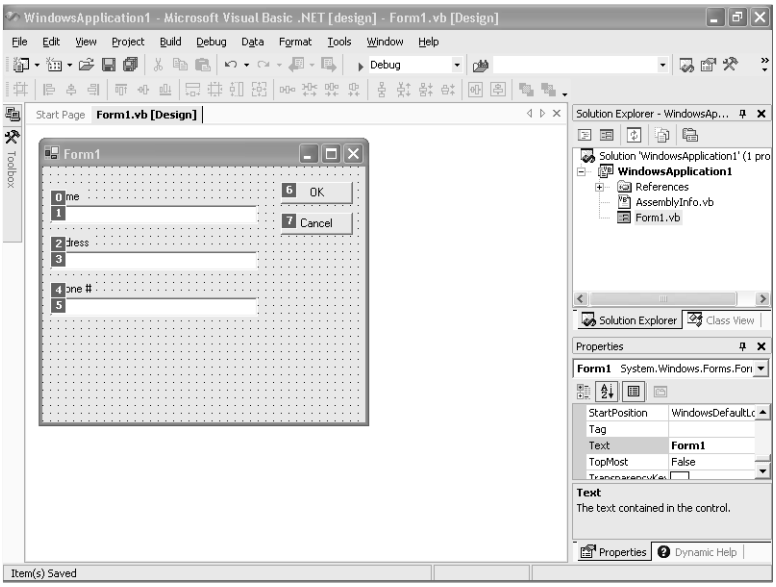

Figure 2-5 Visual Studio .NET Tab Layout Editor in action.

Forms Packages

The forms package that you use in Visual Basic 6 to create standard .exe projects or ActiveX control projects is essentially the same package that has been in existence since Visual Basic 1. Visual Basic .NET offers a brand new forms package called Windows Forms. In addition, Visual Basic .NET gives you a second forms package to help in creating Web applications: the Web Forms package.

A Single Standard for Windows Forms

A significant difference between Visual Basic .NET and Visual Basic 6 is that the forms you use with Visual Basic .NET can be used in any type of .NET project. For example, you can use the same forms with both a Visual Basic application and a C# application.

The forms package found in Visual Basic 6 is local to that environment. You can use Visual Basic 6 forms only in Visual Basic 6. Microsoft has tried in the past to create a single, standard forms package that could be shared across multiple products such as Visual Basic, C++, and Office. The initiative, called Forms3 (pronounced Forms Cubed), never realized this goal. Forms3 is alive and well in Office but was never made fully compatible with the Visual Basic forms package.

The Windows Forms package reignites some hope of having a single forms standard applied across various Microsoft products—at least for client applications based on the .NET platform. The ideal of having a single, universal forms package, however, will need to wait; Visual Studio .NET also introduces a separate forms package for Web applications.

Two Forms Packages for the Price of One

One of the appealing features of Visual Studio .NET is that you can create a Web application more quickly and easily than you ever have before. This ease stems from the marriage between the Web Forms package and Visual Basic .NET. For the first time, you can create a Web application in the same manner that you create a Windows client application. You drag and drop controls onto a Web form and then write code to handle the form and control events. All of the skills that you use to create Visual Basic Windows applications can now be used to create Web applications.

> **Note** The Upgrade Wizard will upgrade your client-based applications to use Windows Forms and will upgrade your WebClasses-based applications to use Web Forms.

Language Differences

With each new version of Visual Basic, Microsoft has expanded the language by offering new keywords, new syntactical elements, new conditional statements or modifiers, new attributes, and so on. Visual Basic .NET is no exception. It makes the same types of additions to the language as previous versions have, but on a much grander scale than before. Table 2-1 gives a complete list of keywords that have been added to the Visual Basic .NET language.

Table 2-1 New Keywords in Visual Basic .NET

Visual Basic .NET Keyword	Description
AddHandler and *RemoveHandle*	Dynamically adds or removes event handlers at runtime, respectively
AndAlso and *OrElse*	Short circuited logical expressions that complement *And* and *Or*, respectively
Ansi, *Auto*, and *Unicode*	Declare statement attributes
CChar, *CObj*, *CShort*, *CType*, and *DirectCast*	Coercion functions
Class, *Interface*, *Module,* and *Structure*	Type declaration statements
Default	Attribute for indexed property declarations
Delegate	Declare pointer to instance method or shared method
GetType	Returns *Type* class for a given type
Handles	Specifies event handled by a subroutine
Imports	Includes given namespace in current code file
Inherits	Optional statement used with a class to declare classes that inherit from another class
MustInherit	Optional statement used with a class to declare the class as an abstract base class
MustOverride	Optional subroutine attribute that specifies an inherited class must implement the subroutine
MyBase	Refers to base class instance
MyClass	Refers to the current class instance. Ignores a derived class.
Namespace	Defines a namespace block
NotInheritable	Optional statement used with *Class* to indicate the class cannot be inherited
NotOverridable	Optional subroutine attribute which specifies that a subroutine cannot be overridden in a derived class
Option Strict	Allows you to turn strict type conversion checking on or off. Default is off.
Overloads	Optional subroutine attribute that indicates the subroutine overloads a subroutine with the same name, but different parameters
Overridable	Optional subroutine attribute which specifies that a subroutine can be overridden in a derived class

Table 2-1 New Keywords in Visual Basic .NET *continued*

Visual Basic .NET Keyword	Description
Overrides	Optional subroutine attribute that indicates the subroutine overrides a subroutine in the base class
Protected	Class member attribute that limits member access to the class and any derived class
Protected Friend	Same as *Protected*, but expands the scope to include access by any other class in the same assembly
ReadOnly and *WriteOnly*	Attribute on a *Property* declaration to specify the property is read-only or write-only
*Return**	Statement used to return, possibly with a value from a subroutine
Shadows	Attribute on class members to specify that a class member is distinct from a same-named base class member
Short	16-bit type known as *Integer* in Visual Basic 6
SyncLock	Specifies the start of a thread synchronization block
Try, *Catch*, *Finally*, and *When*	Keywords related to structured error handling
Throw	Keyword to throw an exception

* Existing keyword with different behavior.

Because the Upgrade Wizard generally does not modify or update your code to take advantage of new Visual Basic .NET features, only a subset of the new features come into play after an upgrade. Therefore, we will focus here on some of the general language differences that affect your upgraded Visual Basic 6 application. Chapter 11covers how to deal with these and other language changes in detail. The sections that follow describe the types of changes you will notice when you look at your upgraded Visual Basic .NET code.

All Subroutine Calls Must Have Parentheses

Parentheses are required on all subroutine calls. If you write code that does not use the *Call* keyword, as follows:

```
MsgBox "Hello World"
```

you are required to use parentheses in your Visual Basic .NET code, as follows:

```
MsgBox("Hello World")
```

ByVal or *ByRef* Is Required

In Visual Basic .NET, all subroutine parameters must be qualified with *ByVal* or *ByRef*. For example, instead of this Visual Basic 6 code:

```
Sub UpdateCustomerInfo(CustomerName As String)
End Sub
```

you will see the following Visual Basic .NET code:

```
Sub UpdateCustomerInfo(ByRef CustomerName As String)
End Sub
```

In this case, an unqualified Visual Basic 6 parameter has been upgraded to use the *ByRef* calling convention. In Visual Basic .NET, the default calling convention is *ByRef*.

Is That My Event?

Visual Basic 6 associates events by name, using the pattern <Object-Name>_<EventName>. For example, the click event associated with a command *CommandButton* is

```
Private Sub Command1_Click()
```

If you change the name of the Visual Basic 6 event to the name of a subroutine that does not match any other event, it becomes a simple subroutine. The name pattern, therefore, determines whether a subroutine is an event or not.

Handles Clause

Visual Basic .NET does not associate events by name. Instead, a subroutine is associated with an event if it includes the *Handles* clause. The name of the subroutine can be any name you want. The event that fires the subroutine is given in the *Handles* clause. For example, the click event associated with a Visual Basic .NET button has the following signature:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles Button1.Click
```

Because the event hookup is an explicit part of the event declaration, you can use unique names for your events. For example, you can change the name of your *Button1_Click* event to *YouClickedMyButton* as follows:

```
Private Sub YouClickedMyButton(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles Button1.Click
```

Event Parameters

Another interesting change related to events is that event parameters are different between Visual Basic 6 and Visual Basic .NET. In Visual Basic 6, the event subroutine contains the name and type of each parameter. In Visual Basic .NET, the parameters are bundled up in an *EventArgs* object and passed in as a reference to that object. Also, the event subroutine for a Visual Basic .NET event includes a reference to the object that fired the event.

As an example of the different handling of event parameters in the two versions of Visual Basic, consider a form with a Listbox control on it, for which you need to write code to show the checked item.

In Visual Basic 6, you would write the following code:

```
Private Sub List1_ItemCheck(Item As Integer)
   MsgBox "You checked item: " & Item
End Sub
```

The equivalent code in Visual Basic .NET is as follows:

```
Private Sub CheckedListBox1_ItemCheck(ByVal sender As Object, _
   ByVal e As System.Windows.Forms.ItemCheckEventArgs) _
   Handles CheckedListBox1.ItemCheck
   MsgBox("You checked item: " & e.Index)
End Sub
```

Observe how the item that is checked is passed directly as a parameter in Visual Basic 6. In Visual Basic .NET, it is passed as a member of the passed-in *ItemCheckEventArgs* object *e*.

Arrays Must Have a Zero-Bound Lower Dimension

You cannot declare an array in Visual Basic .NET to have a nonzero-bound lower dimension. This requirement also means that you cannot use Option Base 1. In fact, you cannot specify a lower dimension in an array declaration, since it must always be zero. The following types of declarations are no longer supported:

```
Dim MyIntArray(-10 To 10) As Integer   '21 elements
Dim  MyStringArray(1 To 100) As String   '100 elements

Option Base 1
Dim MyOptionBase1Array(5) As Long   '5 elements (1-5)
```

Instead, you must use zero-based lower bound arrays, and you need to adjust the bounds to create an array with the same number of elements, such as

```
Dim MyIntArray(20) As Integer   '21 elements (0-20)
Dim  MyStringArray(99) As String   '100 elements (0-99)

'Option Base 1      'Not supported by VB .NET
Dim MyOptionBase1Array(4) As Long      '5 elements (0-4)
```

Refer to Chapter 11 for more information on how you can change your array declarations in Visual Basic .NET to be compatible with your array declarations in Visual Basic 6.

Fixed-Length Strings Are Not Supported

Visual Basic .NET does not support fixed-length strings. For example, the following type of declaration is not supported:

```
Dim MyString As String * 32
```

Instead, you can dimension the string as a fixed-length array of characters, as follows:

```
Dim MyString(32) As Char
```

Or you can use a special class, *VBFixedLengthString*, defined in the Visual Basic .NET compatibility library. If you use the *VBFixedLengthString* class the declaration will be:

```
Imports VB6 = Microsoft.VisualBasic.Compatibility.VB6
   …
Dim MyFixedLenString As New VB6.FixedLengthString(32)
```

To set the value of a *FixedLengthString* variable you need to use the *Value* property as follows:

```
MyFixedLenString.Value = "This is my fixed length string"
```

Refer to Chapter 7 for more information about the Visual Basic .NET compatibility library.

Variant Data Type Is Eliminated

Visual Basic .NET eliminates the *Variant* data type. The main reason is that the underlying .NET Framework does not natively support the *Variant* type or anything like it. The closest approximation that the .NET Framework offers is the *Object* type. The *Object* type works somewhat like the *Variant* type because the *Object* type is the base type for all other types, such as *Integer* and *String*. Just as you can with a *Variant*, you can assign any type to an *Object*. However, in Visual Basic .NET, to get a strong type back out of a *Variant* to assign, for example, to an *Integer* or a *String*, you need to use a type-casting function, such as *CInt* or *CString*. With Visual Basic 6, you can write code such as the following:

```
Dim v As Variant
Dim s As String
v = "My variant contains a string"
s = v
```

When using Visual Basic .NET, however, you need to use type conversion functions such as *CStr*, as follows:

```
Dim v As Variant
Dim s As String
v = "My variant contains a string"
s = CStr(v)
```

Refer to Chapter 11 for more information on differences between the Visual Basic 6 *Variant* and Visual Basic .NET *Object* types.

Visibility of Variables Declared in Nested Scopes Is Limited

Variables that are declared in a nested scope, such as those occurring within an *If…Then* or *For…Next* block, are automatically moved to the beginning of the function. The Upgrade Wizard does this for compatibility reasons. In Visual Basic 6, a variable declared in any subscope is visible to the entire function. In Visual Basic .NET, this is not the case. A variable declared within a subscope is visible only within that subscope and any scope nested beneath it.

Take, for example, the following Visual Basic code:

```
Dim OuterScope As Long

If OuterScope = False Then
   Dim InnerScope As Long
End If

InnerScope = 3
```

This code works fine in Visual Basic 6, but it will lead to a compiler error in Visual Basic .NET. The compiler error will occur on the last line, *InnerScope = 3*, and will indicate that the name *InnerScope* is not declared.

> **Note** The Upgrade Wizard will upgrade your code so that no compiler error occurs. It does this by moving the declaration for *InnerScope* to the top of the function along with all other top-level declarations. Moving the variable declaration to the top-level scope allows the variable to be used from any scope within the function. This move makes the behavior compatible with Visual Basic 6. It is one of the few cases in which the Upgrade Wizard changes the order of code during upgrade.

Changes in the Debugger

Visual Basic .NET shares the same debugger with all .NET languages in Visual Studio .NET. This debugger works much the same as the one in Visual Basic 6 in that you can step through code and set breakpoints in the same way. However, there are some differences that you should be aware of. These are discussed in the following sections.

No Edit and Continue

What percentage of your Visual Basic 6 application would you say is developed when you are debugging your application in what is commonly referred to as break mode? Ten percent? Forty percent? Ninety percent? Whatever your answer, the number is likely above zero. Any problems you encounter while debugging your Visual Basic 6 application are quite easy to fix while in break mode. This is a great feature that allows you to create applications more quickly. You will miss this ability in Visual Basic .NET.

The Visual Studio .NET common debugger does not allow you to edit your code while in break mode. Any time you encounter code that you want to change or fix, you need to stop debugging, make the change, and then start the application again. Doing so can be a real pain.

The Visual Basic .NET team recognizes that this is not what you would call a RAD debugging experience. The team hopes to offer an updated debugger that supports edit and continue in a future release of Visual Studio .NET. Until then, prepare to break, stop, edit, and rerun your application.

Cannot Continue After an Error

If an error or exception occurs while you are running your application, the Visual Basic .NET debugger will stop at the point where the exception occurred. However, unlike Visual Basic 6, in the Visual Basic .NET debugger you cannot fix your code or step around the code that is causing the error. If you attempt to step to another line, the application will terminate and switch to Design view. You will need to determine the source of the exception, fix your code, and then rerun the application.

No Repainting in Break Mode

In Visual Basic 6, the form and all controls on it continue to display even when you are in break mode. This happens because the Visual Basic 6 debugger lets certain events occur and allows certain code to execute when you are in break mode. For example, painting is allowed to occur.

When debugging your application using Visual Basic .NET, you will find that your form does not repaint. In fact, if you place another window over it while you are in break mode, you will find that the form image does not update at all. The Visual Basic .NET debugger does not allow any events or code to run while you are in break mode.

One benefit of the Visual Basic .NET debugger is that you can debug your paint code and watch your form update as each statement that paints the form executes. It allows you to pinpoint the exact statement in your code that is causing a problem with the display. Because the Visual Basic 6 debugger allows the form to repaint constantly, it is difficult to pinpoint painting problems using the Visual Basic 6 debugger.

Conclusion

As you can see, quite a bit is involved in the three "simple" changes that the teams made to create Visual Basic .NET. Despite all of these changes, you should find the development environment, compiler, and language familiar. The skills that you have acquired using Visual Basic 6 are not lost when you upgrade to Visual Basic .NET. The way you create, run, and debug a Visual Basic .NET application is nearly identical to the process you are already familiar with. After all, Visual Basic is still Visual Basic. The spirit is alive and well in Visual Basic .NET.