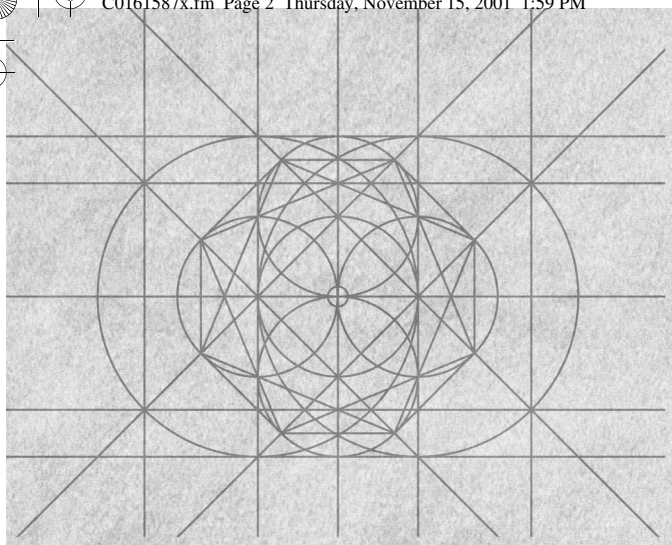
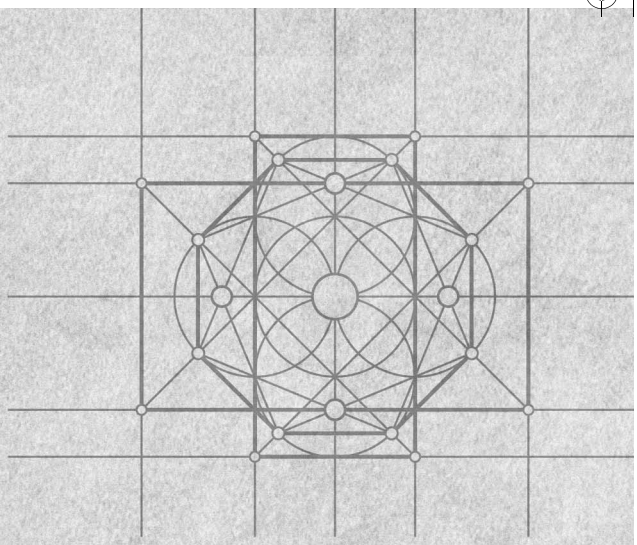
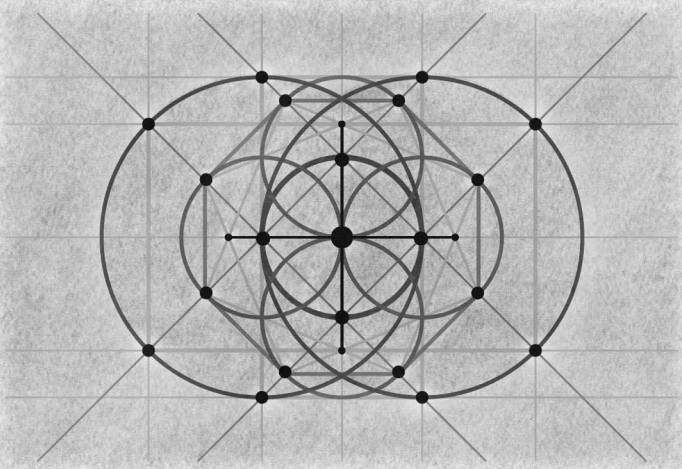
C0161587x.fm Page 1 Thursday, November 15, 2001 1:59 PM

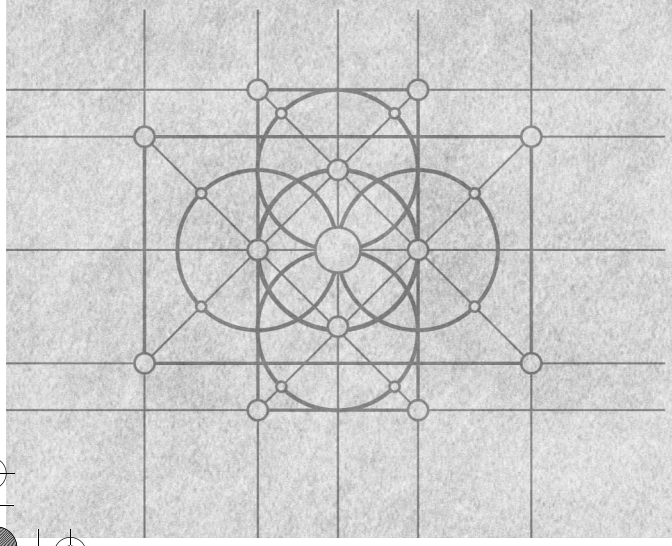
Part I

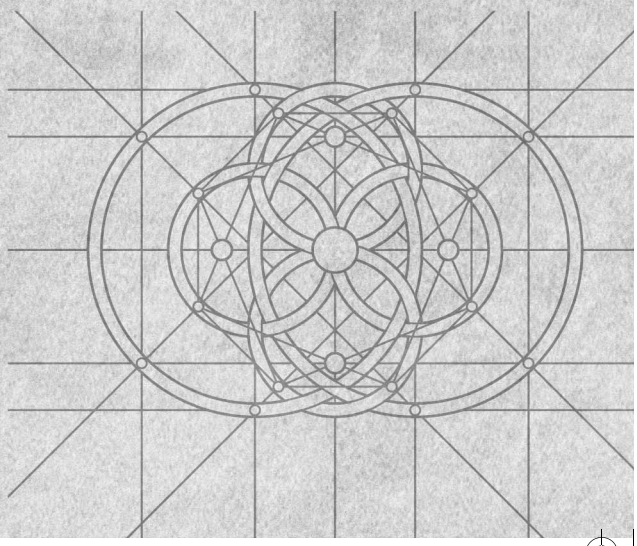
Introduction to Upgrading



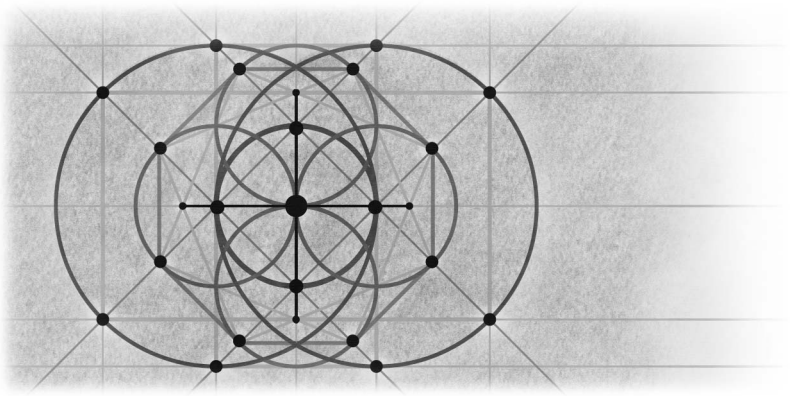








C0161587x.fm Page 3 Thursday, November 15, 2001 1:59 PM



1

Visual Basic .NET Is More Than Visual Basic 6 + 1

If you're familiar with Visual Basic but aren't familiar with Visual Basic .NET, you may be wondering why we have written a book on upgrading. Surely Visual Basic .NET will open Visual Basic 6 projects as effortlessly as Visual Basic 6 opens Visual Basic 5 projects. "How different can Visual Basic .NET be?" you might ask. So before we start discussing the details of upgrading, let's clear up any confusion: Visual Basic .NET, the latest version of Visual Basic, is not merely Visual Basic 6 with a few new features added on. Instead, Visual Basic has been thoroughly redesigned and restructured. The language has been modernized, with new, richer object models for data, forms, transactions, and almost everything else. The file formats have also changed.

Unfortunately, these changes mean that Visual Basic .NET is not entirely backward compatible with Visual Basic 6. Projects from previous versions need to be upgraded before they will compile and run in Visual Basic .NET. The Upgrade Wizard handles much of this work for you, but most real-world projects will require additional modifications before they can be run. Some people consider moving applications from Visual Basic 6 to .NET to be a migration rather than an upgrade, but the changes in the language are a logical step, and they make Visual Basic more powerful than ever before.

This is an exciting time for Visual Basic developers. Sure, upgrading applications takes some effort, but on the other hand Visual Basic .NET is incredibly capable, extending Visual Basic's Rapid Application Development (RAD) model to the server and to the Web. Visual Basic .NET adds more features to Visual Basic than did Visual Basic 2, 3, 4, 5, and 6 combined. Microsoft made the changes because the focus of the language has shifted from previous versions. Whereas Visual Basic 6 was primarily a Windows

C0161587x.fm Page 4 Thursday, November 15, 2001 1:59 PM

development tool, Visual Basic .NET is designed to leverage the .NET platform, enabling the user to create Windows applications, console applications, class libraries, NT services, Web Forms applications, and XML Web services—all while allowing seamless integration with other programming languages.

Let's look a little deeper at Visual Basic .NET to see where it differs from Visual Basic 6. We will look at the following three issues:

- The development environment
- The syntax of the language and object models of the classes
- The run-time behavior of the compiled components

In each of these three areas, Visual Basic .NET departs from the conventions of Visual Basic 6. First, the integrated development environment (IDE) has been redesigned to house all of the Visual Studio languages: Visual Basic, C#, and Visual C++. Second, the language itself has been modernized, removing some keywords, such as *GoSub*; adding new keywords, like *Inherits*; and changing the meaning of other keywords, like *Return* and *Integer*. Finally, the run-time behavior of the compiled components is different. .NET applications are free-threaded; Visual Basic .NET projects are compiled to "assemblies" rather than to familiar Win32 applications, and variables and objects are garbage-collected, meaning that they lack a deterministic lifetime. A noticeable effect of this last change is that the class *Finalize* event is not triggered until sometime after the object is actually destroyed. We will talk more about these differences and about what .NET is in Chapter 2. For now, let's continue to look at what the changes mean to the Visual Basic developer.

With so much that is new, how familiar will it all be to traditional Visual Basic users? To what extent can you leverage your existing Visual Basic skills when you move to Visual Basic .NET? To answer these questions, let's take a quick look at the history of Visual Basic.

In 1991, Microsoft released Visual Basic 1, which opened the doors to Windows RAD. Visual Basic 1 was an instant success, and it's easy to see why. Before Visual Basic, developers had to write *WndProc* handlers, work with pointers, and know when to apply the Pascal calling convention to methods. Visual Basic took over the handling of all of these details, allowing developers to concentrate on building business objects instead of writing the basic plumbing in every program.

In Visual Basic versions 2 though 6, Microsoft kept the underlying architecture of the product the same and simply added new features. Visual Basic 2 and 3 introduced the property grid, Data Access Objects (DAO) database programming, and object linking and embedding (OLE), resulting in a great set of features for Windows 3.1 programming. In 1995, Microsoft released Visual Basic 4,

C0161587x.fm Page 5 Thursday, November 15, 2001 1:59 PM

which enabled developers to write 32-bit EXEs, ActiveX controls, and class libraries. The year 1995 also saw the explosion of the Internet, and people began wanting to build Web sites. With versions 5 and 6, Visual Basic added its own flavor of Web development—WebClasses, ActiveX documents, and Dynamic HTML (DHTML) pages—yet for the most part, it still remained a Windows development tool.

It's interesting to compare Visual Basic 1 with Visual Basic 6 to see how far the language has come. Visual Basic 1 had no IntelliSense, no support for open database connectivity (ODBC), no classes, limited debugging, no support for COM components, no Property Browser, no Web development features, and it created only EXEs. Visual Basic 6 had come a long way from the basic forms and modules development of version 1, yet it still had the spirit of Visual Basic. Visual Basic .NET also has that spirit: It has the same human-readable language, case insensitivity, support for late binding, automatic coercions, and familiar Visual Basic keywords, functions, and constructs, like *Left$*, *MsgBox*, and *On…Error…GoTo*. If you're a Visual Basic 6 programmer, it's a comfortable step to Visual Basic .NET. Yes, there are new concepts to learn, but your existing knowledge of Visual Basic is a great foundation.

After the release of Visual Basic 6, Microsoft was faced with a challenge. Developer needs were changing. More and more programmers were developing for the Web, and the Web development capabilities built into Visual Basic 6 were not addressing their needs. Visual Basic's DHTML pages and ActiveX documents were client-side technologies, meaning that both the component and the Visual Basic runtime had to be installed on client machines. Visual Basic's WebClasses, a server-based technology, stored state on the server and wasn't scalable. In addition, the design experience for both WebClasses and DHTML pages could only be described as rudimentary! In short, the technologies were too limiting. Internet developers wanted "thin" clients, not Visual Basic downloads. They wanted code that ran on the server. They wanted security, and they needed scalability, since the more successful a Web site was, the more people would use it concurrently, and therefore the more capacity it had to have.

Clearly, a better architecture was needed. Programmers had also been asking for some significant new language features: inheritance, easier access to the underlying platform, and a solution for the many different component versioning problems that had collectively been labeled "DLL hell." When looking for a solution to these problems, Microsoft also saw the opportunity to create a unified framework for developing applications. To understand why such a framework was desirable, consider that developers wanting to create forms for Windows in Visual Basic 6, Visual C++, and Microsoft Office Visual Basic for

C0161587x.fm Page 6 Thursday, November 15, 2001 1:59 PM

Applications (VBA) had to learn a different forms package for each language. If only there was a common forms package for all these products. Life would be so much simpler! This objective and others led Microsoft to develop a common framework available to all .NET languages.

One side effect of giving all languages access to a common framework is that each language must support the same data types. This support prevents the sort of headaches familiar to anyone who has tried to use Windows APIs from Visual Basic 6. Once all languages support the same data types, it's simple to add cross-language interoperability: inheritance, debugging, security access, and an integrated compilation process. As you can see, the benefits of such a system would be amazing—and that system is exactly what the .NET platform is: a multiple-language system with a common forms package, set of base classes, and data types. For Visual Basic to be part of this revolution meant more than just changing the language—it meant reconceptualizing it from the ground up for the .NET platform.

Why Break Compatibility?

Why did Microsoft redesign and restructure the language? Why couldn't it add these new features and still keep compatibility with Visual Basic 6? There are several reasons for this, as we discuss in the sections that follow.

Adding New Features

Some of the new features in Visual Basic .NET could not have been added without a redesign. Adding visual inheritance and accessibility support to the forms package required redesigning the forms object model. Adding *Interface* statements and attributes to the language made the language more powerful by enabling a greater degree of fine-tuning but required changing the language and file formats. Fixing "DLL hell" meant that versioning and deployment had to be redesigned.

By far the biggest reason for the changes, however, was the need to integrate Visual Basic with the .NET platform. Cross-language inheritance, debugging, and unfettered access to the underlying APIs required the standardization of data types across languages, which meant changing arrays to be zero based and removing fixed-length strings from the language. Redesigning the Web and data access classes to be more scalable meant even more changes from Visual Basic 6.

C0161587x.fm Page 7 Thursday, November 15, 2001 1:59 PM

Fixing the Language

Visual Basic has grown over time, and as the language has been extended, some areas have become inconsistent and problematic. A good example of such an area is default properties. The rules for when an assignment is to be a default property and when it is to be an object have become inconsistent. Consider the following Visual Basic 6 example, where *Form1* is a form in the current project:

```
Dim v As Variant
v = Form1
```

This code causes an error because Visual Basic 6 tries to assign the default property of the form (the controls collection) to the variable *v*. Contrast this behavior with the following Visual Basic 6 code:

```
Dim v As Variant
Set v = Form1
```

In this example, *v* is assigned the value *Form1*. In both examples, the right side of the expression is exactly the same, yet the value changes depending on the context. To anyone who didn't write the code, it's unclear from looking at the code what is being assigned: the object or the default property of the object. In Visual Basic .NET, parameterless default properties are not supported and must be resolved.

Another example of an inconsistent feature is the *New* statement. Consider the following Visual Basic 6 code:

```
Dim c1 As New Class1
Dim c2 As Class1: Set c2 = New Class1
```

At first glance, the two lines seem to do exactly the same thing. Both *c1* and *c2* are being set to new instances of *Class1*. Yet the two lines have quite different behavior. The statement

```
Dim c1 As New Class1
```

means that the variable will be re-created if it is set to *Nothing* and subsequently reused, whereas the effect of

```
Dim c2 As Class1: Set c2 = New Class1
```

is that *c2* is created once. If *c2* is set to *Nothing*, it will not be re-created automatically if it is referenced again. This subtle difference in behavior can lead to hard-to-find bugs. In Visual Basic .NET, both statements cause one instance of the class to be created. If the class is destroyed, it is not automatically re-created if it is referenced again.

C0161587x.fm Page 8 Thursday, November 15, 2001 1:59 PM

Modernizing the Language

Another reason for breaking compatibility is to modernize the language. For example, the meaning of *Long* is now 64 bits, *Integer* is 32 bits, and the keyword *Type* has been changed to *Structure*. Some of these changes we can probably attribute to the "floodgate effect." Once Microsoft opened the floodgates to new features and changes to fix the language, it became more acceptable to make other changes that were not quite as critical.

It Is Still Visual Basic

Despite the changes, programmers will still recognize the Visual Basic they know and love. Let's now look at what changes you will expect to see moving to Visual Basic .NET.

Expect Subtle Differences

Visual Basic .NET has been rebuilt for the .NET platform. What does this statement mean? It means that the product has been rewritten from the ground up. One of the side effects of rewriting Visual Basic is that any similarities with previous versions of the language had to be added intentionally—you don't get them for free, as you do when you simply add new features to an existing code base. A programming language is composed of a million subtle nuances: the behavior of the *Format* function, the order of events on a form, and the undocumented hacks that are possible, like subclassing a form's message loop. Some of these subtleties are not exactly the same in Visual Basic .NET, and after upgrading an application, you may find small differences in the way the application works.

A good example is the *Currency* data type. In Visual Basic 6, the *Currency* data type has 4 digits of precision. In Visual Basic .NET, the *Currency* data type is renamed *Decimal* and has 12 digits of precision. If you run the following line of code in Visual Basic 6:

```
MsgBox( CCur(10/3) )
```

it produces 3.3333. If you run the equivalent line of code in Visual Basic .NET,

```
MsgBox( CDec(10 / 3) )
```

the result is 3.333333333333. This is not a huge change, but it underlies a principle of upgrading: Visual Basic .NET is subtly different from Visual Basic 6, and therefore upgraded applications will be different from their Visual Basic 6 counterparts in subtle ways. In most cases you will not notice the difference, yet it's

C0161587x.fm Page 9 Thursday, November 15, 2001 1:59 PM

important to be aware of the changes and to test your applications thoroughly after upgrading them. Chapter 2 examines the differences between languages in greater depth. For now, let's turn our attention to upgrading.

The Decision to Break Compatibility

When did Microsoft decide to break compatibility with Visual Basic 6? It was actually in early December 1999, during the development of Visual Basic .NET. Until that time, Visual Basic .NET was being developed to support the notion of "Visual Basic 6 sourced" projects that allowed you to edit and compile Visual Basic 6 projects in Visual Basic .NET. These projects would have a compatibility switch turned on, meaning that the language would be backward compatible with Visual Basic 6 and would even have access to the old Visual Basic 6 forms package.

By the end of 1999, it was obvious that this strategy wasn't working. Little differences were slipping through: The old forms package could not be fully integrated into .NET, and the Visual Basic 6 sourced projects could not use some of the new features of the .NET platform. At that point Microsoft made the decision to break compatibility and instead concentrate on ensuring that people could upgrade their projects from Visual Basic 6 to Visual Basic .NET.

Plan for a 95 Percent Automated Upgrade

The effect of the changes and subtle differences in Visual Basic .NET is that, unlike previous versions of Visual Basic, most real-world projects cannot be upgraded 100 percent automatically. To understand why, consider that for a 100 percent upgrade there has to be a one-to-one correlation between every element of Visual Basic 6 and a corresponding element in Visual Basic .NET. Unfortunately, this correlation does not exist. The upgrade process is closer to 95 percent, meaning that the Visual Basic .NET Upgrade Wizard upgrades 95 percent of your application, and you modify 5 percent of the application to get it working. What does 5 percent mean? If it took you 100 days to write the original Visual Basic 6 application, you might expect to take 5 days to upgrade it. This number is not set in stone—some applications are easier to upgrade than others, and the experience of the person doing the upgrade is an important factor. To prepare yourself, make sure you familiarize yourself with Part IV. It discusses how to design your Visual Basic 6 applications to make the upgrade process much smoother.

C0161587x.fm Page 10 Thursday, November 15, 2001 1:59 PM

Part III of this book is devoted to helping you learn how to make the 5 percent modifications, since this is the essential skill you'll need for upgrading. Once you've gotten your application working, Visual Basic .NET has a bunch of exciting new features that you can add to your application straight away. Chapter 18 of this book discusses some common ways you can add value to your upgraded application. In fact, we encourage you to think of the upgrade as occurring in three steps:

1. Use the Upgrade Wizard to bring your application into Visual Basic .NET.
2. Make the modifications to get your application working.
3. Start adding value with the great new features of Visual Basic .NET.

Why Should I Upgrade?

If it requires work to upgrade your applications from Visual Basic 6 to Visual Basic .NET, you may wonder, "Is upgrading worth the trouble? Why should I bother to upgrade an application that requires modifications when it works in Visual Basic 6 today?" The main reason for upgrading is to take advantage of the new features of Visual Basic .NET. What are these new features? Listing them all would be a book in itself. The following sections discuss some of the features that people commonly add to their upgraded applications.

New Language Features

Visual Basic .NET adds a number of new language features that make the language more powerful and will forever dispel the myth that Visual Basic is a "toy" programming language.

Inheritance

For years we developers had been asking Microsoft to add "real" inheritance to Visual Basic. Sure, Visual Basic 6 supports interface inheritance, but we wanted more; we wanted implementation inheritance. We wanted to benefit from code reuse and to truly implement object-oriented designs. Visual Basic .NET fully supports inheritance, via the new *Inherits* keyword. You can inherit classes from within your own application, from other applications, and from .NET components written in other languages. You can even use inheritance in forms to inherit the layout, controls, and code of another form. This is called *visual inheritance*. The following code illustrates the use of the *Inherits* keyword:

C0161587x.fm Page 11 Thursday, November 15, 2001 1:59 PM


```
Public Class BaseClass
End Class

Public Class InheritedClass : Inherits BaseClass
End Class
```

Interfaces in Code

Along with "real" inheritance, Visual Basic .NET still supports interface inheritance and improves on it by providing the *Interface* keyword. The *Interface* keyword defines the interfaces in code. Your classes then implement the interfaces, as in the following example:

```
Interface myInterface
   Function myFunction()
End Interface
Public Class myImplementedClass
   Implements myInterface
   Function myFunction() _
   Implements myInterface.myFunction
      'Some Code
   End Function
End Class
```

Structured Exception Handling

In addition to supporting the familiar *On Error GoTo* error catching, Visual Basic .NET provides a *Try...Catch...End Try* exception-handling block that adds error handling. This construct allows you to embed code within an error-handling block. A great use for this type of block is to create a global error handler for your application by including a *Try…Catch* block in the startup object such as Sub Main. In the following example, *Sub Main* opens a new instance of *Form1* and will catch and report any errors that are thrown anywhere in the application:

```
Sub Main()
   Try
      Windows.Forms.Application.Run( _
      New Form1())
   Catch ex As Exception
      MsgBox(ex.Message)
   End Try
End Sub
```

C0161587x.fm Page 12 Thursday, November 15, 2001 1:59 PM

Arithmetic Operator Shortcuts

All arithmetic operators in Visual Basic .NET now have shortcuts that let you operate on and assign the result back to a variable. For example, in Visual Basic 6, you might write

```
Dim myString As String
myString = myString & "SomeText"
```

In Visual Basic .NET, you can write this in a much more elegant format:

```
Dim myString As String
myString &= "SomeText"
```

These expression shortcuts apply to &=, *=, +=, -=, /=, \=, and ^=. Note that you can also use the old Visual Basic 6–style expressions.

Overloaded Functions

Visual Basic .NET introduces function overloading. With overloading, you can declare multiple functions with the same name, each accepting a different number of parameters or accepting parameters of different types. For example, suppose that you have a method that deletes a customer from a database. You might want to create two versions of the *deleteCustomer* method, one to delete a customer based on ID and one to delete a customer by name. You can do so in Visual Basic .NET as follows:

```
Sub deleteCustomer(ByVal custName As String)
   'Code that accepts a String parameter

End Sub
Sub deleteCustomer(ByVal custID As Integer)
   'Code that accepts an Integer parameter
End Sub
```

Attributes

Visual Basic .NET also now includes attributes. Attributes give one the ability to fine-tune how an application behaves. They modify the behavior of code elements and can be applied to methods, classes, interfaces, and the application itself. You can use attributes to explicitly declare the GUID for a class or to define how a variable should be marshaled when it is passed to a COM object. Suppose, for example, that you have written a common utility function that you want the debugger always to step over, rather than step into. The *DebuggerHidden* attribute allows you to do this:

```
<DebuggerHidden()> _
Function nToz(ByVal input) As Integer
   If Not IsNumeric(input) Then input = 0
   Return Input
End Function
```

C0161587x.fm Page 13 Thursday, November 15, 2001 1:59 PM

Multithreading

By default, your Visual Basic .NET applications are single threaded, but the language has new keywords that allow you to spawn new threads. This ability can be very useful if you have processes that take a long time to complete and that can run in the background. The following example creates a new thread and uses it to run the subroutine *loadResultsFromDatabase*:

```
Sub Main()
   Dim myThread As New Threading.Thread( _

   AddressOf loadResultsFromDatabase)
   mythread.Start()
End Sub
Sub loadResultsFromDatabase()
   'Some Code
End Sub
```

Reduced Programming Errors

Visual Basic .NET helps you reduce programming errors by supporting stronger type checking. For example, if you use the wrong *enum* value for a property or you assign a variable type to an incompatible type, the compiler will detect and report it at design time. With ADO.NET, you can add strongly typed datasets to your application, and if you refer to an invalid field name, it will be picked up as a compile error instead of a run-time error. These features allow you to catch errors as you write your program. For the ultimate in strong type checking, you can use *Option Strict On* in your application, which prohibits late binding and ensures that you use conversion functions whenever you assign a variable from one type to another. This feature can be useful in applications that don't use late binding, but it enforces a stricter and more verbose coding standard than many developers are used to.

The .NET Framework

In addition to the familiar VBA library of functions, such as *Left$*, *Right$*, and *Command$*, and the Win32 APIs, Visual Basic .NET has access to the .NET Framework, which is designed specifically for Visual Basic, C#, and the other .NET languages. The .NET Framework is a collection of more than 3800 classes and 28,000 methods for forms, graphics, XML, Internet development, file access, transactions, and almost everything else you can think of. The set of .NET classes you will most likely become familiar with first is the new forms package—Windows Forms—which looks a lot like the familiar Visual Basic 6 forms but which is implemented as a set of .NET classes available to all languages. The next section describes the features of this package.

C0161587x.fm Page 14 Thursday, November 15, 2001 1:59 PM

Windows Forms

Windows Forms is a new forms development system that replaces the old Visual Basic forms. Along with features that make it powerful and easy to use, Windows Forms is available to all Visual Studio .NET languages.

Faster Development

Windows Forms in Visual Basic .NET has several features that speed up development. An in-place menu editor and visual editing of tab orders make form design easier. Control anchoring allows you to remove all your old resizing code and instead visually anchor the controls so that they remain a fixed length from the edge of the form and resize whenever the form resizes. Visual inheritance allows you to inherit the controls, properties, layout, and code of a base form. A good use for this feature is to define a standard form layout for an application, with a standard size, header, footer, and close button. You can then inherit all forms from this standard form.

GDI+

GDI+ allows you to add rich visual effects to your application. For example, to make a form semitransparent, you would place the following line of code in the form load event:

```
Me.Opacity = 0.5
```

Figure 1-1 shows the effects of visual inheritance and GDI+ semitransparency in a Windows application.

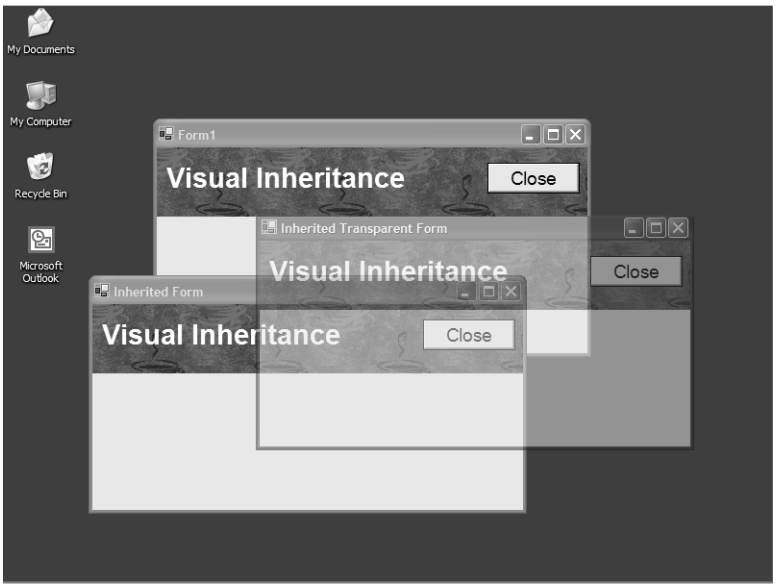


Figure 1-1 Visual inheritance and GDI+ semitransparency.

C0161587x.fm Page 15 Thursday, November 15, 2001 1:59 PM

Internationalization

Windows Forms has built-in support for internationalization. To add support for other languages, you set the *Localizable* property of the form to *True*, set the *Language* property to the desired language, and then change the *Font* and *Size* properties of the form and controls. Every change that you make is saved as specific to the current locale. For example, you can have different-sized controls with different text for both Spanish and English.

New Web Development Features

Visual Basic .NET offers many enhancements to Web development. Two of the most significant involve XML and Web Forms.

Better Support for XML

Visual Basic .NET has designers that allow visual editing of HTML documents, XML documents, and XML schemas. In addition, there are .NET Framework classes that support serializing and deserializing any .NET class to and from XML. Visual Basic .NET can create XML Web services that use HTTP to pass XML backward and forward to other applications. If your application uses XML, Visual Basic .NET has great support for it. If you're looking to add XML support to your application (or to learn how to use XML), Visual Basic .NET is a great tool for doing so.

Web Services and Web Forms

Visual Basic .NET allows you to add Web services to your application. As you will see later in this book, in many cases you can actually convert your business objects to Web services. You can also easily add a Web Forms presentation layer that leverages your Visual Basic 6 or upgraded business objects. Visual Basic .NET makes Web development as easy as Windows development.

Better Development Environment

Along with language, form, and Web development enhancements, the IDE in Visual Basic .NET has a number of new features.

Cross-Language Interoperability

Visual Basic .NET is designed for cross-language interoperability. Because .NET unifies types, controls and components written in one language can easily be used in another. Anyone who has struggled to get a Visual Basic 6 user control to work in a Visual C++ project will immediately recognize the benefits of this. You can inherit from a class written in any other language built on the .NET platform or create components in other languages that inherit from your Visual Basic classes. Visual Studio .NET provides a single environment for developing

C0161587x.fm Page 16 Thursday, November 15, 2001 1:59 PM

and compiling multilanguage applications. Using the unified debugger, you can step from a component written in one language into another. You can also debug COM+ services and step into SQL Server stored procedures.

Background Compiler and Task List

Visual Basic .NET has a compiler that continually works in the background. Compilation errors are flagged in code in real time with blue squiggle underlines. The Task List updates in real time with the list of compilation errors, and it also shows any ToDo comments you add to your code. ToDo comments are a great way to keep track of what you still need to do. For example, if you add a button to a form, and plan to finish the click code later, you can add a comment like

```
'TODO: Finish the code later
```

and the statement appears in the Task List. You can filter the Task List by task type and even choose what sort of comments are shown using the Task List pane. Figure 1-2 shows the Visual Basic .NET Task List.

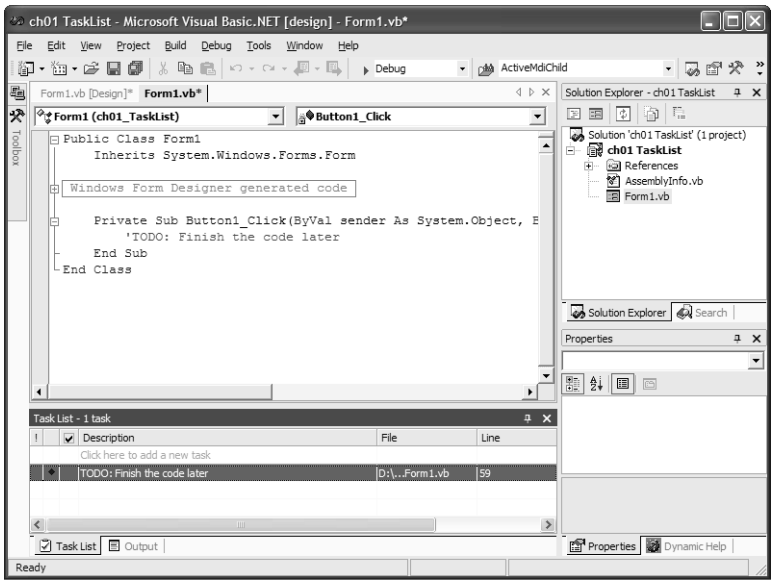


Figure 1-2 ToDo comments in the Task List.

Is Visual Basic Still the Best Choice for Visual Basic Developers?

If you are faced with learning the new features of Visual Basic .NET, you may ask yourself, "Why should I stick with Visual Basic? Why don't I choose another language instead? Why not move to C# (a new language in Visual Studio .NET

C0161587x.fm Page 17 Thursday, November 15, 2001 1:59 PM

derived from C++)?" Although the choice is yours, we should point out that Visual Basic .NET is now as powerful as C#, Visual C++, or any other language. All the .NET languages have access to the same .NET Framework classes. These classes are powerful, and they allow Visual Basic .NET to smash through the "glass ceiling" of previous versions.

Visual Basic .NET also keeps the spirit of Visual Basic alive. It is designed by Visual Basic developers for Visual Basic developers, whereas a language like C# is designed for C developers. Each language has some unique features that are not available in other languages. Let's compare C# with Visual Basic to see what makes each language unique.

Features Found in C# but Not in Visual Basic .NET

The C# language supports pointers. Pointers allow you to write "unsafe" code that modifies memory locations directly. The following code shows how to use pointers in C#:

```
unsafe static void Main(string[] args)
{
   int myInt = 5;
   int * myptr =  & myInt;
   * myptr = 55;
   Console.WriteLine(myInt.ToString() );
}
```

Although Visual Basic .NET doesn't support pointers in the language itself, as you will see in Chapter 16, you can still access memory locations using methods of the .NET *Garbage Collector* class. C# also supports document comments that allow you to embed self-documenting comments in your source code. These comments are compiled into the metadata of the component and can be extracted and built into help files.

Features Found in Visual Basic .NET but Not in C#

Visual Basic .NET's most appealing feature is the human-readable Visual Basic language, which is case insensitive with great IntelliSense. If you declare a variable as *MyVariable* and then later change the case to *myVariable*, Visual Basic .NET automatically changes the case of all occurrences in the code. C# and Visual C++ don't do this. In fact, C# treats *MYVariable* and *myVariable* as two separate variables. Most Visual Basic programmers have grown to know and become comfortable with case-insensitive behavior and will find Visual Basic .NET the most natural language to use.

Visual Basic .NET also supports late binding and optional parameters. C# supports neither of these. In addition, Visual Basic supports automatic coercions between types. For example, in C#, you cannot assign a *Long* to an *Integer* without using a conversion method (since it may cause an overflow). Visual

C0161587x.fm Page 18 Thursday, November 15, 2001 1:59 PM

Basic allows narrowing conversions like this one, since in most cases overflows don't occur. If you want to prevent automatic coercions, you can use a new compiler option, *Option Strict On*, that enforces C-like strict type coercion.

Visual Basic has richer IntelliSense and better automatic formatting than any other language. It automatically indents code, corrects casing, and adds parentheses to functions when you press Enter.

The result is a true Visual Basic experience, enhanced by background compilation as you type. For example, if you misspell the keyword *Function*, as in

```
Funtion myFunction
```

as soon as you move off the line, the compiler parses it and puts a compiler error in the Task List. It also underlines the word "Funtion" with a blue squiggle indicating the location of the compile error. As soon as you correct the line, the compiler removes the Task List item and erases the underline. Background compilation helps you write better code and is unique to Visual Basic—no other language has a background compiler.

The language you use is a matter of choice. However, if you enjoy programming in Visual Basic, you will find Visual Basic .NET a great experience and the best upgrade ever.

Conclusion

Whew—it's time to breathe. We've covered a lot of ground in this chapter. First we established that Visual Basic .NET is not 100 percent backward compatible with Visual Basic 6. We then took a lightning tour of the history of Visual Basic and saw that, although it is redesigned and restructured, Visual Basic .NET is part of the natural progression of Visual Basic. We looked at some of the differences between Visual Basic 6 and Visual Basic .NET and discussed some of the new features you can add to your upgraded applications. We also covered how you can add value to your upgraded applications and why you should continue to use Visual Basic.

The next chapter takes a deeper look at what the .NET platform is and outlines the significant differences in Visual Basic .NET. Later chapters go further into the upgrading options and describe what you can do to prepare your application for the upgrade to Visual Basic .NET. If you feel like jumping straight into upgrading, you may want to skip to Part II, which starts with a walkthrough of upgrading an application. Welcome to Visual Basic .NET, the future of Visual Basic.