SYBEX Sample Chapter

Visual Basic® .NET! I Didn't Know You Could Do That...™

Matt Tagliaferri

Chapter 1: From VB6 to VB.NET

ISBN: 0-7821-2890-4

From VB6 to VB.NET

1 Using the New Operators

The new operator code can be found in the folder prjOperators.

Visual Basic has always been a bit behind the curve in its use of operators. Fortunately, the .NET Framework has allowed Microsoft to easily make some old shortcuts as well as some new operators available to the VB programmer.

Operator Shortcuts

Borrowing from the C family of languages, you can now shorten the line of code

```
x = x + 1
```

with the following

```
x += 1
```

Most of the other basic operators work the same way, as shown in the following table:

Operator Shortcut	Short For	Meaning
x += y	**x = x + y**	**add y to x and put result in x**
x -= y	**x = x - y**	**subtract y from x and put result in x**
x *= y	**x = x * y**	**multiply y by x and put result in x**
x /=y	**x = x / y**	**divide x by y and put result in x**
x \= y	**x = x \ y**	**divide x by y and put result in x (integer divide)**
x ^= y	**x = x ^ y**	**raise x to the y power and put result in x**
x &= y	**x = x & y**	**concatenate y to x and put result in x (string)**

All of the operators shown in the table are arithmetic operators, with the exception of the string concatenation operator &.

Bitwise Operators

Visual Basic has never had operators for performing bitwise functions—until now, that is. The following table shows the three bitwise operators available in VB.NET.

Operator	Short For	Meaning	Example	Result
And	Bitwise And	Both left and right side of operator are 1	1 And 0	0
Or	Bitwise Inclusive Or	Either left or right side of operator is 1	1 Or 0	1
Xor	Bitwise Exclusive Or	Either left or right side of operator is 1, but not both	1 Xor 0	1

As a refresher, the following table shows the four possible combinations of left and right sides of bitwise operators and the result of each:

Left	Right	Bitand	Bitor	Bitxor
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Still Missing

The following lists some operators that you might be familiar with in other languages but that still haven't made their way into Visual Basic yet:

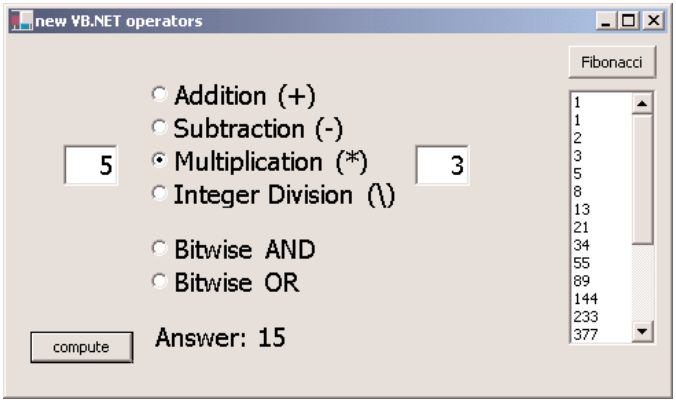
Mod Shortcut Many languages use % as a shortcut for the modulus (remainder) operator and then use x %= y as a shortcut for taking the remainder of x divided by y and putting the result back in x. The Visual Basic modulus operator is still "mod", and there is no corresponding operator shortcut.

Bitwise Shift There are still no operators for shifting a set of bits left or right.

Postfix increment/decrement The C language family allows you to write x++, which is short for x = x + 1, or x—, which is short for x = x - 1. These operator shortcuts are not available in Visual Basic. (One wonders why x += y was borrowed from C, but not x++.)

Using the Operators

The example program (illustrated here) shows all of the new Visual Basic arithmetic operators in action:



It is divided into two sections. The left side of the program is a rudimentary calculator that takes the integer values entered into two text box controls and performs an operation on them, depending on the radio button selected. The code that determines what operation to take is shown here:

```
Private Sub cbCompute_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles cbCompute.Click

        Dim iValueA As Integer
        Dim iValueB As Integer

       'exception handlers catch user putting
       'non-numbers in text boxes
        Try
            iValueA = CInt(tbA.Text)
```

```
        Catch
            tbA.Text = "0"
            iValueA = 0
        End Try

        Try
            iValueB = CInt(tbB.Text)
        Catch
            tbB.Text = "0"
            iValueB = 0
        End Try

        If rbPlus.Checked Then
            iValueA += iValueB          'this is short for
iValueA = iValueA + iValueB.
        ElseIf rbMinus.Checked Then
            iValueA -= iValueB
        ElseIf rbTimes.Checked Then
            iValueA *= iValueB
        ElseIf rbDiv.Checked Then
            Try
                iValueA \= iValueB
            Catch eErr As Exception
                Call MsgBox(eErr.ToString)
            End Try
        ElseIf rbAnd.Checked Then
            iValueA = iValueA And iValueB
        ElseIf rbOR.Checked Then
            iValueA = iValueA Or iValueB
        End If

        lbAnswer.Text = "Answer: " & iValueA
    End Sub
```

The procedure makes use of exception handling to make sure that numeric values are entered in the text boxes (zeros are used as the operands if nonnumeric values are supplied) and to trap any divide-by-zero errors that might occur. The rest of the routine merely checks which radio button is checked and performs the correct operation on the two numbers.

The second part of the program generates the beginning of the Fibonacci sequence of numbers and displays the results in a `Listbox`:

```
Private Sub cbFib_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles cbFib.Click

        Dim i As Integer = 1
        Dim j As Integer = 1
        Dim t As Integer
        Dim iCtr As Integer = 0
        Dim arList As New ArrayList(20)

        arList.Add(i)
        arList.Add(j)

        For iCtr = 0 To 20
            t = i               'save i
            i += j              'add j to i
            j = t               'put save i into j
            arList.Add(i)       'add result to arraylist
        Next

        lbFib.DataSource = arList   'bind arraylist to listbox
    End Sub
```

This procedure makes use of the `ArrayList` class to store the integers and then binds the `ArrayList` to the `Listbox` in the last line. The idea behind the Fibonacci sequence is to start two variables at value 1, add them together, and store the result back into one of the variables. You then repeat this process as long as desired. The previous sample generates the first 21 values in the sequence.

2 New Tricks in Variable Declaration

The variable declaration code can be found in the folder prjVariables.

Usually, a book in this format might not cover something as rudimentary as variable declaration in a programming language. However, Visual Basic.NET has quite a few significant differences in its base data types and variable declaration syntax. These differences bear discussion, because not knowing about them can cause anything from temporary confusion to a hair-pulling bug or two.

Integer Type Changes

The first major change you need to be aware of is that an *Integer* is not an `Integer` anymore (huh?). Likewise, a *Long* is not a `Long`, either. In previous versions of Visual Basic, a variable declared as an `Integer` gave you a 16-bit variable with a range from –32768 to +32767. In VB.NET, an `Integer` is a 32-bit variable with a range from about negative to positive 2 million. In other words, it's what you used to call a `Long`. A variable declared in VB.NET as a `Long` is now a 64-bit integer. So, where did the 16-bit integer go? That's now called a `Short`. Here's a quick translation table:

What You Used to Call	Is Now Called
Integer	**Short**
Long	**Integer**
Really big 64-bit number that I can't define	**Long**

Why in the name of Sweet Fancy Moses did Microsoft change the integer type names in what seems to be the most confusing way imaginable? There's a good reason, actually. The answer lies in the fact that the .NET platform is Microsoft's attempt to bring all (or most, anyway) of their programming languages under a single runtime umbrella: the .NET Framework. One problem in attempting this was that Microsoft's C++ and Visual Basic languages did not use a common naming system for their data types. So, in order to unify the naming system, some changes had to be made in one or the other of the languages, and we VB programmers were chosen to take on the challenging task of learning a new naming convention (because of our superior intelligence, naturally).

If the new integer naming scheme is simply too much for you to keep track of, you have a nice, simple alternative, fortunately. The `Short`, `Integer`, and `Long` data types are the VB equivalents of the .NET Framework data types

`System.Int16`, `System.Int32`, and `System.Int64`. You can always declare your integer variables using these types instead. This would certainly end all confusion as to what type is what size.

Dim Statement Behaves Differently

Consider the following Visual Basic variable declaration:

```
Dim A, B, C as Integer
```

In VB.OLD, a line like this was the source of boundless confusion among programmers because the data type of variables A and B was not well defined. The intention of the programmer was probably to declare three Integer variables, but VB6 and below did not treat this line in this way. Instead, only variable C was declared as an Integer, and A and B are most likely *variants*.

VB.NET corrects this long-time confusion. The previous line behaves as God, Bill Gates, and most likely the programmer who wrote it intended it to behave: it declares three Integer variables.

You can still add each type explicitly, or you can mix types, as shown here:

```
Dim A as Short, B as Integer, C as String
```

No More Variants

The `Variant` data type has gone the way of the mastodon. Instead, the base, catch-all data type in Visual Basic.NET is the *`Object`*. The new `Object` type duplicates all the functionality of the old variant.

Personally, I was never much for using the `Variant` data type because it seemed like all I was ever doing was explicitly converting the contents of my variant variables into integers or strings or whatever in order to perform accurate operations on them. However, I find I'm already using the `Object` data type much more frequently because it's not just for holding base data types like integers and strings, but also for holding actual class instance types like Buttons, Forms, or my own invented classes.

Initializers

Initializers are a cute new feature that let you declare and initialize a variable in the same line, as in these examples:

```
Dim X as Integer = 0
Dim S as String = "SomeStringValue"
Dim B as New Button()
Dim A(4) As Integer = {0, 10, -2, 8}
```

The first two declare and initialize simple data types to default values. The third line is a holdover from prior versions of VB—it declares an object of type button and instantiates it in the same line. The last line creates an array of four integers and sets the initial values of all four elements in the array.

NOTE Arrays in Visual Basic.NET are always zero-based arrays. The Option Base statement is no longer supported.

Local Scope

A variable can now be declared inside a statement block such as an `If` or `Do While` statement, and the variable will have scope only within the block in which it is declared, for example:

```
Dim bDone As Boolean = False
Dim r As New Random()

Do While Not bDone
   Dim Y As Integer

   Y = r.Next(1, 100)
   bDone = (Y < 10)
Loop

Call Console.Writeline("Final value=" & Y)
```

This block of code will not compile properly because the declaration of `Y` is inside the `Do While` block, but the `Console.Writeline` attempts to access

it. Since the `Console.Writeline` is outside the scope of the loop, the variable is also out of scope.

Most programmers might combat the potential for these local scope errors by putting every `Dim` statement at the top of the procedure or function. This can lead to an inefficient use of resources, however. Consider the following code fragment:

```
If not UserHasAlreadyRegistered() then
  Dim f as New RegistrationForm()
  f.ShowDialg
end if
```

In this code, some magic function goes off and checks if the program has already been registered. If it has not, then an instance of the registration form is declared and shown. If the user has already registered the software, why bother creating an instance of a form that will never be displayed? All this does is clog up the garbage collector later. As you can see, clever use of local scope variable can save your program memory, making it run more efficiently.

3 Avoiding Redundant Function Calls

The redundant function calls code can be found in the folder prjRedundantFunctionCalls.

This little coding shortcut seems so obvious that I almost didn't consider it worth inclusion in the book, but I see this rule broken so frequently that I felt it worth repeating. The rule, in its most basic form, is as follows:

> *Why execute code more than once when running it once gives the same result?*

To illustrate the rule with an absurd example, consider the following block of code:

```
For X = 1 to 1000
  Y = 2
Next
```

This loop assigns the value 2 to variable Y, one thousand times in a row. Nobody would ever do this, would they? What's the point? Since no other code executes in the loop except for the assignment statement, you know that nothing could possibly be affecting the value of Y, except the assignment statement itself.

When the previous loop is complete, Y has the value of 2. It doesn't matter if this loop runs one thousand times, one hundred times, or simply once—the end result is the same.

While I've never seen code quite as worthless as this, the following block of code is very close to one that I read in a Visual Basic programming article a while back:

```
Do While instr(cText, "a") > 0
      cText = Left(cText, instr(cText, "a") - 1) & _
             "A" & mid(cText, instr(cText, "a") + 1)
Loop
```

This code scans through the contents of a string variable and replaces all of the lowercase letter *a*'s with uppercase *A*'s. While the function performs exactly what it's intended to perform, it does so in a very inefficient manner. Can you detect the inefficiency?

A Simple Speedup

To determine what rankled my feathers so much about this block of code, you need to think about how long it takes your lines of code to run. All Visual Basic lines of code are not created equal in terms of the length of time they take to execute. Take the `instr` function, for example. The `instr` function scans through a string looking for the occurrence of a second string. Imagine that you had to write a Visual Basic replacement for the `instr` function. You would start at the beginning of the string, compare it to the comparison string, and keep looping through each character until you either found the comparison string, or got to the end of the original string.

The `instr` function built into Visual Basic probably does the same thing, albeit in some optimized fashion. However, you don't get anything for free. If you call `instr`, Visual Basic internally loops through the test string looking for the comparison string. This loop is going to take some finite amount of time (a very small amount of time, to be sure, but a finite

amount, nonetheless). Following my rule, why would you want to run this loop more than once when running it once gives the same result?

The previous tiny little block of code calls the exact same `instr` function three times every time the loop is iterated. If you assume that the `instr` call itself runs as I surmise (some linear search through the input string), the `instr` call will take longer to run on bigger input strings (because the code has to loop through every character in the string). What if the input string to the loop was the entire contents of all the books in the Library of Congress? Let's say, for the sake of argument, that the `instr` call takes one minute to run on a string as large as the entire contents of the Library of Congress. Since I call the `instr` call three times, the loop will require (at least) three minutes for every iteration of the loop. Multiply that by the number of *A*'s found in the Library of Congress, and you'll have the total operating time of the loop.

If I make a simple change to the loop, I can reduce the number of `instr` function calls from three to one:

```
iPos = instr(cText, "a")
Do While iPos > 0
   cText = Left(cText, iPos - 1) & "A" & mid(cText, iPos + 1)
   iPos = instr(cText, "a")

Loop
```

The change I made was to store the result of the `instr` function call into a variable and to use that variable in the first line of the loop, where the lowercase *a* is replaced by an uppercase *A*. The loop result is the same, but the `instr` function is called only once per loop iteration.

Does a change like this really make a difference in speed? The example program proves the difference. The program creates a large string of random letters (with spaces thrown in to make them look a bit more like words) and then runs through one of the previous loops to replace all of the lowercase *a*'s with uppercase *A*'s. The "fast" loop (one `instr` call per loop iteration), runs at about 75 percent of the speed of the "slow" loop (three `instr` calls per loop iteration). A 25 percent speed savings is considered quite good. If a loop of this type were called repeatedly in your application, a 25 percent speed increase might make your application feel faster to the end users. I've learned that the feel of an application is of primary importance to the end user—if the program feels slow, the user might not use the application.

NOTE The example program shows a brief example of random number generation in Visual Basic. A class called `Random` is included in the .NET Framework that handles all types of random number generation. The `Random` class contains methods for generating floating point random numbers between 0.0 and 1.0 or between a numeric range. See the example program function named `RandomBigString` for some sample uses of the `Random` class.

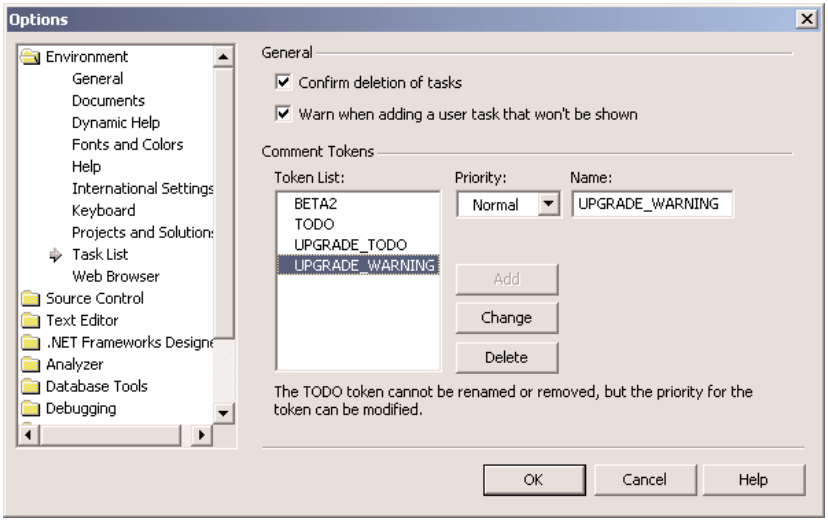
4 The Visual Studio "HoneyDo" List

The Task List code can be found in the folder prjDataset.

At my home, as in many homes, I'm sure, we have what we call a "HoneyDo" list—a list of outstanding jobs around the house for me to do. These jobs range in size from small things like sweeping out the garage or putting up some shelves to larger tasks like removing wallpaper or staining the deck. Sometimes, I'll be working on one chore that reveals a second—like when I pull up old carpet in the basement only to reveal some rust-stained concrete underneath. Or when I discover a hole created by chipmunks while cleaning out the garage. It never ends.

When things like this happen, I often don't have time to get to the second job in the same day (the ballgame awaits, after all…). Instead, I add it to the HoneyDo list, complete the first job, and get back to the second job another day. Visual Studio.NET has a feature much like the HoneyDo list (except that it doesn't call me "honey"—good thing): the Task List. The Task List is similar to that found in Outlook, or even previous versions of Visual Studio, with one important distinction: you can auto-fill Task List entries with specially constructed comments. Let's look at how this works.

Task List categories are set up under the Tools ➢ Options dialog. The Task List settings are under the Environment category, as shown in the next illustration.



NOTE As you can see in the illustration, I created a `BETA2` token that I used throughout the development of this book. Whenever something wasn't working in VS.NET beta 1 and I suspected that the problem might be because the language was an early beta, I left myself a note to recheck the problem once I received VS.NET beta 2.

You can modify the entries under the Tokens list. A token is a special phrase with which you can begin a comment. If you do begin a comment with one of the predefined tokens, an entry is automatically added to the task list. The text of the task is the text of the comment. This code snippet shows a comment entered into the sample project:

```
' TODO - replace connection object later
Dim aConn As New SQLConnection(CONNECTIONSTRING)
```

Because the comment begins with the `TODO` token, a task is automatically placed into the Task list, as shown here:

Once the comment is set up in this way, you can double-click the item in the Task List and it will zoom your code directly to the corresponding comment. Deleting the comment deletes the task in the Task List. This functionality acts as the HoneyDo list for your project. You can set up open tasks as comments and they'll show up in the Task List. Using different tokens allows you to group tasks under different categories and priorities.

5 Delving into Docking and Anchoring

The docking and anchoring code can be found in the folder prjAnchors.

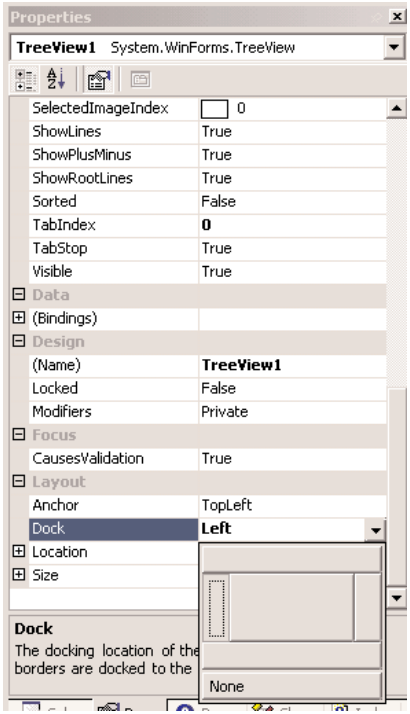
Finally, finally, finally! I am so tired of writing code to resize controls on a form. How many third-party auto-resizer VBXs and OCXs and ActiveX controls have been put on the commercial and freeware market? Being the type of person who would only use a third-party control when its functionality couldn't be duplicated with good old VB code, I never used one of these controls. Instead, I used to spend an hour writing silly little snippets of code in the Resize event of my VB forms to do things like:

- Making sure the Treeview got longer as the form did
- Making sure the grid got wider as the form did
- Keeping the OK and Cancel buttons near the bottom of the form

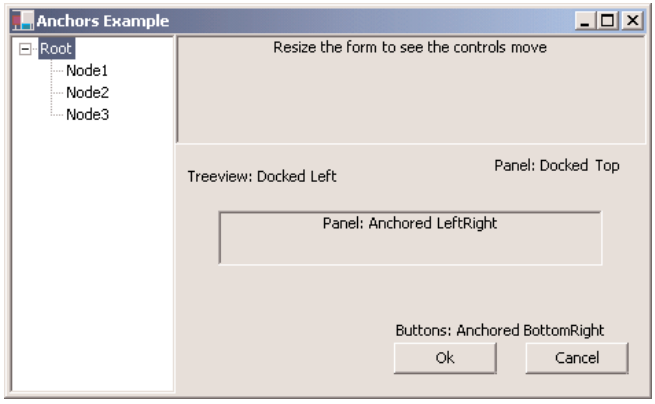
Visual Basic GUI components finally have two properties that save me from having to write this kind of time-wasting code ever again. These are called the `Dock` and `Anchor` properties (any reason why they chose two maritime references?).

The `Dock` property can be set to one of the following values: `None` (the default), `Top`, `Left`, `Right`, `Bottom`, or `Fill`. Setting the property to `None` causes the control to stay right where you put it on the form. A setting of `Top`, `Left`, `Bottom`, or `Right` causes the control to remain attached to that side of the parent of the control. Setting these properties in the Visual

Studio Property Editor is done with a little graphical representation, as shown here:



In the sample project, the Treeview is set with a `Dock` of `Left`, so it remains attached to the left side of its parent, which is the main form. The control `lbDirections` is set with a `Dock` of `Top`, which causes it to remain docked with the top of its parent, which is the upper-panel control. The following illustration shows a picture of the project while it's running:

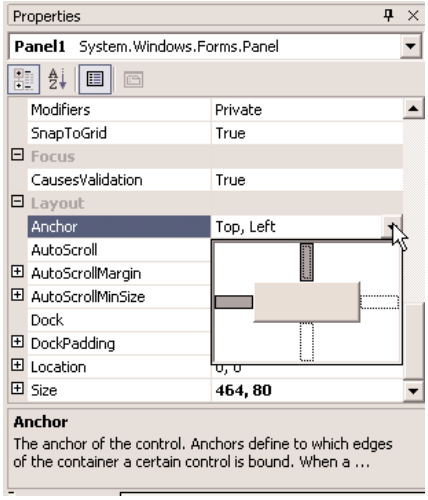


Docked controls grow appropriately if the edges of the parents to which they are docked grow in the following manner:

- A control with a `Dock` set to `Left` or `Right` grows in height as its parent grows in height.
- A control with a `Dock` set to `Top` or `Bottom` grows in width as its parent grows in width.

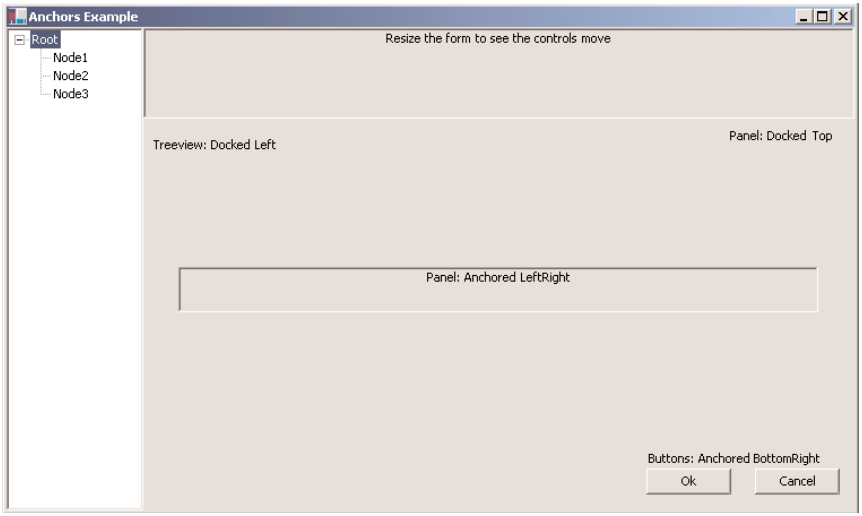
The `Anchor` property is somewhat similar to the `Dock` property, but the control doesn't attach itself directly to the edge of the form. Instead, its edges maintain a constant distance to the edges defined by the property.

Setting the `Anchor` property is also done graphically, as shown in this illustration:



The available settings are some combination of `Top`, `Left`, `Bottom`, and `Right`. The default `Anchor` value is `Top,Left` meaning that the control's top and left side will remain a constant distance from the top and left edges of its parent. If you were to set a control to `Left,Right` the left and right edges would stay anchored to the left and right edges of the form—meaning that the control would have to resize as the form was resized. The lowermost panel in the sample project has an `Anchor` property of `Left,Right` so you can see it resize as the form is resized and it maintains its left and right anchors.

The last illustration shows the same project with the form made both taller and wider. Note how all of the controls on the form have fallen into line without a single line of code!



Looking at the illustration should give you a pretty good idea of the `Dock` and `Anchor` properties in action, but things should really click into place when you run the provided project. Watch all of the controls conform to their `Dock` and `Anchor` properties as you resize the form.

6 Beyond the Tag property

The Tag property code can be found in folder prjCustomTreeNode.

"What? No `Tag` property? Why would they remove that? I use that property in at least 100 different ways. What the heck am I supposed to do now?"

The hue and cry came from all directions when it was learned that Microsoft had removed the `Tag` property from all of their controls in the .NET Framework. That `Tag` property serves as a catch-all property to store user-defined data. It originally started as an `Integer` property, but changed over to a `String` property to meet user demand.

People found myriad uses for the `Tag` property. For example, suppose you were populating a Treeview with the names of employees from a corporate database for the purposes of creating an org chart. While loading each employee into a `TreeNode` object on the Treeview, you could store the primary key for each employee (be it the social security number, a GUID, or some other unique identifying element) into the `Tag` property on the `TreeNode`. Then, when the application user selected a `TreeNode` in the Treeview, you would have instant access to the primary key of the table from which you loaded these employees. This would allow you to query the database to return additional information about the employee (date of birth, service time, current title, and so on).

Along Came Beta 2

I guess Microsoft actually heard the developer screams when they attempted to remove the `Tag` property. As of Visual Studio.NET beta 2, they actually put the user-defined property back, as a member of the `Control` class. Apart from almost rendering this part of the book useless, all Microsoft did was anger the *other* developers, the ones who liked the reasoning behind the removal of this property to begin with. These developers argue that we really don't need Microsoft to give us a property for supplying user-defined data, because the object-oriented features of VB.NET make it really easy (almost trivial, really) to add user-defined properties ourselves. I happen to fall into this camp. I submit that by removing the `Tag` property, Microsoft is actually taking away a crutch that might

prevent you from using object-oriented techniques and therefore not use the new language in the way in which it was intended.

Furthermore, having a `Tag` property on every single component can add up to a great deal of overhead. Do you really need a `Tag` property on every label and button on every form in your application? Perhaps, but probably not. Why have properties on controls that you'll never use? In the long run, it's better to run with stripped down versions of all the controls and use other tools to bolt new things on the side as you need them. This is a core component of object-oriented programming.

To demonstrate the power of using object-oriented programming, I'll take an existing component and bolt a few new properties onto it. In this example, the goal is to load up a Treeview with a list of files on the machine's hard drive. When the user clicks one of the nodes in the Treeview, I would like the program to display the date and size of that file.

There are two basic ways I can solve this problem. The first way is to wait until the user clicks a file in the Treeview, then go back to the file system to load the file date and time and display it. I decided this method might be a bit difficult to implement, mainly because my Treeview node isn't going to have the filename with its complete path on each node. I would probably have to iterate through the parents of the node to reconstruct the full path of the file.

Instead, I decided that it would be much easier to store the date and time of each file somewhere as I was iterating through the file system and loading the file names into the Treeview. The only question was where to store these date and time variables. Since I needed a date and time variable for each file I was going to load into the Treeview, it made sense to bolt these variables onto the `TreeNode` class, as shown here:

```
Class FilePropertitesTreeNode
      Inherits TreeNode

      Private FFileDate As DateTime
      Private FFileSize As Long

      Property FileDate() As DateTime
          Get
              Return FFileDate
```

```
            End Get
            Set
                FFileDate = Value
            End Set
        End Property

        Property FileSize() As Long
            Get
                Return FFileSize
            End Get
            Set
                FFileSize = Value
            End Set
        End Property

    End Class
```

The class is called `FilePropertiesTreeNode`. It inherits off of the base `TreeNode` class, found in the `System.Windows.Forms` namespace. The purpose of the class is to add two additional properties to the standard `TreeNode`. These properties store a date and a number representing the size of a file.

The intention is to use these new `TreeNodes` instead of the standard `TreeNode` when filling a Treeview with file/directory information. While loading the Treeview, I can put the date and time of each file in these new properties, thus giving me easy access to them as a node is selected in the Treeview. I could easily create more properties that further describe each file, such as hidden/read-only attribute information, the file extension, the bitmap associated with this file type, and so on.

Using an Inherited Class

To use your custom inherited `TreeNode` instead of the base `TreeNode`, you merely create an instance of your new class and add it to the Treeview using the same `Add` method you would normally use. The `Add` method takes a `TreeNode` as its parameter—this includes `TreeNode` objects or direct descendants of `TreeNode` objects, like my `FilePropertiesTreeNode`. Here is

some example code to add one of our new TreeNodes to a Treeview named tvStuff

```
oNode = New FilePropertitesTreeNode()
oNode.Text = "C:\WINDOWS\SOMEDUMMYFILE.TXT"
oNode.FileDate = "Jan 1, 2001"
oNode.FileSize = 65536
tvStuff.Nodes.Add(oNode)
```

Of course, the file information just listed is all made up. What would be more useful would be to load actual filenames off disk and store their properties in the new TreeNode class instances. This would be the first step in writing a Windows Explorer–like program. The sample project prjCustomTreeNode does just that. It fills a Treeview with instances of my new FilePropertiesTreeNode class, reading files on the C drive as the source of the file information. The main recursive function that loads the Treeview is listed here:

```
Protected Sub FillTreeView(ByVal cFolder
As String, ByVal oParentFolder As FilePropertitesTreeNode,
ByVal iLevel As Integer)

    Dim d As DirectoryInfo
    Dim f As FileInfo
    Dim o As Object
    Dim oFolder As FilePropertitesTreeNode
    Dim oNode As FilePropertitesTreeNode
    Dim cName As String

   'for this demo, we're only going
   '3 levels deep into the file structure
   'for speed reasons
    If iLevel > 3 Then Exit Sub

    d = New DirectoryInfo(cFolder)
    cName = d.Name

    'fix the entry 'C:\', so we don't
    'have double \\ in filenames

    If cName.EndsWith("\") Then
        cName = cName.Substring(0, cName.Length - 1)
    End If
```

```
'create node for this folder
oFolder = New FilePropertitesTreeNode()

'fill the custom properties
oFolder.Text = cName
oFolder.FileDate = d.LastWriteTime

'add this node. May have to add to Treeview
'if no parent passed in
If oParentFolder Is Nothing Then
    tvFileListing.Nodes.Add(oFolder)
Else
    oParentFolder.Nodes.Add(oFolder)
End If

Try
    For Each f In d.GetFiles()

        oNode = New FilePropertitesTreeNode()
        'set up folder
        oNode.Text = f.Name

        'fill in our custom properties
        oNode.FileDate = f.LastWriteTime
        oNode.FileSize = f.Length
        'add this node
        oFolder.Nodes.Add(oNode)
    Next

    For Each d In d.GetDirectories
        Try
            Call FillTreeView(d.FullName, oFolder, iLevel + 1)

        'catch errors, like access denied
        'errors to system folders
        Catch oEX As Exception
            Console.WriteLine(oEX.Message)
        End Try
    Next
Catch e As Exception
    Console.WriteLine(e.Message)
```

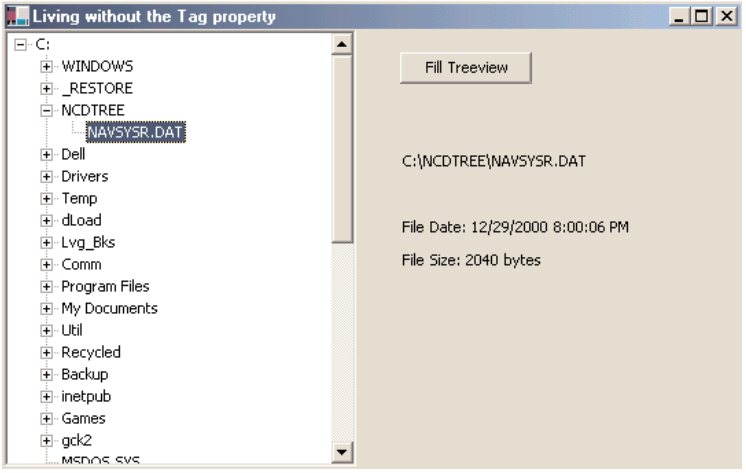
```
        End Try

    End Sub
```

The procedure expects a folder name as its first parameter. It creates an instance of a `DirectoryInfo` object based on this folder name. The `DirectoryInfo` object returns useful information like the name of the directory and the last time it was written to. It also contains methods for looping through all of the structures inside it.

The first step is to create a `FilePropertiesTreeNode` and add it as a child to the passed-in parent node, also a `FilePropertiesTreeNode`. This routine has a depth tester that makes sure that the routine stops loading after four levels of depth in the file system. This is done only as an optimization, so the load routine takes a shorter amount of time.

There are two `For...Each` loops in the routine—the first loops through all the subdirectories in the current directory, and the second loops through all the files in the directory. For each subdirectory, the same procedure is recursively called against the new subdirectory name. For each file, one of the `FilePropertiesTreeNode` instances is created, loaded with the file date and time information, and added to the parent (folder) node.



Once the Treeview is filled, the `OnAfterSelect` event is set up so that the following code runs when the user clicks on a node in the Treeview:

```
Private Sub tvFileListing_AfterSelect(ByVal sender_
As System.Object, ByVal e As_
System.Windows.Forms.TreeViewEventArgs)_
Handles tvFileListing.AfterSelect
```

```
    Dim oNode As FilePropertitesTreeNode

    oNode = CType(e.Node, FilePropertitesTreeNode)
    If Not oNode Is Nothing Then
        lbFileName.Text = oNode.FullPath
        lbDate.Text = "File Date: " & oNode.FileDate()
        lbSize.Text = "File Size: " & oNode.FileSize() &_
        " bytes"
    End If

End Sub
```

This code first returns the node that was clicked and typecasts it to our special node class (the typecast is necessary because the `Node` property on the `System.Windows.Forms.TreeViewEventArgs` object is of the normal `TreeNode` class). If the typecast is successful, some labels are filled with the contents of the custom `FileDate` and `FileSize` properties.

NOTE When I finally got Visual Studio.NET beta 2 installed on my machine, I thought I'd have to throw this part of the book away because Microsoft decided to put the `Tag` property back into the language. As it turns out, though, this example project is still quite valid. Because the sample code adds properties to a `Treenode` class, and because the `Treenode` class is not a descendant of the `Control` class, I wouldn't have been able to use the `Tag` property to store my file info anyway. Now, if Microsoft decides to move the `Tag` property down to the `Object` class instead of the `Control` class, I just might have to scream…

7 Handling Control Arrays Another Way

The control array code can be found in the folder prjNoControlArrays.

From my very first days of Visual Basic, I was enamored with using control arrays. My first "real" Visual Basic program was a card game, and it seemed

a perfect solution to create an array of picture box controls with the appropriate bitmaps for playing cards. I completed my card game, uploaded it to a local BBS (this was a few years before the Internet), and received a few comments about it.

My use of control arrays didn't stop with that first card game. I must have written a half dozen card games, as well as some crossword-type games, the mandatory number scramble game, and a few other simple games that gave me fun projects to work on while I learned Visual Basic. I'll bet almost all of those early programs used control arrays to handle the game elements.

Before I got my first copy of VB.NET, I was reading an online summary of some of the language changes, and one of the differences mentioned that control arrays were no longer a feature of the language.

The main benefit of having an array of controls is, of course, being able to write the same event handling code for multiple controls and the ability to easily tell which control fired the event, as seen here:

```
Sub pnPanel_Click(Index as Integer)
  Msgbox("Panel index " & index & "was clicked")
End Sub
```

This piece of VB6 code handles the `Click` event for an array of controls named `pnPanel` and displays a message about which one was picked.

So what's a closet game programmer like me to do? If I have several similar user interface elements that I want handled all the same way and I can't group them with a control array, is there some other means to have all of these controls share the same event code? The answer is, of course, yes. Visual Basic introduces a `Handles` clause on procedures that allows you to link many event procedures to the same code. Here is an example of the `Handles` clause in action:

```
 Public Sub PanelClick(ByVal sender_
 As Object, ByVal e As System.EventArgs)_
Handles Panel1.Click, Panel2.Click, Panel3.Click,_
Panel4.Click, Panel5.Click, Panel6.Click,_
Panel7.Click,Panel8.Click, Panel9.Click

    Dim p As Panel
    p = CType(sender, Panel)
    If p.BackColor.Equals(Red) Then
        p.BackColor = Blue
```

```
            Else
                p.BackColor = Red
            End If
            p.Invalidate()
        End Sub
```

This `Click`

Again, the paint event for all nine panels is handled by this single event, in which I again typecast the sender variable to a local `Panel` variable so I can do stuff to it. I then write some custom painting code. First, I fill the panel with its defined `BackColor`, and then (just for fun), I draw a circle within the boundary of the panel.

The final effect is that clicking any of the nine panels switches their color from red to blue. You can easily see how this might be the beginning of a tic-tac-toe game or something similar: