For even more tricks and tips go to
http://www.devx.com

WELCOME TO THE EIGHTH EDITION OF THE VBPJ *TECHNICAL TIPS SUPPLEMENT!*

These tips and tricks were submitted by professional developers—*VBPJ* readers—using Visual Basic 3.0, Visual Basic 4.0, Visual Basic 5.0, Visual Basic 6.0, Visual Basic for Applications (VBA), and Visual Basic Script (VBS). The editors at *Visual Basic Programmer's Journal* compiled the tips. Instead of typing the code published here, download the tips from the free, Registered Level of The Development Exchange at http://www.devx.com.

If you'd like to submit a tip to *Visual Basic Programmer's Journal*, please send it to User Tips, Fawcette Technical Publications, 209 Hamilton Ave., Palo Alto, California, USA, 94301-2500. You can also fax it to 650-853-0230 or send it electronically to vbpjedit@fawcette.com. Please include a clear explanation of what the technique does and why it's useful, and indicate if it's for VBA, VBS, VB3, VB4 16- or 32-bit, VB5, or VB6. Please limit code length to 20 lines. Don't forget to include your e-mail and mailing address. If we publish your tip, we'll pay you your choice of $25 or a one-year extension of your *VBPJ* subscription.

VB5, VB6
Level: Beginning

PREVENT CHECKBOX CHANGES

You'll often want to display a checkbox-style listbox to show users the values they have selected in an underlying database. However, you don't want to allow users to change the selections—that is, to change which boxes they checked. You can't disable the listbox because that stops users from scrolling the list to see which items they checked. You can't use Locked, because the listbox doesn't have a Locked property.

Here's one solution: Paint a Command button with the caption "Click to toggle enabled property" and a listbox on a form, then change the listbox style to 1-Checkbox. Add this code:

```
Option Explicit
Dim mbDisabled As Boolean
Private Sub Command1_Click()
    mbDisabled = Not mbDisabled
End Sub
Private Sub List1_ItemCheck(Item As Integer)
    If mbDisabled Then
        List1.Selected(Item) = Not List1.Selected(Item)
    End If
End Sub
```

When mbDisabled is set to True, the changes made by the user to the listbox selections are immediately reversed. It will appear as if the selections haven't changed at all, and the list is still scrollable.

—Ian Champ, received by e-mail

VB4 32, VB5, VB6
Level: Intermediate

ESTABLISH A DATA DICTIONARY

If your SQL looks like this, you need to ask yourself how much code you'd have to inspect and revise if you decided to change a database field or table name, as frequently happens during development:

```
SQLString = "SELECT [first name], [last name], " & _
    "[line preferences]" & _
    " FROM [imaging users] WHERE [user code] = " & _
    "'" & Trim(UCase(UserIDText.Text)) & "'"
ODBCstatus = SQLExecDirect(ODBChandle1, SQLString, _
    Len(SQLString))
```

What happens if SQL command conventions (field name delimiters) change? Because a compile doesn't reveal such name misspellings or convention flaws, code in obscure procedures can be in production before defects are detected.

Our group established a table and field dictionary in a module used for a recent large project. This helped us ensure that we correctly pasted table and field names into all SQL commands. It also provided a repository that simplified maintenance.

As database resource names changed or new name-delimiting conventions were required, we revised the dictionary before recompiling. We also used the dictionary to convey descriptive information about tables and fields to developers. Our dictionary looks like this:

```
'tables:
Public Const tblUsers As String = "[imaging users]"
'data fields:
Public Const fldFirstName As String = "[first name]"
    '16 characters
Public Const fldLastName As String = "[last name]"
    '16 characters
Public Const fldLinePreferences As String = _
    "[line preferences]"
    '20 characters
Public Const fldUserCode As String = "[user code]"
    '10 characters
```

Our SQL looks like this:

```
SQLString = "SELECT " & fldFirstName & _
    ", " & fldLastName & ", " & fldLinePreferences & _
    " FROM " & tblUsers & " WHERE " & fldUserCode & " = " & _
    "'" & Trim(UCase(UserIDText.Text)) & "'"

ODBCstatus = SQLExecDirect(ODBChandle1, SQLString, _
    Len(SQLString))
```

Programmers don't have to know the actual names of database components. They always use the constants that *refer* to the database components. A clean compile ensures you'll use correct names and name-delimiting conventions in your SQL statements.

—Doug Hagy, Greensburg, Pennsylvania

VB4 32, VB5, VB6

Level: Intermediate

Context-Sensitive Help for Disabled Controls

If you want a form to support context-sensitive help, set the WhatsThisButton and WhatsThisHelp properties on the form to True, and set the WhatsThisHelpID property to a corresponding help-file topic ID for any control on that form for which you want help to be displayed.

Unfortunately, the help isn't shown if the control's Enabled property is set to False. To solve this problem, create a label under the control with the same dimensions, and clear its caption to make it invisible. Set the WhatsThisHelpID property to the same value as the disabled control's property.

—Frank Addati, Melbourne, Australia

VB3, VB4 16/32, VB5, VB6

Level: Intermediate

Improve on the Bubble Sort

A bubble sort's execution time is a multiple of the square of the number of elements. Because of this, the bubble sort is said to be an n-squared algorithm. You can easily make improvements to a bubble sort to speed it up.

One way is to reverse the direction of passes reading the array, instead of always reading the array in the same direction. This makes out-of-place elements travel quickly to their correct position. This version of a bubble sort is called the shaker sort, because it imparts a shaking motion to the array:

```
Public Sub Shaker(Item() As Variant)
     Dim Exchange As Boolean
     Dim Temp As Variant
     Dim x As Integer
     Do
          Exchange = False
          For x = (UBound(Item)) To (LBound(Item) + 1) Step -1
               If Item(x - 1) > Item(x) Then
                    Temp = Item(x - 1)
                    Item(x - 1) = Item(x)
                    Item(x) = Temp
                    Exchange = True
               End If
          Next x
          For x = (LBound(Item) + 1) To (UBound(Item))
               If Item(x - 1) > Item(x) Then
                    Temp = Item(x - 1)
                    Item(x - 1) = Item(x)
                    Item(x) = Temp
                    Exchange = True
               End If
          Next x
     Loop While Exchange
End Sub
```

Although the shaker sort improves the bubble sort, it still executes as an n-squared algorithm. However, because most programmers can code a bubble sort with their eyes closed, this is a nice way to shave 25 to 33 percent off the required execution time without having to dig out the algorithm books. Still, you don't want to use either a bubble or shaker sort for extremely large data sets.

—Tan Shing Ho, Kuala Lumpur, West Malaysia

VB4 32, VB5, VB6

Level: Intermediate

Slam Selected Items Into an Array

Use this code to retrieve all selected list items in a multiselect-style listbox in one API call. It's a lot easier than iterating through a large list using For…Next. This code works against both normal and checkbox-style lists:

```
Private Declare Function SendMessage Lib "user32" Alias _
     "SendMessageA" (ByVal hWnd As Long, ByVal wMsg _
     As Long, ByVal wParam As Long, lParam As Any) As Long
Private Const LB_GETSELCOUNT = &H190
Private Const LB_GETSELITEMS = &H191
Private Sub Command1_Click()
     Dim numSelected As Long
     Const LB_ERR = -1
     Dim r As Long
     Dim i As Integer
     'get the number of items selected.
     'If the listbox is single-select style,
     'numSelected will return -1 (LB_ERR).
     'If the listbox is multiselect style,
     'and nothing is selected, numSelected
     'returns 0. Otherwise, numSelected returns
     'the number selected (ala List1.SelCount)
     numSelected = SendMessage(List1.hWnd, LB_GETSELCOUNT, _
          0&, ByVal 0&)
     'debug ...
     Debug.Print numSelected; " items selected:"
     Debug.Print "index", "item"
     If numSelected <> LB_ERR Then
          'dim an array large enough to hold
          'the indexes of the selected items
          ReDim sSelected(1 To numSelected) As Long
          'pass the array so SendMessage can fill
          'it with the selected item indexes
          Call SendMessage(List1.hWnd, LB_GETSELITEMS, _
               numSelected, sSelected(1))
          'debug ...
          'print out the items selected
          'note that their index is 0-based
          For i = 1 To numSelected
               Debug.Print List1.List(sSelected(i))
          Next
     End If
End Sub
```

—Randy Birch, East York, Ontario, Canada

VB5, VB6

Level: Intermediate

Call Up Windows Shell Features

Here's a little routine that provides a quick and dirty way to call up some of the more oddball features of the Windows shell. It works by emulating user keystrokes, so you'll need to modify the keys for non-English versions. Simply paste this code into a standard module and pass the Enum of choice:

```
Private Declare Sub keybd_event Lib "user32" (ByVal bVk As _
     Byte, ByVal bScan As Byte, ByVal dwFlags As Long, _
     ByVal dwExtraInfo As Long)
Public Enum SystemKeyShortcuts
     ExplorerNew = &H45    ' Asc("E")
     FindFiles = &H46      ' Asc("F")
     MinimizeAll = &H4D    ' Asc("M")
```

```
	RunDialog = &H52	' Asc("R")
	StartMenu = &H5B	' Asc("[")
	StandbyMode = &H5E	' Asc("^") -- Win98 only!
End Enum
Public Sub SystemAction(VkAction As SystemKeyShortcuts)
	Const VK_LWIN = &H5B
	Const KEYEVENTF_KEYUP = &H2
	Call keybd_event(VK_LWIN, 0, 0, 0)
	Call keybd_event(VkAction, 0, 0, 0)
	Call keybd_event(VK_LWIN, 0, KEYEVENTF_KEYUP, 0)
End Sub
```

—Randy Birch, East York, Ontario, Canada

VB5, VB6
Level: Beginning

KEEP TRACK OF INDEX NUMBERS

When using control arrays, I find it difficult to keep track of the index number of each control. Even if I use constants, I often have to look up the constant name for each field. Now, instead of using constants for each index number, I use this code. First, I declare an enumerated type for the index numbers:

```
Enum FieldConstants
	LastName = 0
	FirstName = 1
	Age = 2
	Address = 3
End Enum
```

I then create a property wrapper for the control array. The wrapper takes in an enumerated constant that represents the index number and returns the control in the array:

```
Property Get Fields(ByVal FieldNum As FieldConstants) _
	As TextBox
	Set Fields = txtFields(FieldNum)
End Property
```

The advantage to this wrapper is that when you type the property name, Fields, VB prompts you with the constant names listed in the enumerated type. This way, you can refer to controls in the array by name, and you never have to look up constant names again. Also, it makes the code more legible:

```
Private Sub Form_Load()
	Fields(LastName).Text = "Mojica"
End Sub
```

—Jose Mojica, Davie, Florida

VB5, VB6
Level: Beginning

INCLUDE CODE FOR DEBUGGING

VB supports conditional compilation, just like Visual C++. However, Visual C++ has a predefined constant named _DEBUG that makes it easy to include code only while debugging, as in this code:

```
#ifdef _DEBUG
	MessageBox(NULL,"Begin Procedure", _
		"Debug Message",MB_OK);
#endif
```

In VB, you can do the same thing, but you need to declare the variable in the Conditional Compilation Arguments fields in the Make tab of the Project properties dialog, then remember to remove it before shipping the executable. Using the Debug.Assert command is an easier way to have statements executed only while debugging and not when running a compiled program.

For example, this line displays a message box only when running the program in the design environment and not in a compiled program:

```
Debug.Assert MsgBox("Begin Form_Load")
```

This happens because Debug.Assert works only in the design environment. To evaluate the assertion, VB executes the statement. However, when you compile the program, the compiler removes this line from the executable.

—Jose Mojica, Davie, Florida

VB4 32, VB5, VB6
Level: Intermediate

NIX THE X

Sometimes, you want to show a form that you don't want users to be able to cancel by clicking on the X—it might not make sense for your app. The best VB solution is to cancel the unload in the form's QueryUnload event. However, this allows users to do something wrong, for which you then have to handle and scold them. If you do nothing, it looks as if the form has a bug and won't cancel. Add this routine to a standard BAS module:

```
Private Declare Function GetSystemMenu Lib "user32" _
	(ByVal hWnd As Long, ByVal bRevert As Long) As Long
Private Declare Function RemoveMenu Lib "user32" _
	(ByVal hMenu As Long, ByVal nPosition As Long, _
	ByVal wFlags As Long) As Long

Private Const MF_BYPOSITION = &H400&

Public Sub RemoveCancelMenuItem(frm As Form)
	Dim hSysMenu As Long
	'get the system menu for this form
	hSysMenu = GetSystemMenu(frm.hWnd, 0)
	'remove the close item
	Call RemoveMenu(hSysMenu, 6, MF_BYPOSITION)
	'remove the separator that was over the close item
	Call RemoveMenu(hSysMenu, 5, MF_BYPOSITION)
End Sub
```

Then call the routine from any form as it loads:

```
	Private Sub Form_Load()
		RemoveCancelMenuItem Me
	End Sub
```

After this call, the Close menu item in the system menu and the option to cancel [X] will be disabled. Note that if you're doing other things with the system menu, you might have to adjust the position number in the RemoveMenu call.

—Josh Frank, Parsippany, New Jersey

VB3, VB4 16/32, VB5, VB6
Level: Beginning

Compute Credit Card Check Digits

The last digit in your credit card number is a check digit derived from the other digits using the Luhn Formula as described in ISO/IEC 7812-1:1993. Its primary purpose is to ensure accurate entries of the credit card number during transactions. You can apply the same technique to other applications such as employee numbers or patient numbers. Using check digits for these numbers also ensures more accurate data entries:

```
Public Function CheckDigit(strNum As String) As Integer
     Dim i As Integer
     Dim iEven As Integer
     Dim iOdd As Integer
     Dim iTotal As Integer
     Dim strOneChar As String
     Dim iTemp As Integer
     ' Add digits in even ordinal positions
     ' starting from rightmost
     For i = Len(strNum) - 1 To 2 Step -2
          strOneChar = Mid$(strNum, i, 1)
          If IsNumeric(strOneChar) Then
               iEven = iEven + CInt(strOneChar)
          End If
     Next i
     ' Process digits in odd ordinal positions
     ' starting from rightmost
     For i = Len(strNum) To 1 Step -2
     strOneChar = Mid$(strNum, i, 1)
          If IsNumeric(strOneChar) Then
               ' Double it
               iTemp = CInt(strOneChar) * 2
               If iTemp > 9 Then
                    ' Break the digits (e.g., 19 becomes 1+9)
                    iOdd = iOdd + (iTemp \ 10) + (iTemp - 10)
               Else
                    iOdd = iOdd + iTemp
               End If
          End If
     Next i
     ' Add even and odd
     iTotal = iEven + iOdd
     ' Return the 10's complement
     CheckDigit = 10 - (iTotal Mod 10)
End Function
```

To test, pass your credit card number, excluding the last digit, as a string parameter. The result should be the last digit of your credit card number.

—Arnel J. Domingo, Hong Kong

VB4 32, VB5, VB6
Level: Advanced

Use This Higher-Resolution Stopwatch

Use this code to create a class called HiResTimer:

```
'The number is codified as HighPart*2^32+LowPart
Private Type LARGE_INTEGER
     LowPart As Long
     HighPart As Long
End Type
Private Declare Function QueryPerformanceCounter Lib _
     "kernel32" (lpPerformanceCount As LARGE_INTEGER) _
     As Long
Private Declare Function QueryPerformanceFrequency Lib _
     "kernel32" (lpFrequency As LARGE_INTEGER) As Long
Private m_TicksPerSecond As Double
Private m_LI0 As LARGE_INTEGER
Private m_LI1 As LARGE_INTEGER
Friend Sub Class_Initialize()
     Dim LI As LARGE_INTEGER
     If QueryPerformanceFrequency(LI) <> 0 Then
          m_TicksPerSecond = LI2Double(LI)
     Else
          m_TicksPerSecond = -1
     End If
End Sub
Friend Property Get Resolution() As Double
     Resolution = 1# / m_TicksPerSecond
End Property
Friend Sub EnterBlock()
     QueryPerformanceCounter m_LI0
End Sub
Friend Sub ExitBlock()
     QueryPerformanceCounter m_LI1
End Sub
Friend Property Get ElapsedTime() As Double
     Dim EnterTime As Double, ExitTime As Double
     EnterTime = LI2Double(m_LI0) / m_TicksPerSecond
     ExitTime = LI2Double(m_LI1) / m_TicksPerSecond
     ElapsedTime = ExitTime - EnterTime
End Property
Friend Function LI2Double(LI As LARGE_INTEGER) As Double
     Dim Low As Double
     Const TWO_32 = 4# * 1024# * 1024# * 1024#
     Low = LI.LowPart
     If Low < 0 Then Low = Low + TWO_32
          'Now Low is in the range 0...2^32-1
          LI2Double = LI.HighPart * TWO_32 + Low
End Function
```

Here's an example of the HiResTimer in use:

```
Dim hrt As HiResTimer, d As Double
Set hrt = New HiResTimer
Debug.Assert hrt.Resolution > 0
MsgBox "Resolution [usecs]:" & hrt.Resolution * 1000000#
hrt.EnterBlock
hrt.ExitBlock
MsgBox "Call overhead [usecs]:" & hrt.ElapsedTime * _
     1000000#
hrt.EnterBlock
d = 355# / 113#
hrt.ExitBlock
MsgBox "Elapsed Time [usecs]:" & hrt.ElapsedTime * _
     1000000#
```

Believe it or not, you can time even native-compiled code division. For more information, look at the MSDN Library description of the kernel APIs used here. On x86 architectures, resolution is better than 1 microsecond. Be careful, however, of trusting single instance timings, as you'll find the "resolution" of this performance counter varies over time. In fact, the overhead of simply calling QueryPerformanceCounter in VB is quite a measurable time period itself.

Although you can time single operations, you're still better off averaging the time required for hundreds or thousands of similar operations.

—Alessandro Coppo, Rapallo, Italy

VB4 32, VB5, VB6
Level: Intermediate

DRAW FRAMES ON FORM WITHOUT CONTROL

The DrawEdge API provides a convenient way to draw a number of interesting effects. You can change the EDGE_ constants to give different border effects; the BF_ constants determine which borders are drawn (for example, BF_BOTTOM):

```
Private Declare Function DrawEdge Lib "user32" (ByVal hDC _
     As Long, qrc As RECT, ByVal edge As Long, ByVal _
     grfFlags As Long) As Long
Private Declare Function GetClientRect Lib "user32" _
     (ByVal hWnd As Long, lpRect As RECT) As Long
Private Type RECT
     Left As Long
     Top As Long
     Right As Long
     Bottom As Long
End Type
Const BDR_INNER = &HC
Const BDR_OUTER = &H3
Const BDR_RAISED = &H5
Const BDR_RAISEDINNER = &H4
Const BDR_RAISEDOUTER = &H1
Const BDR_SUNKEN = &HA
Const BDR_SUNKENINNER = &H8
Const BDR_SUNKENOUTER = &H2
Const BF_RIGHT = &H4
Const BF_LEFT = &H1
Const BF_TOP = &H2
Const BF_BOTTOM = &H8
Const EDGE_BUMP = (BDR_RAISEDOUTER Or BDR_SUNKENINNER)
Const EDGE_ETCHED = (BDR_SUNKENOUTER Or BDR_RAISEDINNER)
Const EDGE_RAISED = (BDR_RAISEDOUTER Or BDR_RAISEDINNER)
Const EDGE_SUNKEN = (BDR_SUNKENOUTER Or BDR_SUNKENINNER)
Const BF_RECT = (BF_LEFT Or BF_RIGHT Or BF_TOP Or BF_BOTTOM)
```

In the Form_Paint event, put this code where you wish to draw the rectangle:

```
Private Sub Form_Paint()
     Static Tmp As RECT
     Static TmpL As Long
     TmpL = GetClientRect(hWnd, Tmp)
     TmpL = DrawEdge(hDC, Tmp, EDGE_SUNKEN, BF_RECT)
End Sub
```

If the rectangle doesn't draw, do a Debug.Print on the TmpL variable. It should read a nonzero value upon success.

—Jeff Shimano, Mississauga, Ontario, Canada

VB5, VB6
Level: Intermediate

QUICK TIMER CONTROL REPLACEMENT

Timer controls can be practical when you need to add a small delay in program execution. However, if you need the delay in a class module (instead of on a form), the actual control can be hard to get at. Instead, use these functions in a single code module:

```
Option Explicit
     Private Declare Function SetTimer Lib "user32" (ByVal _
     hWnd As Long, ByVal nIDEvent As Long, ByVal uElapse _
     As Long, ByVal lpTimerFunc As Long) As Long
Private Declare Function KillTimer Lib "user32" (ByVal _
     hWnd As Long, ByVal nIDEvent As Long) As Long
Private m_cb As Object
Public Function timerSet(lTime As Long, cb As Object) _
     As Long
     Set m_cb = cb
     timerSet = SetTimer(0, 0, lTime, AddressOf _
          timerProcOnce)
End Function
Private Sub timerProcOnce(ByVal lHwnd As Long, ByVal _
     lMsg As Long, ByVal lTimerID As Long, ByVal lTime _
     As Long)
     On Error Resume Next
     Call KillTimer(0, lTimerID)
     m_cb.cbTimer
End Sub
```

The class module then calls the function like this:

```
...
timerSet 10, Me
...
```

After 10 milliseconds, the code triggers the cbTimer method in the class module:

```
Public Sub cbTimer()
     ' Do some stuff
End Sub
```

You can also use the function on forms instead of the intrinsic Timer control.

—Bo Larsson, Copenhagen, Denmark

VB4 32, VB5, VB6
Level: Intermediate

QUIRKS OF THE DIR$ FUNCTION

If you use the intrinsic VB Dir$ function to check for the existence of a certain file, then subsequently try to remove the directory where the file is found using the VB RMDir statement, you get the error 75, "Path/File access error." This error occurs even if you kill the file prior to removing the directory.

You can see the problem if you manually create a directory and file with the names C:\dummy\bla-bla.txt. Then try to go step-by-step through this sample code to see what's going on:

```
Private Sub Command1_Click()
     If Dir$("C:\dummy\bla-bla.txt") = "" Then
          'do nothing, file is not found
     Else
          'kill the file and remove the directory
          Kill "C:\dummy\bla-bla.txt"
          RmDir "C:\dummy"
     End If
End Sub
```

The statement RmDir "C:\dummy" causes error 75 because this directory is locked and cannot be removed.

To work around this problem, check for the existence of a file by trying to open it for sequential/random/binary (anything should work) access and close it immediately afterwards. If this file exists, your routine will proceed with its code, where you can kill the file and remove the directory. A trappable error 53, “File not found,” indicates the file does not exist. After you trap the error, you can redirect the execution of your code as required. This code is a good example to start with:

```
Private Sub Command1_Click()
     Dim FHandle As Long
     Dim FileNAme As String
     FileNAme = "C:\dummy\bla-bla.txt"
     FH = FreeFile
     On Error Goto ErrHadler
     Open FileNAme For Input As FHandle
     Close FHandle
     Kill FileNAme
     RmDir "C:\dummy"
CleanUp:
Exit Sub
ErrHadler:
     Select case Err
     Case 53 'File not found
          Resume CleanUp
     Case Else
          'display error info
     End Select
End Sub
```

You can’t use Sequential Access for Output or Append. If the file does not exist, it is created automatically when the Open statement executes, and all your code loses sense.

Also note that the RmDir statement causes error 75, “Path/File access error,” if you try to remove a directory that’s not empty. Kill all the files one by one prior to removing a directory, or opt to use an API to do the job.

Of course, you would never want to go to this extreme unless you find that there’s no alternative. This is an incredibly bizarre behavior, and most apps would never be affected by it.

—Brian Hunter, Brooklyn, New York

VB5
Level: Intermediate

SETTING THE DESCRIPTION OF AN ADD-IN

When you use the Add-Ins project template to build your own add-ins, the description of the add-in appearing in the Add-Ins Manager window is always “My Addin.” It isn’t immediately clear how you can change this string. At first, it seems the string corresponds to the project’s Description property, but it actually corresponds to the Description property of the Connect class.

To change this property, press F2 to display the Object Browser, then right-click on the class name and select the Properties menu item. Enter a description in the dialog box and also a HelpContextID for the Class Module. The class description immediately appears in the Object Browser when you click on the class name in the left-most pane.

—Francesco Balena, Bari, Italy

VB3, VB4 16/32, VB5, VB6
Level: Beginning

KEEP MENU ITEMS IN SYNC WITH ENABLED PROPERTY

How many times have you had to control the same processes with both command buttons and corresponding menus? I expect you’d say “many,” which means you’ve had to set the Enabled property of both the buttons and the menus, depending on the availability of certain features, and you’ve had to keep the status of the buttons and menus in sync in many places in your code. This job can be tedious if conditions dictating the Enabled status continuously change in your application. I’ll show you how to cut your code in half.

You’ve probably noticed that if a main menu has submenus, the user can use the main menu only to open a list of menu items. However, the main menu’s Click event procedure is available to you, the programmer. This procedure fires every time the user opens the main menu, always before the user has a chance to select a menu item. Use this event to check your Command buttons’ Enabled property that you set in other procedures of your application, then set the Enabled property of the corresponding menu items to the same value. This way, by the time the user sees the menu item list, all menu items’ Enabled property is set to the appropriate value.

For example, if you have a menu structure similar to this, you can code your Edit menu Click event procedure:

```
Edit
   Undo
   Cut
   Copy
   Paste
   Delete
```

Use this code in your Edit menu Click event procedure:

```
Private Sub mnuEdit_Click
     mnuEditUndo.Enabled = cmdUndo.Enabled
     mnuEditCut.Enabled = cmdCut.Enabled
     mnuEditCopy.Enabled = cmdCopy.Enabled
     mnuEditPaste.Enabled = cmdPaste.Enabled
     mnuEditDelete.Enabled = cmdDelete.Enabled
End Sub
```

No matter how many times you change the Enabled property of buttons, menu items will always be kept in sync.

—Serge Rodkopf, Brooklyn, New York

VB5, VB6
Level: Intermediate

ENUMERATE YOUR ERRORS

To keep a list of available error codes in your components, declare a private enumeration:

```
Private Enum Errors
     InvalidUserID = vbObjectError + 513
     InvalidPassword
     SearchNotFound
End Enum
```

Setting the first value sets the seed number for all subsequent items in the list, each one incrementing by one. (Microsoft recommends starting at “512 plus 1” above vbObjectError.)

Now you won’t have to remember error numbers throughout your code. Simply raise your errors like this:

```
Err.Raise Errors.InvalidUserID, "Login", "Invalid UserID"
```

When you type the enumeration name Errors, VB pops up a list of available choices. Be careful, though, not to add new items in the middle of the list, because the value of all entries below the new item increases by one. This can cause the parent of the object to handle errors incorrectly because the error numbers will be different. You can avoid this by specifying exactly what value you want for each Enum, instead of relying on the default increment.

—Gregory Alekel, Portland, Oregon

VB4 32, VB5, VB6
Level: Intermediate

ALLOW MULTIPLE WINSOCK CONNECTIONS TO ONE SERVER

The Winsock control allows you to make only one connection between two computers. However, you can create multiple connections (many computers to one) by creating multiple instances of the Winsock control at run time.

Add a Winsock control to your form and set its index to 0, then add this code into the server machine to allow multiple connections to it:

```
Option Explicit
Public NumSockets As Integer
'//Public Variable to track number of Connections
Private Sub Form_Load()
     Caption = Winsock1(0).LocalHostName & _
          Winsock1(0).LocalIP
     Winsock1(0).LocalPort = 1066
     Print "Listening to " + Str(Winsock1(0).LocalPort)
     Winsock1(0).Listen
End Sub
Private Sub Winsock1_Close(Index As Integer)
     Print "Connection Closed :" & _
          Winsock1(Index).RemoteHostIP
     Winsock1(Index).Close
End Sub
Private Sub Winsock1_ConnectionRequest(Index As Integer, _
     ByVal requestID As Long)
     Print "Connection Request from : " & _
          Winsock1(Index).RemoteHostIP
     NumSockets = NumSockets + 1
     '//Increase Number of Sockets by one.
     Load Winsock1(NumSockets)
     '//Load a New Winsock Object Nusockets as Index Value
     Winsock1(NumSockets).Accept requestID
     '//Accept the New Connection
End Sub
Private Sub Winsock1_DataArrival(Index As Integer, ByVal _
     bytesTotal As Long)
     Dim vtData As String
     Winsock1(Index).GetData vtData, vbString
     Print vtData
End Sub
```

You should now be able to continue to accept connections from multiple sources.

—Stuart Snaith, Blyth, Northumberland, England

VB3, VB4 16/32, VB5, VB6
Level: Intermediate

VIEW SELECTED TEXT FROM THE BEGINNING

Most professional and commercial applications exhibit a specific behavior when using dialog boxes with text fields: When you tab to an input field, or hot-key (using some ALT-key combination), you fully select the existing text in the field. Any typing then replaces the entire field. However, simply clicking on the text field with the mouse just sets the focus without making any text selection.

The VB Knowledge Base documents how to set this behavior by capitalizing on the GetKeyState API call. However, the technique results in some inconvenience where the text itself is too large for the field width. You wind up looking at the tail end of the highlighted text, not the front end, so it can be difficult to tell what the current text contains.

By combining the GetKeyState API with use of SendKeys and the TextWidth method, you can define a complete subroutine solution where a tab or hot-key to a large text field selects the field, but leaves you looking at the text from the beginning, not the end.

First, declare the GetKeyState API function, and add the SelectWholeText routine, in your form:

```
Option Explicit
     ' Recall that in a form, you need "Private" on an API.
     #If Win16 Then
          Private Declare Function GetKeyState Lib "User" _
               (ByVal iVirtKey As Integer) As Integer
     #Else
          Private Declare Function GetKeyState Lib "User32" _
               (ByVal lVirtKey As Long) As Integer
     #End If
     ' vbTab       ' same as Chr$(&H9) - character constant
     ' vbKeyTab    ' same as decimal 9 - key code constant
     ' vbKeyMenu   ' same as decimal 18 (Alt key) - key code
                   ' constant
Private Sub SelectWholeText(Ctl As Control)
     ' If you got to the "Ctl" field via either TAB or an
     ' Alt-Key, highlight the whole field. Otherwise select
     ' no text, since it must have received focus using a
     ' mouse-click.
     ' Note difference between vbTab (character) and vbKeyTab
     ' (numeric constant). If vbTab were used, we'd have to
     ' Asc() it to get a number as an argument.
     ' Use VB4/5's "With" to improve maintainability in case
     ' the parameter name changes.
     With Ctl
          If (GetKeyState(vbKeyTab) < 0) Or _
               (GetKeyState(vbKeyMenu) < 0) Then
               ' We tabbed or used a hotkey - select all text. In
               ' the case of a long field, use Sendkeys so we
               ' see the beginning of the selected text.
               ' TextWidth Method tells how much width a string
               ' takes up to display (default target object is
               ' the Form).
               If TextWidth(.Text) > .Width Then
                    SendKeys "{End}", True
                    SendKeys "+{Home}", True
               Else
                    .SelStart = 0
                    .SelLength = Len(.Text)
               End If
```

```
        Else
            .SelLength = 0
        End If
    End With
End Sub
```

Next, call the subroutine in the GotFocus event of any text field:

```
Private Sub txtPubID_GotFocus()
    SelectWholeText txtPubID
End Sub
Private Sub txtTitle_GotFocus()
    SelectWholeText txtTitle
End Sub
```

—Mark Cohen, Winnipeg, Manitoba, Canada

VB4 32, VB5, VB6

Level: Intermediate

DETERMINE WHICH SCROLLBARS ARE VISIBLE

Sometimes, it's useful to know whether a control is displaying vertical scrollbars, horizontal scrollbars, both horizontal and vertical, or no scrollbars at all. For instance, when determining the amount of room inside a control, you might have to take into account the space taken up by the scrollbars. This function, written in VB5, returns an enumerated constant representing the scrollbar state of the given control:

```
Private Declare Function GetWindowLong Lib "user32" Alias _
    "GetWindowLongA" (ByVal hWnd As Long, ByVal nIndex _
    As Long) As Long
'GetWindowLong Constants
Private Const GWL_STYLE = (-16)
Private Const WS_HSCROLL = &H100000
Private Const WS_VSCROLL = &H200000
'Used by VisibleScrollBars function
Enum Enum_VisibleScrollBars
    vs_none = 0
    vs_vertical = 1
    vs_horizontal = 2
    vs_both = 4
End Enum
Public Function VisibleScrollBars(ControlName As Control) _
    As Enum_VisibleScrollBars
    '
    ' Returns an enumerated type constant depicting the
    ' type(s) of scrollbars visible on the passed control.
    '
    Dim MyStyle As Long
    MyStyle = GetWindowLong(ControlName.hWnd, GWL_STYLE)
    'Use a bitwise comparison
    If (MyStyle And (WS_VSCROLL Or WS_HSCROLL)) = _
        (WS_VSCROLL Or WS_HSCROLL) Then
        'Both are visible
        Let VisibleScrollBars = vs_both
    ElseIf (MyStyle And WS_VSCROLL) = WS_VSCROLL Then
        'Only Vertical is visible
        Let VisibleScrollBars = vs_vertical
    ElseIf (MyStyle And WS_HSCROLL) = WS_HSCROLL Then
        'Only Horizontal is visible
        Let VisibleScrollBars = vs_horizontal
    Else
        'No scrollbars are visible
        Let VisibleScrollBars = vs_none
    End If
End Function
```

Hard-coding a scrollbar with a predetermined width or height is not a good idea because these might vary depending on the user's display settings (accessibility options, desktop themes, and so on). Call GetSystemMetrics to always ensure the proper value for this metric.

—Michael B. Kurtz, McKees Rocks, Pennsylvania

VB4 32, VB5, VB6

Level: Intermediate

EASY CONFIRMATION

Sometimes you want to give your users the chance to confirm that they wish to proceed with the closure of a form. Instead of using a MsgBox function and a Select Case statement, put this code in the Form_Unload event:

```
Private Sub Form_Unload(Cancel as Integer)
    Cancel = (MsgBox("Quit now?", vbOKCancel Or _
        vbQuestion, "Confirmation Demo") = vbCancel)
End Sub
```

Behind the Exit button and the Exit menu option, put a simple Unload Me. Whenever users choose to exit, they will be asked to confirm their action.

—Rae MacLeman, Peterborough, United Kingdom

VB4 32, VB5, VB6

Level: Beginning

USE MOUSEMOVE FOR EASY STATUSBAR UPDATES

You can easily make your program show descriptive text on a StatusBar control in response to mouse movement. Assign the text to the appropriate panel in the MouseMove events of the appropriate controls, then use the Form_MouseMove event to clear text from the panel:

```
Private Sub txtAddress_MouseMove(Button As Integer, Shift _
    As Integer, X As Single, Y As Single)
    StatusBar1.Panels(1).Text = "Enter Address here."
End Sub
Private Sub txtName_MouseMove(Button As Integer, Shift _
    As Integer, X As Single, Y As Single)
    StatusBar1.Panels(1).Text = "Enter Name here."
End Sub
Private Sub Form_MouseMove(Button As Integer, Shift _
    As Integer, X As Single, Y As Single)
    StatusBar1.Panels(1).Text = ""
End Sub
```

—Ron Schwarz, Mt. Pleasant, Michigan

VB5, VB6
Level: Advanced

Provide a Horizontal Scroll Event for a Listbox

Subclassing a listbox allows you to monitor horizontal scrolling. To subclass a listbox, you store the original WndProc in the UserData area of the listbox, allowing a single replacement of WndProc to work for all ListBox controls. WndProc notifies your form of a horizontal scroll message by sending a WM_MOUSEMOVE message with negative coordinates. The MouseMove event receives negative X and Y values, plus a Button value when horizontally scrolled, which is impossible under normal operation. Be sure to restore the original WndProc in the Form_Unload event:

```
'--- Form code
Private Sub Form_Load()
     SetWindowLong List1.hwnd, GWL_USERDATA, _
          SetWindowLong(List1.hwnd,GWL_WNDPROC, _
          AddressOf WndProc)
     SetWindowLong List2.hwnd, GWL_USERDATA, _
          SetWindowLong(List2.hwnd, GWL_WNDPROC, _
          AddressOf WndProc)
End Sub
Private Sub Form_Unload(Cancel As Integer)
     SetWindowLong List1.hwnd, GWL_WNDPROC, _
          GetWindowLong(List1.hwnd, GWL_USERDATA)
     SetWindowLong List2.hwnd, GWL_WNDPROC, _
          GetWindowLong(List2.hwnd, GWL_USERDATA)
End Sub
Private Sub List1_MouseMove(Button As Integer, Shift _
     As Integer, X As Single, Y As Single)
     If Button > 0 And X < 0 And Y < 0 Then Debug.Print _
          "List1 Horizontal Scroll"
End Sub
Private Sub List2_MouseMove(Button As Integer, Shift _
     As Integer, X As Single, Y As Single)
     If Button > 0 And X < 0 And Y < 0 Then Debug.Print _
          "List2 Horizontal Scroll"
End Sub
'--- Module code
Public Declare Function CallWindowProc Lib "user32" Alias _
     "CallWindowProcA" (ByVal lpPrevWndFunc As Long, ByVal _
     hwnd As Long, ByVal Msg As Long, ByVal wParam As Long, _
     ByVal lParam As Long) As Long
Public Declare Function GetWindowLong Lib "user32" Alias _
     "GetWindowLongA" (ByVal hwnd As Long, ByVal nIndex _
     As Long) As Long
Public Declare Function SendMessage Lib "user32" Alias _
     "SendMessageA" (ByVal hwnd As Long, ByVal wMsg As _
     Long, ByVal wParam As Long, lParam As Any) As Long
Public Declare Function SetWindowLong Lib "user32" Alias _
     "SetWindowLongA" (ByVal hwnd As Long, ByVal nIndex As _
     Long, ByVal dwNewLong As Long) As Long
Public Const GWL_WNDPROC = (-4)
Public Const GWL_USERDATA = (-21)
Public Const WM_HSCROLL = &H114
Public Const WM_MOUSEMOVE = &H200
Public Function WndProc(ByVal hwnd As Long, ByVal Msg As _
     Long, ByVal wParam As Long, ByVal lParam As Long) _
     As Long
     If Msg = WM_HSCROLL Then SendMessage hwnd, _
          WM_MOUSEMOVE, 1, ByVal &HFFFF
          WndProc = CallWindowProc(GetWindowLong(hwnd, _
               GWL_USERDATA), hwnd, Msg, wParam, lParam)
End Function
```

For brevity, this example omits the code that would add horizontal scrollbars to the List1 and List2 controls, but it is readily available in the Knowledge Base (Q192184).

—Matt Hart, Tulsa, Oklahoma

VB5, VB6
Level: Intermediate

Show Data With Outlook Grid

You can easily create a calendar grid similar to the one in Outlook to show data. First, start a VB project, then place an MSFlexGrid on the form, with two textboxes to set the start and end dates and a command button to populate the grid. Paste this code in the Form1 code pane:

```
Private Sub Command1_Click()
     Dim ldBegDate As Date
     Dim ldEndDate As Date
     Dim lsMonth As String
     Dim li, lsStr, lsStartDate, lsEndDate, liStartRow, _
          liEndRow, liCnt, lsDate, liX
     ldBegDate = Format(Text1, "mm/dd/yyyy")
     ldEndDate = Format(Text2, "mm/dd/yyyy")
     MSFlexGrid1.Clear: MSFlexGrid1.Cols = 5: _
          MSFlexGrid1.Rows = 1: MSFlexGrid1.RowHeightMin _
          = 800
     For li = 0 To DateDiff("M", ldBegDate, ldEndDate)
          lsMonth = Format(DateAdd("m", _
               li, ldBegDate), "mmmyyyy")
          '== check what month is being processed
          lsStartDate = "": lsEndDate = ""
          '== Starting Month
          If InStr(1, lsMonth, Format(ldBegDate, "mmmyyyy")) _
               Then lsStartDate = ldBegDate
          '== Ending Month
          If InStr(1, lsMonth, Format(ldEndDate, "mmmyyyy")) _
               Then lsEndDate = ldEndDate
          '== Neither Starting or Ending Month
          If lsStartDate = "" Then lsStartDate = _
               Format(DateAdd("m", li, ldBegDate), "mm/01/yyyy")
          If lsEndDate = "" Then lsEndDate = _
               Format(DateAdd("m", 1, Format(lsStartDate, _
               "mm/01/yyyy")) - 1, "mm/dd/yyyy")
          liStartRow = MSFlexGrid1.Rows
          liEndRow = liStartRow + DateDiff("d", lsStartDate, _
               lsEndDate)
          MSFlexGrid1.Rows = MSFlexGrid1.Rows + _
               liEndRow - liStartRow + 1
          MSFlexGrid1.Col = 0
          '== Put Dates in grid
          For liCnt = 0 To DateDiff("d", _
               lsStartDate, lsEndDate)
               lsDate = DateAdd("d", liCnt, lsStartDate)
               MSFlexGrid1.Row = liStartRow + liCnt
               MSFlexGrid1.Text = lsDate
               MSFlexGrid1.CellFontSize = 10: _
                    MSFlexGrid1.CellFontBold = True
               MSFlexGrid1.CellAlignment = _
                    flexAlignCenterCenter
               If WeekDay(lsDate, vbMonday) > 5 Then
                    For liX = 1 To MSFlexGrid1.Cols - 1
                         MSFlexGrid1.Col = liX
                         MSFlexGrid1.CellBackColor = _
                              &HC0C0C0: MSFlexGrid1.CellForeColor _
                              = &H80000015
```

```
                Next
                MSFlexGrid1.Col = 0
            End If
        Next
    Next '== li..Start to End Date

End Sub
```

—Danny Patel, Duluth, Georgia

VB3, VB4 16/32, VB5, VB6

Level: Beginning

Avoiding Hard-Coded Path for Data Control Targets

You can easily create simple database projects in design mode by assigning a set of database properties to the Data control. Once you assign a database and record source, you can bind controls to the available fields. However, when you assign a database name, VB inserts an explicit path to the location where it resides on *your* machine, such as, "D:\Program Files\Microsoft Visual Studio\VB98\Tips Fall 98\NWIND.MDB". If you compile the application and send it to other people, it doesn't run unless their database is in the same location on *their* machines. You can assign the properties in code and have them take effect at run time. However, this prevents you from linking the bound controls at design time. You can easily work around this by designing the project on your machine, letting VB insert whatever path information it wants, and reassigning the DatabaseName property at run time:

```
Private Sub Form_Load()
    Data1.DatabaseName = App.Path & "\NWIND.MDB"
End Sub
```

In this code, the database resides in the same directory as the application itself. If you place it in a different directory, use the appropriate path name.

—Ron Schwarz, Mt. Pleasant, Michigan

VB5, VB6

Level: Advanced

Provide Status Messages for Menus

Subclassing a form lets you give a helpful message whenever a user highlights a menu item. Use the Caption property to identify the menu item, then display the help message in a label (lblStatus), which is on the form:

```
' --- Form code
Private Sub Form_Load()
    origWndProc = SetWindowLong(hwnd, GWL_WNDPROC, _
        AddressOf AppWndProc)
End Sub
Private Sub Form_Resize()
    lblStatus.Move 0, ScaleHeight - lblStatus.Height, _
        ScaleWidth
End Sub
Private Sub Form_Unload(Cancel As Integer)
    SetWindowLong hwnd, GWL_WNDPROC, origWndProc
End Sub
'--- Module code
Type MENUITEMINFO
    cbSize As Long
    fMask As Long
    fType As Long
    fState As Long
    wID As Long
    hSubMenu As Long
    hbmpChecked As Long
    hbmpUnchecked As Long
    dwItemData As Long
    dwTypeData As String
    cch As Long
End Type
Public Declare Function CallWindowProc Lib "user32" Alias _
    "CallWindowProcA" (ByVal lpPrevWndFunc As Long, ByVal _
    hwnd As Long, ByVal Msg As Long, ByVal wParam As Long, _
    ByVal lParam As Long) As Long
Public Declare Sub CopyMemory Lib "kernel32" Alias _
    "RtlMoveMemory" (hpvDest As Any, hpvSource As Any, _
    ByVal cbCopy As Long)
Public Declare Function GetMenuItemInfo Lib "user32" Alias _
    "GetMenuItemInfoA" (ByVal hMenu As Long, ByVal un As _
    Long, ByVal b As Boolean, lpMenuItemInfo As _
    MENUITEMINFO) As Long
Public Declare Function SetWindowLong Lib "user32" Alias _
    "SetWindowLongA" (ByVal hwnd As Long, ByVal nIndex As _
    Long, ByVal dwNewLong As Long) As Long
Public Const GWL_WNDPROC = (-4)
Public Const WM_MENUSELECT = &H11F
Public Const MF_SYSMENU = &H2000&
Public Const MIIM_TYPE = &H10
Public Const MIIM_DATA = &H20
Public origWndProc As Long
Public Function AppWndProc(ByVal hwnd As Long, ByVal Msg _
    As Long, ByVal wParam As Long, ByVal lParam As Long) _
    As Long
    Dim iHi As Integer, iLo As Integer
    Select Case Msg
        Case WM_MENUSELECT
            Form1.lblStatus.Caption = ""
            CopyMemory iLo, wParam, 2
            CopyMemory iHi, ByVal VarPtr(wParam) + 2, 2
            If (iHi And MF_SYSMENU) = 0 Then
                Dim m As MENUITEMINFO, aCap As String
                m.dwTypeData = Space$(64)
                m.cbSize = Len(m)
                m.cch = 64
                m.fMask = MIIM_DATA Or MIIM_TYPE
                If GetMenuItemInfo(lParam, CLng(iLo), _
                    False, m) Then
                    aCap = m.dwTypeData & Chr$(0)
                    aCap = Left$(aCap, _
                        InStr(aCap, Chr$(0)) - 1)
                    Select Case aCap
                        Case "&Open": _
                            Form1.lblStatus.Caption = _
                            "Open a file"
                        Case "&Save": _
                            Form1.lblStatus.Caption = _
                            "Save a file"
                    End Select
                End If
            End If
    End Select
    AppWndProc = CallWindowProc(origWndProc, hwnd, Msg, _
        wParam, lParam)
End Function
```

—Matt Hart, Tulsa, Oklahoma

VB5, VB6
Level: Intermediate

EXTEND INTRINSICS IN USERCONTROL WRAPPERS

Sometimes you might want to create autoselecting combo boxes, like those in Intuit Quicken or Microsoft Access, where items in the list are selected as you type. Typically, techniques to do this require code in the KeyPress and KeyDown events, which select the items from the list. I'll show you how to create a more intuitive user-interface design than using a standard combo box.

Instead of repeating this code (or calls to a generic function) in the KeyPress event of every combo box in every project, create your own combo box control with this functionality built in. Create a new UserControl, add a combo box, and use the ActiveX Control Interface Wizard to create Property Let and Get procedures to set and retrieve the combo box's standard properties. Add this code to make sure the combo box always sizes to the same size as the control:

```
Private Sub UserControl_Resize()
     cbo.Move 0, 0, ScaleWidth
     '  The Combo Box height determines the height
     '  of the control
     UserControl.Height = Screen.TwipsPerPixelY * cbo.Height
End Sub
```

Add an AutoSelect property as a Boolean value, so the developer can decide when to use the autoselecting functionality. The Property Let procedure sets the value of the module-level Boolean variable mfAutoSelect. Check the value of mfAutoSelect in the KeyPress event:

```
Private Sub cbo_KeyPress (KeyAscii as Integer)
     If mfAutoSelect Then
          '  Call generic function to select the first matching
          '  value in the list.
          Call ComboBox_AutoSelect(cbo, KeyAscii)
     End if
End Sub
```

You should also make a number of simple but useful improvements to the standard combo box control. First, add a variant ItemData property, create a LimitToList property similar to combo boxes in Access, and select the text in the GotFocus event. If you set the Appearance property to Flat, the combo box still appears in 3-D, so work around this bug. By wrapping this code in an ActiveX control, you reduce the code in your applications.

—Craig Randall, Wilmington, Massachusetts

VB4 32, VB5, VB6
Level: Beginning

ACCESS PROPERTIES ONLY THROUGH THE OBJECT BROWSER

You can access certain properties of forms, modules, classes, and events *within* them only through the Object Browser. To see and update the properties, right-click on the element and select Properties. The actual properties, as well as the fine points of the update window, vary according to your version of VB, as well as the type of object you're editing.

—Ron Schwarz, Mt. Pleasant, Michigan

VB4 32, VB5, VB6
Level: Intermediate

CREATE AUTOMATIC HOURGLASS CURSORS

If you create an ActiveX control that uses a custom MousePointer property, place the control on a form, and at run time change the Screen.MousePointer property to an hourglass. You'll see that the mouse pointer reverts to the ActiveX control's custom cursor while over the control.

I discovered this when I created an ActiveX control that appeared as a hyperlink and could be placed on a VB form. I changed the MousePointer for the control to a pointing hand cursor, similar to the cursor in Internet Explorer. When I used the control in a project, the control's Click event changed the screen's MousePointer to an hourglass, but this had no effect while the mouse was over the ActiveX control.

To prevent this inconsistency, disable the form when you show the hourglass, but enable it when the hourglass is turned off. While the form is disabled, the MousePointer property of the ActiveX control no longer takes precedence over the Screen.MousePointer property. Use this generic clsHourglass class module to change the mouse pointer and disable the current window:

```
<clsHourglass>
Option Explicit
     Private mintOldPointer As Integer
     Private mlngHwnd As Long
     Private Declare Function EnableWindow Lib "user32" _
          (ByVal hWnd As Long, ByVal fEnable As Long) As Long
     Private Sub Class_Initialize()
          On Error Resume Next
          '  Save current mouse pointer
          mintOldPointer = Screen.MousePointer
          '  Change to hourglass
          Screen.MousePointer = vbHourglass
          '  Save the window handle and disable the
          '  current window.
          mlngHwnd = Screen.ActiveForm.hWnd
          EnableWindow mlngHwnd, 0
          DoEvents
     End Sub
     Private Sub Class_Terminate()
          On Error Resume Next
          '  Set pointer to old pointer
          Screen.MousePointer = mintOldPointer
          '  Enable the previously visible window
          EnableWindow mlngHwnd, 1
     End Sub
```

Now, instead of explicitly changing the mouse pointer to an hourglass and back and manually disabling and re-enabling forms, use this code at the beginning of your event procedures:

```
     Dim objHourglass as clsHourglass
     Set objHourglass = New clsHourglass
```

With this approach, the Initialize event changes the mouse pointer to an hourglass and disables the current form when the procedure creates the objHourglass. The hourglass can't accidentally be left on because when the objHourglass variable loses scope, the Terminate event fires, which returns the mouse pointer to its original state and re-enables the form if it's still loaded.

By using the EnableWindow API call, you can't reload a form accidentally if it was unloaded during the time the objHourglass object had life. That could happen if you used the syntax "frmCurrent.Enabled = True". Multiple instances of objHourglass won't cause the mouse pointer to flicker, because the mouse pointer's current state is checked and saved before setting it to an hourglass.

If the user keeps clicking on the form while the hourglass is shown, the extra clicks are discarded because the form isn't enabled. And unlike earlier versions of VB, a form doesn't deactivate when disabled, so the title bar doesn't flicker.

—Craig Randall, Wilmington, Massachusetts

VB3, VB4 16/32, VB5, VB6

Level: Beginning

TRANSLATE COLOR VALUES

With the RGB function, VB provides a neat and valuable tool for converting separate Red, Green, and Blue values into a single Long color value. However, VB won't let you translate back from this color value to its constituent RGB values. But, you can pick the individual colors out of a hexadecimal representation of the Long value produced by RGB. The colors fall in "BBGGRR" order. Put this code in a module:

```
Type RGB_Type
    R As Long
    G As Long
    B As Long
End Type
Function ToRGB(ByVal Color As Long) As RGB_Type
    Dim ColorStr As String
    ColorStr = Right$("000000" & Hex$(Color), 6)
    With ToRGB
    .R = Val("&h" & Right$(ColorStr, 2))
    .G = Val("&h" & Mid$(ColorStr, 3, 2))
    .B = Val("&h" & Left$(ColorStr, 2))
    End With
End Function
```

To use this function, put a picture in a form's Picture property, and insert this code in that form:

```
Private Sub Form_MouseUp(Button As Integer, Shift _
    As Integer, X As Single, Y As Single)
    Dim RGB_Point As RGB_Type
    RGB_Point = ToRGB(Point(X, Y))
    Caption = RGB_Point.R & " " & RGB_Point.G & " " & _
        RGB_Point.B
End Sub
```

Click on different places on the picture. VB3 users must return the values differently, because VB didn't support the return of a user-defined type until VB4.

—Brian Donovan, Bakersfield, California

VB5, VB6

Level: Intermediate

DELEGATE GENERIC EVENT HANDLING

It can be useful to create generic controls of similar properties. For example, if a project has 10 textboxes on different forms that need to accept numeric input only, instead of repeating the same code in every textbox, you can create a class called clsGeneric and declare the control using WithEvents. You can then trap the events in one place.

Create a collection of the clsGeneric class and keep adding controls to the collection. When you end the app, set the Collection object to Nothing. You can use control arrays, but if you're using them across the forms, you don't have to repeat the code:

```
' --- Save the following code in clsGeneric.cls
Option Explicit
Public WithEvents txtAny As TextBox
Private Sub txtAny_GotFocus()
    If Len(Trim$(txtAny)) > 0 Then
        txtAny.SelStart = 0
        txtAny.SelLength = Len(txtAny)
    End If
End Sub
' -- Save the following code in clsGenerics.cls
Option Explicit
Private mColGenerics As New Collection
Public Function Add(ByVal txtAny As TextBox, Optional _
    ByVal Key As String = "") As clsGeneric
    Dim clsAny As New clsGeneric
    Set clsAny.txtAny = txtAny
    If Key = "" Then
        mColGenerics.Add clsAny
    Else
        mColGenerics.Add clsAny, Key
    End If
    Set Add = clsAny   ' Return a reference to the new textbox
End Function
Public Function Count() As Long
    Count = mColGenerics.Count
End Function
Public Sub Delete(ByVal Index As Variant)
    mColGenerics.Remove Index
End Sub
Public Function Item(ByVal Index As Variant) As clsGeneric
    Set Item = mColGenerics.Item(Index)
End Function
Public Function NewEnum() As IUnknown
    Set NewEnum = mColGenerics.[_NewEnum]
End Function
' -- In any form or global module where you want to have
' -- this generic textboxes
Private clsTexts As New clsGenerics
' In form load add the controls to the collection like this.
    clsTexts.Add Text1
    clsTexts.Add Text2
    clsTexts.Add Text3
' You can even declare clsTexts globally and keep adding
' controls in whatever forms needed.
```

—Badari Syam Mysore, Scotch Plains, New Jersey

VB3, VB4 16/32, VB5, VB6

Level: Beginning

EASILY DETERMINE WHETHER A RECORDSET IS EMPTY

Use this quick and dirty routine to help find empty recordsets:

```
Public Function IsEmptyRecordset(rs As Recordset) As Boolean
    IsEmptyRecordset = ((rs.BOF = True) And (rs.EOF = True))
End Function
```

—Badari Syam Mysore, Scotch Plains, New Jersey

For even more tricks and tips go to
http://www.devx.com

VB3, VB4 16/32, VB5, VB6
Level: Beginning

Remove Unwanted Characters

When working with or fixing Access databases and Excel spreadsheets created by users at my company, I often have to use their strings from the table or spreadsheet to save the resultant fixed file. The problem is that the strings they use often contain illegal file-name characters. This function strips away the illegal characters and replaces them with an underscore character. To use the character, call the function, passing the string you want checked and stripped:

```
Public Function fConvert(ByVal sStr As String) As String
    Dim i As Integer
    Dim sBadChar As String
    ' List all illegal/unwanted characters
    sBadChar = "/<>?\|*:'"
    ' Loop through all the characters of the string
    ' checking whether each is an illegal character
    For i = 1 To Len(sStr)
        If InStr(sBadChar, Mid(sStr, i, 1)) Then
            Mid(sStr, i, 1) = "_"
        End If
    Next I
    fConvert = sStr
End Function
```

—Grant Porteous, St. Cloud, Minnesota

VB4 32, VB5, VB6
Level: Beginning

Use Unadvertised Controls

When you open VB5's Components list, you'll see many controls and libraries not available for your development. Some are controls you downloaded from Web pages; others come from who knows where.

If you've ever tried adding an unknown control to your IDE, you probably saw an icon added to your control's palette. However, since you couldn't use the control, you probably just ignored them all and selected the controls that you're positive came with your copy of VB.

Wait! Open that Component list again and select these items:

Wang Image Admin Control
Wang Image Scan Control
Wang Image Edit Control
Wang Image Thumbnail Control

Under Windows 98, the name "Kodak" is used, rather than "Wang." Add these items to your palette, then add them to a form. Select the control and press F1. Up pops the developer's help on using the controls in your projects.

These may not be the final word on imaging controls, but with all their properties and methods for image manipulation, conversions, displays, and more, they're leaps and bounds beyond picture and image controls, and they're free—with Windows 95/OSR2, Windows 98, and NT4. The one restriction you need to be aware of is that these controls are *not* redistributable, and Windows 95 users must download them (from http://www.eastmansoftware.com) and perform the separate install themselves.

—Robert Smith, San Francisco, California

VB4 32, VB5, VB6
Level: Beginning

Activate Single Control on All Tabs

A single object, employing a single set of event routines, may be used across all pages in the SSTab control. Draw the object on the form that contains the SSTab and drag it on any page on SSTab. The object appears in the same location on all pages and can be operated from any page, because it occupies a higher position in the ZOrder. I've found this useful for items such as a command button to exit the program from any page.

You can also use this technique for textboxes, listboxes, and combo boxes. Information entered on one page appears on all others and can be changed on any page. If you don't want the control to appear on certain pages, then code the click event for SSTab, like this:

```
Private Sub SSTab1_Click(PreviousTab As Integer)
    If SSTab1.Tab = 1 or SSTab1.Tab = 3 Then
        cmdQuit.Visible = False
    Else
        cmdQuit.Visible = True
    End If
End Sub
```

You can use similar coding to change other properties on different pages, such as repositioning by setting up a Select Case block on SSTab1.Tab for more complex instructions.

—Marvin Boehm, Skokie, Illinois

VB3, VB4 16/32, VB5, VB6
Level: Beginning

Perform Look-Ahead Typing

This subroutine lets the user perform look-ahead typing, as in Netscape Navigator, Microsoft Internet Explorer, and other apps. The sub takes an array of strings and a TextBox control as parameters. You can easily change the subroutine to accept a ListBox control instead of an array of strings. You can call this sub from the TextBox's KeyUp event:

```
Public Sub DoLookAhead(strArray() As String, ctlText _
    As TextBox)
    Dim strText As String
    Dim strLength As Integer
    Dim x As Integer
    strText = LCase$(ctlText.Text)
    strLength = Len(strText)
    If strLength > 0 Then
        For x = LBound(strArray) To UBound(strArray)
        If strText = LCase$(Left$(strArray(x), strLength)) Then
            'we found something
            If Len(strArray(x)) > strLength Then
                ctlText.Text = ctlText.Text + _
                    Mid$(strArray(x), strLength + 1)
                ctlText.SelStart = strLength
                ctlText.SelLength = Len(strArray(x)) - _
                    strLength
            End If
            Exit For
        End If
        Next
    End If
End Sub
```

—Robert Gelb, Huntington Beach, California

VB3, VB4 16/32, VB5, VB6
Level: Beginning

DO YOU KNOW ABOUT DATE LITERALS?

Using a Date literal is about 12 times faster—according to NuMega TrueTime—than using the CDate function and converting a string literal. Here's an example:

```
Dim TestDate as Date
    'The following 2 lines produce the same results
    TestDate = #7/1/98#
    TestDate = CDate("7/1/98")
```

Just as you enclose a string literal with quotes ("Hello"), you can enclose Date literals with pound signs (#07/07/1998#). So, these are all valid Date literals: #July 7, 1998#, #7-JUL-98#, and #07/07/1998#.

—James Bragg, received by e-mail

VB5, VB6
Level: Beginning

KEEP TRACK OF GLOBAL VARIABLES

Forgetting the names of global variables is easy to do. You can solve this problem by letting VB remember the variables for you by making the variables into fields of a type definition:

```
Private Type GlobalVars
INIFilePath As String
    SqlServerConnection As rdoConnection
    UserName As String
    Password As String
    DSN As String
End Type
Public Globals As GlobalVars
```

With this code included in a code module, you need to type only "Globals" and VB lists them all.

—Bennett Sy, Toronto, Ontario, Canada

VB5, VB6
Level: Beginning

IS THE ARRAY DIMENSIONED?

Use this function to determine whether a dynamic array has been dimensioned. I use dynamic arrays to store small, foreign key tables locally for fast lookups. Not wanting to load all the arrays initially, I load them only when needed and only once.

I wanted to exploit the fact that the array hasn't yet been dimensioned as the indicator to load, instead of a Boolean variable, but I ran into complications with the standard tests in VB. My function returns True if the dynamic array passed to it has yet to be ReDim'd:

```
Public Function IsArrayEmpty(aArray As Variant) As Boolean
    On Error Resume Next
    IsArrayEmpty = UBound(aArray)
    IsArrayEmpty = Err ' Error 9 (Subscript out of range)
End Function
```

Use this test code:

```
Option Explicit
Private Sub cmdTest_Click()
    Dim aDynamic() As Integer
    MsgBox IsArrayEmpty(aDynamic)
    ReDim aDynamic(8)
    MsgBox IsArrayEmpty(aDynamic)
End Sub
```

—Richard Hundhausen, Boise, Idaho

VB4 16/32, VB5, VB6
Level: Beginning

QUICK CHECK ON COLLECTION KEY

You might need to determine whether a certain key is in a collection. The Collection object doesn't provide a method to retrieve a key after you add an object to the collection. Instead of keeping a separate list of the keys in the collection, use this function:

```
Public Function IsKeyInCollection(col As Collection, sKey _
    As String) As Boolean
    On Error Resume Next
    col (sKey)
    ' this will cause an error if key is not in collection
    IsKeyInCollection = (Err.Number = 0)
End Function
```

You can easily modify this function to make it a property of a Collection class.

—Alan Borowski, Cudahy, Wisconsin

VB5, VB6
Level: Intermediate

SET THE WIDTH OF A LISTVIEW COLUMN

The ListView Windows control has some additional features that haven't been exposed in the OCX provided with VB. One feature can shrink or enlarge columns automatically to ensure that data in each column is visible and that no screen space is wasted. Use this function to make an API call:

```
Public Declare Function SendMessage Lib "user32" Alias _
    "SendMessageA" (ByVal hwnd As Long, ByVal wMsg As _
    Long, ByVal wParam As Long, lParam As Any) As Long
Const LVM_SETCOLUMNWIDTH = &H1000 + 30
Const LVSCW_AUTOSIZE = -1
Const LVSCW_AUTOSIZE_USEHEADER = -2

Sub AdjustColumnWidth(LV As ListView, AccountForHeaders _
    As Boolean)
    Dim col As Integer, lParam As Long
    If AccountForHeaders Then
        lParam = LVSCW_AUTOSIZE_USEHEADER
    Else
        lParam = LVSCW_AUTOSIZE
    End If
    ' Send the message to all the columns
    For col = 0 To LV.ColumnHeaders.Count - 1
        SendMessage LV.hwnd, LVM_SETCOLUMNWIDTH, col, _
            ByVal lParam
    Next
End Sub
```

You can resize all columns, taking the text in column headers into account by passing True as the second argument:

```
AdjustColumnWidth ListView1, True
```

If you pass False as the second argument, the text in column headers is ignored in determining the correct width.

—Marco Losavio, Gioia del Colle, Italy

VB4 32, VB5, VB6

Level: Beginning

DON'T DUPLICATE IMAGELIST CONTROLS

A common control, such as a TreeView or a ListView, can use an ImageList control placed on a form different from the current one. This is useful when many forms in your application contain an ImageList control with the same images.

Instead of using multiple ImageList controls, you can gather all your images into one ImageList control placed (if possible) on a form that's always loaded, such as the main form in your application. Assign that ImageList control at run time to the ImageList property of other controls:

```
Private Sub Form_Load()
     Set TreeView1.ImageList = frmMain.ImageList1
     Set ListView1.ImageList = frmMain.ImageList1
End Sub
```

This approach lets you save some space in the compiled EXE file and use fewer resources at run time.

—Francesco Balena, Bari, Italy

VB6

Level: Beginning

INSTALL THE TOOLBAR WIZARD

VB6 comes with a Toolbar Wizard that lets you easily create a Toolbar control and the code that responds when users click on its buttons. However, how you install this wizard is unclear, because it doesn't appear in the list displayed in the Add-Ins Manager dialog box. You have to install the VB6 Application Wizard, which also contains the Toolbar Wizard. You can then activate the Toolbar Wizard from the Add-Ins menu or automatically when you drop a Toolbar control on a form.

—Francesco Balena, Bari, Italy

VB4 32, VB5, VB6

Level: Beginning

USE MAXFILESIZE WITH OPENFILE COMMON DIALOGS

When working with OpenFile common dialogs, set a large value for the MaxFileSize property, which affects the maximum length of the string returned in the FileName property. This is the string containing the names of all the files selected by the user. For example, you can reserve 10 kilobytes for this string by executing this statement before showing the dialog:

```
CommonDialog1.MaxFileSize = 10240
```

The returned string's format depends on how many files the user selects. If the user selects only one file, the string returns in the FileName property that contains the complete name of this file (path + name). If the user selects multiple files, the returned value contains the directory name, followed by all the names without the path. If you use the cdlOFNExplorer flag, the separator character is the Null character. If you're using VB6, you can use the Split function to quickly parse the returned value:

```
CommonDialog1.Filter = "All files (*.*)|*.*"
CommonDialog1.FilterIndex = 1
CommonDialog1.Flags = cdlOFNAllowMultiselect Or _
    cdlOFNFileMustExist Or cdlOFNExplorer
CommonDialog1.MaxFileSize = 10240
CommonDialog1.CancelError = True
CommonDialog1.Filename = ""
CommonDialog1.ShowOpen
If Err = 0 Then
     ' Parse the result into an array of strings
     Dim names() As String, i As Integer
     names() = Split(CommonDialog1.Filename, vbNullChar)
     ' Print file names, including path
     If UBound(names) = 0 Then
          ' only one file was selected
          Print names(0)
     Else
          ' multiple files were selected
          For i = 1 To UBound(names)
               Print names(0) & "\" & names(i)
          Next
     End If
End If
```

—Francesco Balena, Bari, Italy

VB5, VB6

Level: Beginning

HANDLING THE SYSINFO CONTROL

You can use the SysInfo control, distributed with VB5 and VB6, to write applications that can sport the Windows logo and that can behave intelligently when a system setting changes. The control fires the DisplayChanged event when the screen resolution changes, and it fires the SysColorChange event when the user modifies one or more system colors in the Control Panel.

For example, when you have a maximized form and the user switches to a higher screen resolution, VB correctly resizes the form to occupy a larger screen area. However, when the user switches to a lower resolution, VB doesn't resize the form accordingly. This code does the trick:

```
Private Sub SysInfo1_DisplayChanged()
     ' If the form is maximized, restore it and
     ' maximize it again
     With Me
          If .WindowState = vbMaximized Then
               .Visible = False
               .WindowState = vbNormal
               .WindowState = vbMaximized
               .Visible = True
          End If
     End With
End Sub
```

For more information on this topic, look in VB's Help file under "SysInfo."

—Francesco Balena, Bari, Italy

VB5, VB6
Level: Beginning

Take Taskbar Into Account When Resizing Forms

The SysInfo control lets you resize your forms to take any taskbar into account. For example, you might want to move and resize your form so it always appears at the bottom of the work area—that is, the area not taken by the Windows or Office taskbar or any other taskbar currently visible:

```
Private Sub Form_Load()
    MoveForm
End Sub
' this form is 1000 twips tall, and is located
' near the bottom border of the workarea
Sub MoveForm()
    With SysInfo1
        Me.Move .WorkAreaLeft, _
            .WorkAreaTop + .WorkAreaHeight _
            - 1000, .WorkAreaWidth, 1000
    End With
End Sub
```

To move and resize the form automatically when the user moves the taskbars, creates new taskbars, or hides them, you have to trap the SysInfo control's SettingChanged event:

```
Private Sub SysInfo1_SettingChanged(ByVal Item As Integer)
    Const SPI_SETWORKAREA = 47
    If Item = SPI_SETWORKAREA Then
        MoveForm
    End If
End Sub
```

For more information on this topic, look in VB's Help file under "SysInfo."

—Francesco Balena, Bari, Italy

VB4 16/32, VB5, VB6
Level: Beginning

Use TypeName Instead of TypeOf…Is

To write reusable routines that work with multiple types of controls, test the control type using the TypeName function in place of the TypeOf…Is statement. For example, take a look at this routine—you can reuse it in another project only if you also add the RichTextBox control to the Components list:

```
' save the selected text to an open file
' works with TextBox and RichTextBox controls
Sub SaveSelectedText(ctrl As Control, filenum As Integer)
    If TypeOf ctrl Is TextBox Then
        Print #filenum, ctrl.SelText
    ElseIf TypeOf ctrl Is RichTextBox Then
        Print #filenum, RichTextBox1.SelRTF
    End If
End Sub
```

To avoid this problem and gain additional benefits such as the ability to use a Select Case block, use the TypeName function instead:

```
Sub SaveSelectedText(ctrl As Control, filenum As Integer)
    Select Case TypeName(ctrl)
        Case "TextBox"
            Print #filenum, ctrl.SelText
        Case "RichTextBox"
            Print #filenum, RichTextBox1.SelRTF
    End Select
End Sub
```

—Francesco Balena, Bari, Italy

VB4 16/32, VB5, VB6
Level: Beginning

Don't Use VarType to Test for Objects

To test whether an argument passed to a procedure is an object or something else, use the IsObject function in place of the VarType function. Consider this routine:

```
Sub TestType(arg As Variant)
    If VarType(arg) = vbObject Then
        Print "It's an object"
    ElseIf VarType(arg) = vbString Then
        Print "It's a string"
    End If
End Sub
```

If you pass a control or an object that exposes a default property, the routine incorrectly reports the default property type:

```
TestType Text1    ' displays "It's a string"
```

This is the correct way to test for an object:

```
    If IsObject(arg) Then
        Print "It's an object"
    ElseIf VarType(arg) = vbString Then
        Print "It's a string"
    End If
```

For more information on this topic, look in VB's Help file under "VarType."

—Francesco Balena, Bari, Italy

VB5, VB6
Level: Intermediate

Read the Serial Number of a Disk

The new Microsoft Scripting Runtime library includes a FileSystemObject hierarchy containing several objects that let you obtain information about your drives, folders, and files. For example, you can retrieve the serial number of a disk using this code:

```
' Get the serial number of drive C:
Dim fso As New Scripting.FileSystemObject
Dim dr As Scripting.Drive
' Get a reference to the Drive object
Set dr = fso.GetDrive("C")
Print Hex$(dr.SerialNumber)
```

You can also easily check whether a drive has enough free space on it, using the Drive object's FreeSpace property:

```
Print "Drive C: has " & dr.FreeSpace " bytes free."
```

For more information, look in VB's Help file under "Dictionary" and "FileSystemObject."

—Francesco Balena, Bari, Italy

VB3, VB4 16/32, VB5, VB6
Level: Beginning

Check Inequality of Database Field Values

Conventional wisdom suggests this code works fine when checking a field in a database—or any Variant, for that matter—for inequality to some other value:

```
If ![Name] <> "abc" Then
     Debug.Print "<>"
Else
     Debug.Print "="
End If
```

In reality, however, this isn't always the case. For example, if the field you're checking is of value Null, then the test also returns Null, invoking the Else clause. A quick, though perhaps obscure solution, is to check for equality instead:

```
If ![Name] = "abc" Then
     Debug.Print "="
Else
     Debug.Print "<>"
End If
```

—Andrew L. Ayers, Phoenix, Arizona

VB3, VB4 16/32, VB5, VB6
Level: Beginning

Make the Ants March

To implement a quick and easy version of the old "marching ants" line around a control, place two CommandButtons, a Shape, and a Timer control on a form. Then insert this code into the form:

```
Private Sub Command1_Click()
     StartMarchingAnts
End Sub
Private Sub Command2_Click()
     StopMarchingAnts
End Sub
Private Sub StartMarchingAnts()
     Timer1.Interval = 200
     Timer1.Enabled = True
     Shape1.Visible = True
End Sub
Private Sub StopMarchingAnts()
     Timer1.Enabled = False
     Shape1.Visible = False
End Sub
Private Sub Timer1_Timer()
     If Shape1.BorderStyle = vbBSDot Then
          Shape1.BorderStyle = vbBSDashDot
     Else
          Shape1.BorderStyle = vbBSDot
     End If
End Sub
```

Pressing Command1 starts the animation; Command2 stops it. This works for all shape types.

—George Hughen, Hamilton, Massachusetts

VB3, VB4 16/32, VB5, VB6
Level: Beginning

Ensure Foolproof Data Entry Times

Asking users to enter time values can often lead to problems formatting what the user enters. I use this piece of code to create a drop-down list of hours, minutes, and AM/PM, which ensures the user specifies the proper time and eliminates problems manipulating the time later in the program.

To use this example, create a combo-box array with three combo boxes: lower bound = 0; upper bound = 2; and a command button. Use this code:

```
Private Sub Form_Load()
     Dim i As Integer, strTemp As String
     'Fill hour combo box
     For i = 12 To 1 Step -1
          Combo1(0).AddItem i
     Next i
     'Fill minutes combo box
     For i = 0 To 59
          strTemp = i
          If Len(strTemp) = 1 Then strTemp = "0" & i
          Combo1(1).AddItem ":" & strTemp
     Next i
     'Fill AM/PM combo box
     Combo1(2).AddItem "AM"
     Combo1(2).AddItem "PM"
     'First item in each combo box is displayed
     For i = Combo1.LBound To Combo1.UBound
          Combo1(i).ListIndex = 0
     Next i
End Sub
Private Sub Command1_Click()
     Dim sTime
     'Format the time VB style
     sTime = "#" & Combo1(0) & Combo1(1) & ":00 " _
          & Combo1(2) & "#"
     MsgBox sTime
End Sub
```

—Jason Natale, Mississauga, Ontario, Canada

VB4 32, VB5, VB6
Level: Intermediate

Implement a ListView ItemDoubleClick Event

Double-clicking an icon or file name in Explorer is the standard way of launching an application in Windows. However, if you're developing an app that uses the ListView control from the Microsoft Windows Common Controls library (COMCTL32.ocx), this functionality is not directly exposed through an event. You have a DoubleClick event for the ListView control, but this event gets raised when a user double-clicks *anywhere* on the control. You also have an ItemClick event, but this event is only fired for a single-click on a ListItem object. Wouldn't it be nice to have an ItemDoubleClick event?

Use the ListView's MouseUp event to trap the X and Y coordinates of where the user last clicked the mouse. Here's a way to implement this functionality in your code:

```
Option Explicit
Private sngListViewX As Single
Private sngListViewY As Single
Private Sub ListView1_MouseUp(Button As Integer, Shift As _
     Integer, x As Single, y As Single)
```

```
    sngListViewX = x
    sngListViewY = y
End Sub
```

After trapping these coordinates, pass them to the HitTest method of the ListView control during the DoubleClick event to determine whether a user has double-clicked on a particular ListItem object:

```
Private Sub ListView1_DblClick()
    Dim lListItem As ListItem
    Set lListItem = ListView1.HitTest(sngListViewX, _
        sngListViewY)
    If (lListItem Is Nothing) Then
        MsgBox "You did not double-click on a ListItem."
    Else
        MsgBox "You double-clicked on ListItem=" & _
            lListItem.Text
    End If
    Set lListItem = Nothing
End Sub
```

—Dwayne Bradley, Mooresville, North Carolina

VB4 16/32, VB5, VB6

Level: Intermediate

USE CUSTOM FORM PROPERTIES

You can easily find out whether your user clicked on OK or on Cancel on a modal dialog. This example also prevents the user from unloading the form, and thereby prevents you from inadvertently reloading the form when you reference the properties of controls on the form:

```
Option Explicit
Private mUserHitOK As Boolean
Public Property Get UserHitOK() As Boolean
    UserHitOK = mUserHitOK
End Property
Private Sub cmdCancel_Click()
    mUserHitOK = False
    Call Hide
End Sub
Private Sub cmdOK_Click()
    mUserHitOK = True
    Call Hide
End Sub
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode _
    As Integer)
    If UnloadMode = vbFormControlMenu Then
        Cancel = True
        cmdCancel.Value = True
    End If
End Sub
```

You only need this code to check the user action:

```
Call frmModalDialog.Show(vbModal)
If frmModalDialog.UserHitOK Then
    ' Do Something Here
Else
    ' Do Something Else Here
End If
Call Unload(frmModalDialog)
Set frmModalDialog = Nothing
```

—Thomas Weiss, received by e-mail

VB4 32, VB5, VB6

Level: Intermediate

RETRIEVING A CONTROL FROM THE CONTROLS COLLECTION WITH AN HWND

The GetDlgCtrlID API, when passed a valid hWnd, returns a value that directly corresponds to the Index property of the Controls collection:

```
Private Declare Function GetDlgCtrlID Lib "user32" _
    (ByVal hWnd As Long) As Long
Private Sub Form_Load()
    Dim i As Long
    On Error Resume Next
    For i = 0 To Controls.Count - 1
        Debug.Print Controls(i).Name,
        Debug.Print _
          Controls(GetDlgCtrlID(Controls(i).hWnd) - 1).Name
    Next i
End Sub
```

This loop, located in the Form_Load event of a form with a number of controls on it, loops through all the controls and prints the name of each windowed control twice, demonstrating that it has correctly located the control without looping through the control collection.

—Jeremy Adams, Tiverton, Devon, United Kingdom

VB4 32, VB5, VB6

Level: Advanced

CREATE SEE-THROUGH CONTROLS

Most standard controls send WM_CTLCOLORXXXXX messages to their parent when they're about to draw themselves. VB normally handles these messages and responds appropriately with the ForeColor and BackColor properties that you have set for the control. However, it's possible to override the standard behavior and achieve a transparent background with several basic control types. This example uses the ubiquitous MsgHook for subclassing, but you can use whichever method you prefer. You must ensure that ClipControls is set to False for the form; otherwise, you'll see whatever is below your form as the background for the controls instead of the background bitmap:

```
Option Explicit
Private Declare Function SetBkMode Lib "gdi32" (ByVal hdc _
    As Long, ByVal nBkMode As Long) As Long
Private Declare Function SetTextColor Lib "gdi32" _
    (ByVal hdc As Long, ByVal crColor As Long) As Long
Private Declare Function GetStockObject Lib "gdi32" _
    (ByVal nIndex As Long) As Long
Private Const WM_CTLCOLORSTATIC = &H138
Private Const TRANSPARENT = 1
Private Const NULL_BRUSH = 5
Private Sub Form_Load()
    ' Me.ClipControls = False
    ' Must be set at design-time!
    Msghook1.HwndHook = Me.hWnd
    Msghook1.Message(WM_CTLCOLORSTATIC) = True
End Sub
Private Sub Msghook1_Message(ByVal msg As Long, ByVal wp _
    As Long, ByVal lp As Long, result As Long)
    Select Case msg
        Case WM_CTLCOLORSTATIC
            ' Call the original windowproc to handle the
            ' foreground color for the Controls etc.
            Call Msghook1.InvokeWindowProc(msg, wp, lp)
            ' Set the background mode to transparent
            Call SetBkMode(wp, TRANSPARENT)
            ' Get the stock null brush and return it
            ' The brush does nothing when the control
            ' paints using it, hence giving the
            ' transparency effect
            result = GetStockObject(NULL_BRUSH)

        Case Else
            ' [Replace this line with your own code
            ' to call the original windowproc]
            result = Msghook1.InvokeWindowProc(msg, wp, lp)
    End Select
End Sub
```

This code works for option buttons and checkboxes. I haven't found any side effects yet. You can make other controls transparent in a similar way, but some of them—including textboxes and listboxes—don't work correctly because the background doesn't get erased. You can probably get them to work correctly with additional code. Frames are even harder to get to work correctly; I resorted to using a button created using CreateWindowEx with the BS_GROUPBOX style. Even then, I had to return a brush of approximately the color of the background image; otherwise, you could see the line under the text for the frame.

—Jeremy Adams, Tiverton, Devon, United Kingdom

VB4 16/32, VB5, VB6
Level: Beginning

Loop on Non-Numeric Indices
You might occasionally need to execute a group of statements with different and unrelated values of a variable. For example, say you need to verify that a number isn't a multiple of 2, 3, 5, 7, or 11. In these circumstances, you can't use a regular For…Next loop, unless you store these values in a temporary array. Here's a more concise solution:

```
Dim n As Variant
For Each n In Array(2, 3, 5, 7, 11)
    If (TestNumber Mod n) = 0 Then
        Print "Not prime"
        Exit For
    End If
Next
```

You can use the same technique to iterate on non-numeric values:

```
' check if a string embeds a shortened weekday name
Dim d As Variant
For Each d In Array("Sun", "Mon", "Tue", "Wed", "Thu", _
    "Fri", "Sat")
    If Instr(1, TestString, d, vbTextCompare) Then
        Print "Weekday = " & d
        Exit For
    End If
Next
```

—Francesco Balena, Bari, Italy

VB6
Level: Beginning

Make Sure Data is Updated
When using bound controls, data is updated automatically when you Move to a different record. Data is not updated when you close the form. To ensure that data is saved when you close, perform a Move 0. This saves the changes before unloading the form.

—Deborah Kurata, Pleasanton, California

VB4 32, VB5, VB6
Level: Advanced

Is the Active Desktop Active?
Sometimes you want to know if the desktop is in Active Desktop mode—for example, to set an HTML wallpaper. I couldn't find a function to accomplish this, but this hack works on all Windows 95/98 and NT4 desktops that I've tested it on:

```
Private Declare Function FindWindow& Lib "user32" Alias _
    "FindWindowA" (ByVal lpClassName$, ByVal lpWindowName$)
Private Declare Function FindWindowEx& Lib "user32" Alias _
    "FindWindowExA" (ByVal hWndParent&, ByVal _
    hWndChildAfter&, ByVal lpClassName$, ByVal _
    lpWindowName$)

Public Function IE4ActiveDesktop() As Boolean
    Dim Templong&
    Templong = FindWindow("Progman", vbNullString)
    Templong = FindWindowEx(Templong, 0&, _
        "SHELLDLL_DefView", vbNullString)
    Templong = FindWindowEx(Templong, 0&, _
        "Internet Explorer_Server", vbNullString)
    If Templong > 0 Then
        IE4ActiveDesktop = True
    Else
        IE4ActiveDesktop = False
    End If
End Function
```

—Don Bradner, Arcata, California

VB4 32, VB5, VB6
Level: Advanced

Is the NT Screen Saver Running?
Use this code to determine whether NT is running on its screen saver desktop. NT5 has an SPI function, but this code should work on any NT version:

```
Private Declare Function OpenDesktop& Lib "user32" Alias _
    "OpenDesktopA" (ByVal lpszDesktop$, ByVal dwFlags$, _
    ByVal fInherit&, ByVal dwDesiredAccess&)
Private Declare Function CloseDesktop& Lib "user32" _
    (ByVal hDesktop&)
Public Function NTSaverRunning() As Boolean
    Dim hDesktop As Long
    Const MAXIMUM_ALLOWED = &H2000000
    If winOS <> WinNT Then
    'Make your OS determination elsewhere
        NTSaverRunning = False
        Exit Function
    End If
    NTSaverRunning = False
    hDesktop = OpenDesktop("screen-saver", 0&, 0&, _
        MAXIMUM_ALLOWED)
    If hDesktop = 0 Then
        If Err.LastDllError = 5 Then
            NTSaverRunning = True
        End If
    Else
        Templong = CloseDesktop(hDesktop)
        NTSaverRunning = True
    End If
End Function
```

—Don Bradner, Arcata, California

VB4 32, VB5, VB6

Level: Intermediate

Grab System Fonts Easily

At times, you might want to retrieve the current system font settings, such as the font being used for window title bars, or the menu or message box font. You could delve into the Registry, but why go to the trouble if the SystemParametersInfo API does it for you? Here's how:

```
Private Declare Function SystemParametersInfo Lib "user32" _
    Alias "SystemParametersInfoA" (ByVal uAction As Long, _
    ByVal uParam As Long, lpvParam As Any, ByVal fuWinIni _
    As Long) As Long
    Private Type LOGFONT
        lfHeight As Long
        lfWidth As Long
        lfEscapement As Long
        lfOrientation As Long
        lfWeight As Long
        lfItalic As Byte
        lfUnderline As Byte
        lfStrikeOut As Byte
        lfCharSet As Byte
        lfOutPrecision As Byte
        lfClipPrecision As Byte
        lfQuality As Byte
        lfPitchAndFamily As Byte
        lfFaceName As String * 32
    End Type
    Private Type NONCLIENTMETRICS
        cbSize As Long
        iBorderWidth As Long
        iScrollWidth As Long
        iScrollHeight As Long
        iCaptionWidth As Long
        iCaptionHeight As Long
        lfCaptionFont As LOGFONT
        iSMCaptionWidth As Long
        iSMCaptionHeight As Long
        lfSMCaptionFont As LOGFONT
        iMenuWidth As Long
        iMenuHeight As Long
        lfMenuFont As LOGFONT
        lfStatusFont As LOGFONT
    lfMessageFont As LOGFONT
    End Type
    Private Const SPI_GETNONCLIENTMETRICS = 41
    Public Function GetCaptionFont() As String
        Dim NCM As NONCLIENTMETRICS
        NCM.cbSize = Len(NCM)
        Call SystemParametersInfo(SPI_GETNONCLIENTMETRICS, _
            0, NCM, 0)
        If InStr(NCM.lfCaptionFont.lfFaceName, Chr$(0)) _
            > 0 Then
            GetCaptionFont = _
                Left$(NCM.lfCaptionFont.lfFaceName, _
                InStr(NCM.lfCaptionFont.lfFaceName, _
                Chr$(0)) - 1)
        Else
            GetCaptionFont = NCM.lfCaptionFont.lfFaceName
        End If
    End Function
```

Keep in mind this function—GetCaptionFont—returns only the name of the font. However, all the other font information is there in the LOGFONT structures as well.

—Ben Baird, Twin Falls, Idaho

VB6

Level: Intermediate

Fast String Array Load and Save

VB6 offers a couple new string functions that work with string arrays. One of these new string functions, Join, concatenates all the items of an array into an individual string using the delimiter string of choice. This builds a routine that quickly saves the contents of a string array to disk without iterating on individual array items:

```
Sub StringArraySave(Filename As String, Text() As String)
    Dim f As Integer
    f = FreeFile
    Open Filename For Output As #f
    Print #f, Join(Text, vbCrLf);
    Close #f
End Sub
```

The Split function does the opposite, first splitting a longer string into individual components delimited by the selected separator, then loading the components into a string array. If you couple this feature with a VB6 function to return an array, you can easily build a routine that loads a text file into a string array:

```
Function StringArrayLoad(Filename As String) As String()
    Dim f As Integer
    f = FreeFile
    Open Filename For Input As #f
    StringArrayLoad = Split(Input$(LOF(f), f), vbCrLf)
    Close #f
End Function
```

Use this function like this:

```
Dim Text() As String
Text = StringArrayLoad("c:\autoexec.bat")
```

—Francesco Balena, Bari, Italy

VB3, VB4 16/32, VB5, VB6

Level: Intermediate

Encrypt a String Easily

This quick and dirty encryption/decryption function takes whatever string you pass it, assigns it to a byte array, Xor's each byte by a constant, then returns the string. The offsetting done on every other character adds just a little to the confusion. Passing a string through the function once encrypts it; passing it through a second time decrypts it.

This function won't fool anyone from the National Security

Agency, but it does protect the data from 99 percent of prying eyes, which is good enough for everything I ever need to do. It's only a few lines of code, but it encrypts as much data as you can fit into a string, and does so quickly:

```
    Private Sub Form_Click()
        Dim szTest As String
        szTest = "My dog has fleas."
        ''' Passing the string through the function once
        ''' encrypts it.
        szTest = szEncryptDecrypt(szTest)
        Debug.Print szTest

        ''' Passing the string through the function again
        ''' decrypts it.
        szTest = szEncryptDecrypt(szTest)
        Debug.Print szTest
    End Sub
Function szEncryptDecrypt(ByVal szData As String) As String
    ''' This key value can be changed to alter the
    ''' encryption, but it must be the same for both
    ''' encryption and decryption.
    Const KEY_TEXT As String = "ScratchItYouFool"
    ''' The KEY_OFFSET is optional, and may be any
    ''' value 0-64.
    ''' Likewise, it needs to be the same coming/going.
    Const KEY_OFFSET As Long = 38
    Dim bytKey() As Byte
    Dim bytData() As Byte
    Dim lNum As Long
    Dim szKey As String
    For lNum = 1 To ((Len(szData) \ Len(KEY_TEXT)) + 1)
        szKey = szKey & KEY_TEXT
    Next lNum
    bytKey = Left$(szKey, Len(szData))
    bytData = szData
    For lNum = LBound(bytData) To UBound(bytData)
        If lNum Mod 2 Then
            bytData(lNum) = bytData(lNum) Xor (bytKey(lNum) _
                + KEY_OFFSET)
        Else
            bytData(lNum) = bytData(lNum) Xor (bytKey(lNum) _
                - KEY_OFFSET)
        End If
    Next lNum
    szEncryptDecrypt = bytData
End Function
```

—Rob Bovey, Edmonds, Washington

VB6
Level: Beginning

COUNTING THE OCCURRENCES OF A SUBSTRING

VB6 has introduced the Replace function, which replaces all occurrences of a substring with another substring. Although this function is useful in itself, you can also use it in unorthodox ways. For instance, you can use it to count how many times a substring appears inside another string:

```
Function InstrCount(Source As String, Search As String) _
    As Long
    InstrCount = Len(Source) - Len(Replace(Source, Search, _
        Mid$(Search, 2)))
End Function
```

This code uses the Replace function to replace the searched substring with another substring one character shorter. This means the difference between the original string and the string returned by the Replace function is equal to the number of occurrences of the substring.

—Francesco Balena, Bari, Italy

VB4 32, VB5, VB6
Level: Advanced

FLY THE FLAG

The clock applet that comes with Microsoft Plus! has an interesting feature: Its window is round instead of rectangular. Surprisingly, giving your form an odd shape is easy. Add this code to a new form to give your window the shape of the Microsoft Windows logo:

```
Private Type RECT
    Left As Long
    Top As Long
    Right As Long
    Bottom As Long
End Type
Private Declare Function BeginPath Lib "gdi32" _
    (ByVal hdc As Long) As Long
Private Declare Function TextOut Lib "gdi32" _
    Alias "TextOutA" (ByVal hdc As Long, _
    ByVal X As Long, ByVal Y As Long, _
    ByVal lpString As String, _
    ByVal nCount As Long) As Long
Private Declare Function EndPath Lib "gdi32" _
    (ByVal hdc As Long) As Long
Private Declare Function PathToRegion Lib "gdi32" _
    (ByVal hdc As Long) As Long
Private Declare Function GetRgnBox Lib "gdi32" _
    (ByVal hRgn As Long, lpRect As RECT) As Long
Private Declare Function CreateRectRgnIndirect Lib "gdi32" _
    (lpRect As RECT) As Long
Private Declare Function CombineRgn Lib "gdi32" _
    (ByVal hDestRgn As Long, ByVal hSrcRgn1 As Long, _
    ByVal hSrcRgn2 As Long, _
    ByVal nCombineMode As Long) As Long
Private Const RGN_AND = 1
Private Declare Function DeleteObject Lib "gdi32" _
    (ByVal hObject As Long) As Long
Private Declare Function SetWindowRgn Lib "user32" _
    (ByVal hwnd As Long, ByVal hRgn As Long, _
    ByVal bRedraw As Boolean) As Long
Private Declare Function ReleaseCapture Lib "user32" _
    () As Long
Private Declare Function SendMessage Lib "user32" _
    Alias "SendMessageA" (ByVal hwnd As Long, _
    ByVal wMsg As Long, ByVal wParam As Long, _
    lParam As Any) As Long
Private Const WM_NCLBUTTONDOWN = &HA1
Private Const HTCAPTION = 2
Private Function GetTextRgn() As Long
    Dim hRgn1 As Long, hRgn2 As Long
    Dim rct As RECT
    'Create a path for the window's shape
    BeginPath hdc
    TextOut hdc, 10, 10, Chr$(255), 1
    EndPath hdc
    '... Convert the path to a region...
    hRgn1 = PathToRegion(hdc)
    GetRgnBox hRgn1, rct
    hRgn2 = CreateRectRgnIndirect(rct)
    CombineRgn hRgn2, hRgn2, hRgn1, RGN_AND
    'Return the region handle
    DeleteObject hRgn1
    GetTextRgn = hRgn2
End Function
Private Sub Form_DblClick()
    'Need to be able to close the form
    Unload Me
End Sub
Private Sub Form_Load()
    Dim hRgn As Long
    Me.Font.Name = "Wingdings"
    Me.Font.Size = 200
    hRgn = GetTextRgn()
    SetWindowRgn hwnd, hRgn, 1
End Sub
Private Sub Form_MouseDown(Button As Integer, Shift _
    As Integer, X As Single, Y As Single)
    'Give us some way to move the form
    ReleaseCapture
    SendMessage hwnd, WM_NCLBUTTONDOWN, HTCAPTION, ByVal 0&
End Sub
```

While this is a sort of novelty shape for a form, you can give a form any shape you want, provided you have a way to create the shape of the region. Look at the various region-related API calls to find methods of creating regions other than using font characters.

—Ben Baird, Twin Falls, Idaho

VB6
Level: Beginning

Dictionary Has Advantages Over Collections

The Dictionary object can contain items, which can be any form of data, just like a collection. However, the Dictionary object has many advantages over a collection. It allows retrieval of the key using the keys(index) syntax. It features an Exists property that determines whether a particular key exists. It allows for the changing of a key, as well as the changing of a value associated with a key. The Dictionary object is zero-based, and does not provide an enumerator.

Also, note that to use the Dictionary object, you must set a reference to the Microsoft Scripting Runtime.

For more information, look in VB's Help file under "Dictionary" and "FileSystemObject."

—Deborah Kurata, Pleasanton, California

VB3, VB4 16/32, VB5, VB6
Level: Beginning

Watch the Parens

If you want to pass a parameter to a subroutine, use this code:

```
    Call doFormat(txtPerson)
```

You can also call the subroutine without the Call statement. However, if you don't include the Call statement, you can't include parentheses:

```
    doFormat (txtPerson)
```

In VB, expressions in parentheses are evaluated before they're processed. So by putting parentheses around the control name, you're telling it to evaluate it. Because a control can't be evaluated, it gives you the value of the default property. This code actually passes the Text string value—because Text is the default property—to the subroutine instead of passing the control. Because the routine expects a textbox and not a string, it generates the type mismatch.

—Deborah Kurata, Pleasanton, California

VB4 16/32, VB5, VB6
Level: Intermediate

Display Two Fields

Let's say you want to bind your listbox to a recordset, but you also want to display two fields concatenated together, such as "Jones, Barbara." You can do this by using a calculated expression in the SQL statement of the query:

```
SELECT PersonID, LastName, FirstName, LastName + ', ' _
    + FirstName AS FullName FROM Person ORDER BY _
    LastName, FirstName
```

—Deborah Kurata, Pleasanton, California

VB3, VB4 16/32, VB5, VB6
Level: Beginning

Use Select Case to Evaluate Different Expressions

Select Case is commonly used to check different values of a specific variable or expression. You can also use Select Case to evaluate multiple expressions, by starting out with "Select Case True" and listing each expression as a separate "Case":

```
Select Case True
    Case Option1(0).Value
        'Do something
    Case Option1(1).Value
        'Do something
    Case Option1(2).Value
        'Do something
End Select
```

—Russell Davis, Garden Grove, California

VB6
Level: Beginning

Compare Floating Point Values Using the Round Function

When you have to compare the results of floating point expressions, you can't rely on the "=" operator due to the finite precision of Single or Double variables. To see this concept demonstrated, use this code:

```
Dim i As Integer, d As Double
For i = 1 To 10
    d = d + CDbl(0.1)
Next
MsgBox (d = 1)    ' displays "False"
```

To more easily compare two floating numbers, use the new VB6 Round function, which rounds a number to the desired number of decimal digits. For example, you can rewrite the previous test like this:

```
' the difference is less than 1E-12
MsgBox (Round(d, 12) = 1)  ' displays "True"
```

You can also use this method to check whether A and B variables contain values that match up to their 12th decimal digit:

```
If Round(A - B, 12) = 0 Then Print "Equal"
```

—Francesco Balena, Bari, Italy

VB4 32, VB5, VB6
Level: Beginning

JOIN TWO FILES TOGETHER

The DOS Copy command allows you to take the contents of two different files and put them one after the other into a third file. You can do the same thing in VB with this subroutine:

```
Public Sub JoinFiles(Source1 As String, Source2 As String, _
     Dest As String)
     Dim Buffer() As Byte
     Open Source1 For Binary Access Read As #1
     Open Source2 For Binary Access Read As #2
     Open Dest For Binary Access Write As #3
     ReDim Buffer(1 To LOF(1))
     Get #1, , Buffer
     Put #3, , Buffer
     ReDim Buffer(1 To LOF(2))
     Get #2, , Buffer
     Put #3, , Buffer
     Close #1, #2, #3
End Sub
```

In a production app, use FreeFile rather than hard-code the file handles.

—Russell Davis, Garden Grove, California

VB5, VB6
Level: Advanced

SUBCLASS GRID CONTROLS

Sometimes a class needs to communicate with the object that created it. For example, I use a class to subclass grid controls so I can handle things such as tabbing between columns and unbound population automatically. The class has a SubclassGrid method that makes the subclassing mechanism work, and at the same time saves a copy of the creator object:

```
Dim WithEvents m_InternalGrid As DBGrid
Dim m_ParentForm As Form
Public Sub SubclassGrid(AnyGrid As DBGrid)
     Set m_InternalGrid = AnyGrid
     Set m_ParentForm = AnyGrid.Parent
End Sub
```

In some instances, the class needs to get information from the form using the class. To get information from the form, the class can fire an event, or the form can have a Public property, function, or method that the class can call. However, both mechanisms are optional. The creator of the form can forget to write code to handle the events, or forget to add a public function, or even name it differently from what the class expects. To prevent this, I write an IGridCallback interface definition:

```
IGridCallback
Public Function GetTableName() As String
End Function
```

I change my SubclassGrid function in the class to look like this:

```
Dim WithEvents m_InternalGrid As DBGrid
Dim m_ParentForm As Form
Public Sub SubclassGrid(AnyGrid As DBGrid)
     Dim pIGridCallback As IGridCallback
     Set m_InternalGrid = AnyGrid
     Set m_ParentForm = AnyGrid.Parent
     On Error Resume Next
     Set pIGridCallback = m_ParentForm
     If Err.Number <> 0 Then
          MsgBox "Your form must implement the " & _
               "IGridCallback interface in order to " & _
               "use the SubclassGrid class"
     End If
End Sub
```

The SubclassGrid routine displays an error message box if it doesn't find the IGridCallback interface in the form using it. This mechanism guarantees that everyone using the SubclassGrid class must implement the IGridCallback interface. Because implementing the interface means you must support each method in the interface, it also means everyone using the class has the correct functions.

—Jose Mojica, Davie, Florida

VB4 32, VB5, VB6
Level: Intermediate

ACCESS THE TREEVIEW CONTROL'S CURRENT NODE PROPERTIES

The TreeView control reports its current node—the selected node—only in its NodeClick event's Node parameter. If you need to access the Node's properties—such as Key or Text—outside this event, you need to declare a Node variable with scope appropriate to your intended usage. To share within a form, include a variable declaration in the Declarations section:

```
Option Explicit
Private CurrentNode As node
```

Set your variable—in this illustration, CurrentNode—to the node passed in the event:

```
Private Sub tvwDB_NodeClick(ByVal node As node)
Set CurrentNode = node
```

You can now access CurrentNode's properties from anywhere on the form:

```
Debug.Print CurrentNode.Key
Debug.Print CurrentNode.Text
```

—Ron Schwarz, Mt. Pleasant, Michigan

VB3, VB4 16/32, VB5, VB6
Level: Beginning

AVOID UNWANTED RECURSION FROM EVENT CASCADES
Sometimes, an event might fire while it's already executing from a prior invocation. To prevent this unwanted behavior, assign a static variable within the procedure, and test it before allowing any more code in the procedure to execute. Then set the variable to True at the start of the main block of the procedure code. When your code finishes, set it to False. This prevents a new instance of the procedure from being invoked while it's already executing. You might want to add additional code in the test block to deal with situations where you need to do more than simply cancel the execution:

```
Private Sub Form_Resize()
    Static Executing As Boolean
    If Executing Then
        Exit Sub
    End If
    Executing = True
    If Width > 6000 Then
        Width = 6000
        GoDoSomeStuff
    End If
    Executing = False
End Sub
```

—Ron Schwarz, Mt. Pleasant, Michigan

VB4 32, VB5, VB6
Level: Beginning

CLEAN UP PROJECT FILE PATHS BEFORE SHARING SOURCE CODE
As you work with a VB project, the project file—VBP—can become littered with relative path references such as "..\..\..\..\myfolder\myform.frm". The project loads, but only on *your* machine. If you send the project to someone else, or move it to another path on your own machine, you need to edit the project file to remove the ambiguous entries. You can avoid this by ensuring that all the needed files are indeed in the same directory as the project file. It's not uncommon to load a file from a different directory, in which case VB does *not* automatically move it into your project directory. Load the project file into Notepad and edit out all path references, leaving only the actual file names. When VB goes to load the project, it looks for them in the current directory.

—Ron Schwarz, Mt. Pleasant, Michigan

VB3, VB4 16/32, VB5, VB6
Level: Beginning

USING THE FORMAT FUNCTION WITH STRINGS
You'll use the Format function most often with numbers, but it can be useful when applied to strings as well. For example, you can format a credit card number—which is held in a string variable, even if it contains only digits—and subdivide the number into four groups of four characters each, using a complex string expression:

```
' X holds the sequence of 16 digits
CreditCardNum = Left$(x, 4) & " " & Mid$(x, 5, 4) & " " & _
    Mid$(x, 9, 4) & " " & Right$(x, 4)
```

The Format function lets you accomplish the same result in a more readable and efficient way:

```
CreditCardNum = Format$(x, "!@@@@ @@@@ @@@@ @@@@")
```

—Francesco Balena, Bari, Italy

VB6
Level: Intermediate

USE WINDOWLESS CONTROLS FOR PERFORMANCE AND ECONOMY
VB6 comes with an ActiveX control named MSWLESS.ocx ("Microsoft Windowless Controls 6.0"), which contains lightweight equivalents for these standard controls:

WLText
WLFrame
WLCommand
WLCheck
WLOption
WLCombo
WLList
WLHScroll
WHVScroll

The controls have lower resource consumption than their regular counterparts, as well as better performance. Because they don't have windows, they don't have handles or DDE capability, and they can't serve as containers. They're also not installed automatically in the Visual Studio 6 setup, so you'll need to dig them off your CD manually.

—Ron Schwarz, Mt. Pleasant, Michigan

VB5, VB6
Level: Intermediate

TESTING COM INTERFACES AT RUN TIME
VB5 provides interface inheritance through the use of the Implements keyword. For example, CFullTimeEmployee can implement the IEmployee interface. This interface might include basic information such as name, Social Security number, and date of birth. Another class, CPartTimeEmployee, can also implement the IEmployee interface. You can then write code against the IEmployee interface without regard to the type of employee. To supply additional functionality, you might create an IEmp2 interface. To test whether an object is of a certain type, use the TypeOf keyword at run time. The format is "TypeOf object Is class/interface". Here's how to define two classes:

```
Class CFullTimeEmployee:
Implements IEmployee
Implements IEmp2
Class CPartTimeEmployee
Implements IEmployee
```

```
Dim objMyFTE as New CFullTimeEmployee
Dim objMyPTE as New CPartTimeEmployee
```

Using TypeOf, you can query at run time which interfaces these objects support:

Query	Return
TypeOf objMyFTE Is CFullTimeEmployee	True
TypeOf objMyFTE Is IEmployee	True
TypeOf objMyFTE Is IEmp2	True
TypeOf objMyFTE Is CPartTimeEmployee	False
TypeOf objMyFTE Is Object	True
TypeOf objMyFTE Is IUnknown	True
TypeOf objMyPTE Is CPartTimeEmployee	True
TypeOf objMyPTE Is IEmployee	True
TypeOf objMyPTE Is IEmp2	False
TypeOf objMyPTE Is Object	True
TypeOf objMyPTE Is IUnknown	True
TypeOf objMyPTE Is CFullTimeEmployee	False

—Mark Tucker, Gilbert, Arizona

VB5
Level: Intermediate

USING LABEL CONTROL AS SPLITTER

Here's a demo for using a Label control as a splitter between two controls, as well as sample code for employing the splitter in an Explorer-like application:

```
Option Explicit
Private mbResizing As Boolean
    'flag to indicate whether mouse left
    'button is pressed down
Private Sub Form_Load()
    TreeView1.Move 0, 0, Me.ScaleWidth / 3, Me.ScaleHeight
    ListView1.Move (Me.ScaleWidth / 3) + 50, 0, _
        (Me.ScaleWidth * 2 / 3) - 50, Me.ScaleHeight
    Label1.Move Me.ScaleWidth / 3, 0, 100, Me.ScaleHeight
    Label1.MousePointer = vbSizeWE
End Sub
Private Sub Label1_MouseDown(Button As Integer, Shift As _
    Integer, X As Single, Y As Single)
    If Button = vbLeftButton Then mbResizing = True
End Sub
Private Sub Label1_MouseMove(Button As _
    Integer, Shift As Integer, X As _
    Single, Y As Single)
    'resizing controls while the left mousebutton is
    'pressed down
    If mbResizing Then
        Dim nX As Single
        nX = Label1.Left + X
        If nX < 500 Then Exit Sub
        If nX > Me.ScaleWidth - 500 Then Exit Sub
        TreeView1.Width = nX
        ListView1.Left = nX + 50
        ListView1.Width = Me.ScaleWidth - nX - 50
        Label1.Left = nX
    End If
End Sub
Private Sub Label1_MouseUp(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    mbResizing = False
End Sub
```

—Rajesh R. Vakharia, Mumbai, India

VB3, VB4 16/32, VB5, VB6
Level: Beginning

ALWAYS USE CASE ELSE WITH SELECT CASE

Select Case statements can lead to errors when the test expression in the Select Case line doesn't yield a true result for any of its individual Case expressions. Therefore, you should always use a Case Else as the final Case within a Select Case statement. This sample raises a custom error when an application invokes the Select Case statement with an operation other than the four basic ones:

```
Select Case Operation
    Case "Addition"
        Computer2 = dblNumber1 + dblNumber2
    Case "Subtraction"
        Computer2 = dblNumber1 - dblNumber2
    Case "Multiplication"
        Computer2 = dblNumber1 * dblNumber2
    Case "Division"
        Computer2 = dblNumber1 / dblNumber2
    Case Else
        Err.Raise 1, , "Wrong operation."
End Select
```

—Rick Dobson, Louisville, Kentucky

VB3, VB4 16/32, VB5, VB6
Level: Beginning

QUICKER TEXTBOX ADDITIONS

Consider these two examples of code that add a string to a textbox.

Example 1:

```
    Text1.text = Text1.text & MyString
    Text1.SelStart = Len(Text1.text)
```

Example 2:

```
    Text1.SelStart = Len(Text1.text)
    Text1.SelText = MyString
```

In the first example, you must copy the complete text from the textbox into a separate buffer to perform concatenation with the string MyString. You then need to copy the resulting string back to the textbox. This code requires allocating additional memory and performing two copy operations.

The code in the second example does not need an additional buffer. The concatenation of strings is executed inside the textbox buffer without transferring the text first outside the textbox and then back inside. This code is much faster and less memory-extensive. This second approach has an additional advantage of setting an insertion point directly at the end of the displayed text. In the first example, the insertion point is initially set to the beginning of the text, then transferred to the end, causing the control to flicker on the screen.

—Krystyna Zyzdryn, Palm Beach Gardens, Florida

VB4 32, VB5, VB6
Level: Intermediate

Move the Cursor to the Center

Use this simple subroutine to move the mouse/cursor to the center of an object. It's useful for tab-like functions and default settings:

```
Private Declare Function SetCursorPos& Lib _
    "user32" (ByVal x As Long, ByVal y As Long)
Private Declare Function GetWindowRect& Lib _
    "user32" (ByVal hwnd As Long, lpRect As Rect)
Private Type Rect
    left As Long
    top As Long
    right As Long
    bottom As Long
End Type
Public Sub SetMouseFocus(ByVal Obj As Object)
    Dim Rect As Rect
    'Get the bounding rectangle for window
    GetWindowRect Obj.hwnd, Rect
    'Set the cursor position to the center of the object
    SetCursorPos Rect.right - ((Rect.right - Rect.left) _
        / 2), Rect.bottom - ((Rect.bottom - Rect.top) / 2)
End Sub
```

Here's an example:

```
Private Sub cmdQuit_Click()
    ' Move Cursor to command button "cmdAreYouSure"
    SetMouseFocus cmdAreYouSure
End Sub
```

—Rich Myott, Colorado Springs, Colorado

VB4 16/32, VB5, VB6
Level: Intermediate

Assure Compatibility

When working with ActiveX DLLs, many VB programmers build a reference DLL first, copy it to a subdirectory of the project directory, and set version compatibility to binary with a pointer to the reference build. Many of these same programmers believe—or have been taught—they can now forget about compatibility because VB prevents them from ever building an incompatible version. They're dead wrong. Suppose you later add a method to one of the public classes in the project, which is defined as:

```
Public Sub DoSomething(Prm1 As String, Prm2 as Integer)
```

Version 2, which includes the new DoSomething method, is built without VB complaining, because the new build is version-compatible. For example, you might modify and recompile client applications to take advantage of the new method.

Suppose you change the Prm2 parameter of DoSomething from Integer to Long to prevent an overflow error. Ordinarily, such a change breaks backward compatibility. But, because the reference build is version 1, which doesn't have the DoSomething method, VB assumes it's completely new and happily builds version 3. The truth, however, is that this change does break compatibility, and all the client applications of version 2 crash with runtime error 430, "Class doesn't support Automation."

Changing a parameter's type isn't the only way a programmer can fool VB into thinking it's maintaining binary compatibility. In fact, removing or changing any method or property that wasn't in version 1 does it, as well as removing a class that wasn't in version 1.

When working with ActiveX DLLs, the only way to ensure ongoing compatibility is by updating your reference build after each change in the interface(s) it exposes. You should always point the reference to your last-shipped build, which assures that each version is compatible with the last-shipped version.

—Thomas Weiss, Deerfield, Illinois

VB4 16/32, VB5, VB6
Level: Beginning

Set Objects to Nothing

If you set objects to Nothing, then test their nothingness, this suggestion might help you. If you declare an object as New and later set it to Nothing, VB won't set the object to Nothing. For example, if you declare an object using this code:

```
Dim oSomeObject As New SomeClass
```

And later set it to Nothing using this code:

```
Set oSomeObject = Nothing
```

The object does terminate, but a new instance is immediately instantiated, so it appears the object hasn't been set to Nothing. You can test this by going to the Immediate window and testing '?(oSomeObject = Nothing)'. It displays False.

If you declare an object and explicitly set it to New, you can set the object to Nothing. For example, if you declare an object using this code, the object will be set to Nothing:

```
Dim oSomeObject As SomeClass
Set oSomeObject = New SomeClass
'.... some code ...
Set oSomeObject = Nothing
```

—Rahuldeo S. Vadodkar, Kentwood, Michigan

VB4 32, VB5, VB6
Level: Beginning

Scrub Out Unused Constants

Low-level parsing in VB doesn't come up every day. This routine demonstrates how to search a VB module for dead module-level constants with only a few lines of code. Reading the entire contents of a file into a string variable is key to this approach:

```
Private Sub FindDeadConstants(ByVal PathAndFile As String)
    Dim FileHandle As Integer
    Dim FileContents As String
    Dim PositionOfDeclaration As Long
    Dim StartOfConstantName As Long
    Dim EndOfConstantName As Long
    Dim ConstantName As String
    'open the file and read the contents
    'into a string variable:
    FileHandle = FreeFile
    Open PathAndFile For Binary Access Read As FileHandle
    FileContents = Input$(LOF(FileHandle), FileHandle)
    Close FileHandle
    'loop through all the module-level constants:
    Do
```

```
        PositionOfDeclaration = InStr(PositionOfDeclaration _
            + 1, FileContents, "Private Const ")
        If PositionOfDeclaration > 0 Then
            'we've found a constant:
            StartOfConstantName = PositionOfDeclaration _
                + Len("Private Const ")
            EndOfConstantName = InStr( _
                StartOfConstantName, FileContents, " ")
            ConstantName = Mid$(FileContents, _
                StartOfConstantName, EndOfConstantName _
                    - StartOfConstantName)
            'if the constant is not
            'referenced beyond its
            'declaration, then it's dead:
            If InStr(EndOfConstantName, FileContents, _
                ConstantName) = 0 Then
                lstDeadConstants.AddItem ConstantName
            End If
        End If
    Loop Until PositionOfDeclaration = 0
End Sub
```

—Dave Doknjas, Surrey, British Columbia, Canada

VB3, VB4 16/32, VB5, VB6
Level: Beginning

TRACK MOUSEPOINTER CHANGES

If you develop large apps, this routine can help you keep a count of calls to change the mousepointer. The routine, called MouseStack, keeps a stack of those calls and resets the pointer to the default when the stacks are equal or when a reset is forced.

Often, one routine sets the MousePointer to hourglass, then calls another routine that also sets it to hourglass, then back to default. However, you might not want it reset just yet. MouseStack() changes the pointer back only if the counts of calls to hourglass and default are equal, and/or the user calls MouseStack() with vbResetMouse to set the pointer to the default:

```
Public Enum MousePointers
    vbDefault = 0
    'set pointer to default
    vbHourglass = 11
    'set pointer to hourglass
    vbResetMouse = 99
    'reset static variables, set pointer to default
End Enum
Public Sub MouseStack(nMouseSetting As MousePointers)
    Static nHourglass As Integer
    Static nDefault As Integer
    ' Based on the setting entered in,
    'increment the proper variable
    Select Case nMouseSetting
        Case vbHourglass      ' 11
            nHourglass = nHourglass + 1
        Case vbDefault        ' 0
            nDefault = nDefault + 1
        Case Else
    End Select
    ' If the variables are now equal, or
    ' a reset was called, reset the
    ' variables, and set the mouse
    ' pointer to the default
    If (nHourglass = nDefault) Or (nMouseSetting _
        = vbResetMouse) Then
        nHourglass = 0
        nDefault = 0
        Screen.MousePointer = vbDefault
            Exit Sub
        End If
        ' If the difference is > 1, the
        ' pointer is already set to where it
        ' should be. Leave without setting
        ' pointer.
        If (Abs(nHourglass - nDefault) > 1) Then Exit Sub
        ' If one is ahead, set the
        ' mousepointer to it
        If (nHourglass > nDefault) Then
            Screen.MousePointer = vbHourglass
        Else
            Screen.MousePointer = vbDefault
        End If
End Sub
```

—T.J. (Tom) Gondesen, Tucker, Georgia

VB4 32, VB5, VB6
Level: Beginning

USE NATIVE TRACKSELECT IN TREEVIEW AND LISTVIEW

You can make your TreeView or ListView control behave like a menu so that as you move the mouse cursor over the items, the highlighted item moves with the cursor. Use this code in the TreeView or ListView's MouseMove event:

```
Private Sub TreeView1_MouseMove(Button _
    As Integer, Shift As Integer, x As Single, y As Single)
    Dim AnyNode As Node
    ' HitTest returns a node object under the cursor
    Set AnyNode = TreeView1.HitTest(x, y)

    If Not AnyNode Is Nothing Then
        Set TreeView1.DropHighlight = AnyNode
        TreeView1.DropHighlight. Selected = True
    End If
End Sub
Private Sub ListView1_MouseMove(Button _
    As Integer, Shift As Integer, x As Single, y As Single)
    Dim AnyItem As ListItem
    'HitTest returns a node object under the cursor
    Set AnyItem = ListView1.HitTest(x, y)
    If Not AnyItem Is Nothing Then
        Set ListView1.DropHighlight = AnyItem
        ListView1.DropHighlight. Selected = True
    End If
End Sub
```

—Arnel J. Domingo, Hong Kong

VB5, VB6
Level: Beginning

HUNT FOR DEVELOPERS

Want to see a list of the developers who worked on VB5 and VB6? Try this: From VB's View menu, select Toolbars, then Customize…. In the resulting dialog, click on the Commands tab. In the Categories list, select Help. Select "About Microsoft Visual Basic" in the Commands list, and drag it to any menu or toolbar. Right-click on the item you just dragged and rename it to "Show VB Credits" (without the quotes). Then close the "Customize" dialog and click on the "Show VB Credits" item.

—Phil Weber, Tigard, Oregon

VB6
Level: Intermediate

MAXIMIZE VB'S MDI MEMORY

If you use the VB development environment in MDI mode (the default) and you like your code windows maximized, you might have noticed that, unlike VB5, VB6 doesn't remember this preference between sessions. To jog its memory, create a text file containing this code, and name it MAXIMIZE.reg:

```
REGEDIT4
[HKEY_CURRENT_USER\Software\Microsoft\Visual Basic\6.0]
"MdiMaximized"="1"
```

Double-click on the file name to update your system registry. Next time you start VB6, its code windows will maximize automatically.

—Phil Weber, Tigard, Oregon

Visual Studio 6
Level: Beginning

ACCESS HELP MORE EASILY

Several developers have complained that in order to use Visual Studio 6.0's online help, they must keep the MSDN Library CD in their CD-ROM drive, or copy all 680 MB to their hard disk. Fortunately, this is not the case.

When you install Visual Studio's MSDN Library, choose the "Custom…" option. You can select which items, if any, you want to install to your hard disk (VB6's help requires about 12 MB). You can access the help topics on your hard disk without using the MSDN CD; you can still access from the CD any topics you choose not to install.

—Phil Weber, Tigard, Oregon

Visual Studio 6
Level: Beginning

CUSTOMIZE HELP TOPICS

VB developers might wonder why, when they perform a search in Visual Studio 6.0's online help, they get help topics from other Visual Studio languages such as Visual C++ and Visual FoxPro. If you're interested in only VB help topics, try this: In the Active Subset combo box (located above the navigation tabs—Contents, Index, and so on—in the left-hand pane), select Visual Basic Documentation. If the navigation tabs are not visible, click on the Show button in the toolbar, or select Navigation Tabs from the View menu.

If you want to see partial or multiple subsets—say, VB docs and Microsoft Knowledge Base articles—you can define your own: Select Define Subset… from the View menu.

—Phil Weber, Tigard, Oregon

10 HOT VB WEB SITES

(besides DevX, of course <g>)

1. ADVANCED VISUAL BASIC

http://vb.duke.net
This tightly focused Visual Basic site features a collection of articles, a discussion board, and an interface to WinError, a service for interpreting Windows error numbers.

2. JOE GARRICK'S WORLD OF VISUAL BASIC

http://www.citilink.com/~jgarrick/vbasic
This site is loaded with useful information, reusable code, tips and tricks, a Q&A section, and a search engine. The site specializes in database applications and offers several articles on database programming and security.

3. VISUAL BASIC ONLINE

http://www.vbonline.com
This e-zine features tips and tricks, product reviews, and more. You can purchase component software from the online catalog.

4. VISUALBASICSOURCE

http://www.kather.net/VisualBasicSource
This site features a large quantity of tips and code snippets, although the organization and architecture of the site could use some improvement.

5. CARL AND GARY'S HOME PAGE

http://www.cgvb.com
Carl and Gary's Home Page, the first VB Web site, has been updated to include a categorized list of links, as well as new sections on ASP development, JavaScript, Microsoft IIS, and more.

6. ACTIVE-X.COM

http://www.active-x.com
Active-X.com offers an extensive array of commercially developed components and links to their manufacturers' Web sites.

7. ONE-STOP SOURCESHOP

http://www.mvps.org/vb
This site features various techniques for manipulating the Windows API from VB, including a really useful one for deciphering error codes.

8. VBNET

http://www.mvps.org/vbnet
This easy-to-use site contains an assortment of about 70 VB code tips, a FAQ listing, articles, VB links, and a few files to download.

9. COOL VISUAL BASIC

http://www.beadsandbaubles.com/coolvb/index3.shtml
This site's value lies in its message boards and help desk. You'll also find product reviews and a list of links to VB-oriented companies and developers.

10. VB HELPER

http://www.vb-helper.com
VB Helper is a useful site for beginning and intermediate VB programmers to browse and learn. It features several how-to articles and a number of example code files available for download.