# An Introduction to Visual Basic for Visual FoxPro Developers

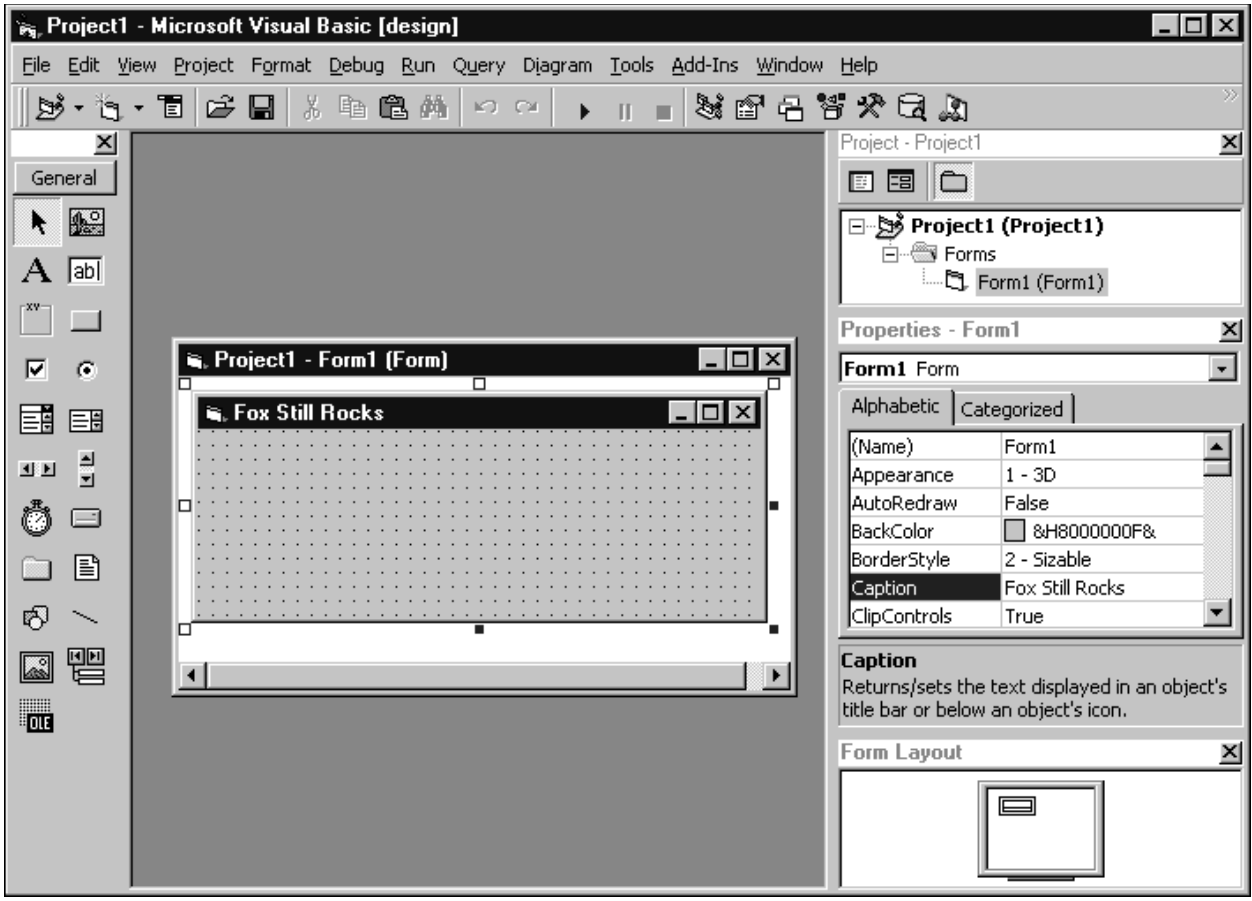
*Whil Hentzen*
*Hentzenwerke Corporation*
*980 East Circle Drive*
*Milwaukee, WI 53217-5361*
*USA*
*Voice: 414.332.9876*
*Fax: 414.332.9463*
*www.hentzenwerke.com*
*whil@hentzenwerke.com*

## Overview

Let's face it. Due to VB's omnipresence, you're going to have to learn it sooner or later. Perhaps you're going to run into sample code that you need to adapt to run in VFP, or maybe you want to build a VB component to use with your VFP application. But it's a waste of time to read through a 900-page manual or sit through a two day class that is aimed at a novice. You already know most of the material. This session distills VB into the essentials you need – and compares it to Fox each step of the way. Covers the IDE, the language constructs, user interface components, building applications and executables, VB's version of object orientation, ActiveX controls, and, of course, data.

# The Visual Basic IDE

Once VB loads, you'll see an MDI window filled with windows, much like Visual InterDev. This may startle some of you who have seen previous versions of VB - those used an SDI interface which was rather disconcerting to us Fox developers - the windows just sat on the desktop and you could see the desktop showing through the holes. (You can turn SDI on if you like through the Tools, Options, Advanced tab.)




Toolbox – One central location for all controls. Can be divided into multiple tabs.

Project Window – A single Explorer-style interface for all project components.

Properties Window – Similar to VFP's properties window, except that (1) doesn't include method tab, and (2) allows sorting either alphabetically or by category

Form Layout Window – Birds Eye view of where the form will display on the desktop.

Immediate Window – (Not shown above) – similar to a Command Window except only used to display results of functions and commands. Is not used to execute commands.

# Variables, Data Types, Arrays, Operators

## Data Types and Variables

```
Boolean Logical
Currency       Floating point number with four decimal places
Date    Date and/or Time
Double  Large floating point (decimal) number (in the gazillions)
Integer Small integer
Long    Large integer
Single  Small floating point (decimal) number (in the billions)
String  We know these as character strings
```

Dim <name> creates variants. Flexible but slow and dangerous.

Dim <name> as <type> creates strongly typed variables.

Option Explicit to force variable types to be declared. (Can also set this option in Tools, Options dialog.)

```
Dim bIsAlive as Boolean
Dim cNetPay as Currency
Dim dBirth as Date
Dim fNationalDebt as Double
Dim iCounter as Integer
Dim lWorldCitizens as Long
Dim fTaxRate as Single
Dim sName as String
Dim sName as String * 32
```

```
Fox            VB
c Character    s String
l Logical      b Boolean
y Currency     c Currency
d Date         d Date
t DateTime
n Numeric      f Double
               l Long
               f Single
i Integer      i Integer
m Memo
g General
```

### *Dates*

VB dates are always stored as m/d/yyyy but Windows controls how they are presented to the world.

```
d1 = #1/2/98#
? d1 + 5
1/7/98
' change the Short Date Style in the Date tab of Regional Settings to m/d/yyyy
d1 = #1/2/2098#
? d1 - 4
12/29/2097
```

### *Times*

Times work similarly to dates, in that the pound signs are the delimiters. For example, to assign the time ten past noon to a variable, you would use the following code:

```
dLunchTime = #12:10:17 PM#
```

Doing arithmetic with time is a little trickier. For example, to add 10 minutes and 13 seconds to a time variable, you'd use an expression like so:

```
? dlunchtime + #0:10:13#
12:22:30 PM
```

#### *Logicals*

```
lIsAlive = True
lIsTalented = False
```

Note that these are not character strings!

## Arrays

```
Dim MyArray(10,3) as Integer
Dim MySmallArray(10 to 20, 2 to 4) as Integer

Dim MyOtherArray(3) as String
MyOtherArray(0) = "Al"
MyOtherArray(1) = "Bob"
MyOtherArray(2) = "Connie"

ReDim MyOtherArray(5) as String
```

## Operators

### Arithmetic (in order of precedence)

```
^       Exponentation
-       Negation
*, /    Multiplacation, Division
\       Integral Division
Mod     Modulus arithmetic
+, -    Addition, Subtraction
&       String concatenation
```

### Comparison (all have equal precedence)

```
=       Equality
<>      Inequality
<       Less than
>       Greater than
<=      Less than or equal to
>=      Greater than or equal to
```

### Logical (in order of precedence)

```
Not     Flips logical value
And     And operations
Or       Or operations
Xor     Exclusive Or
Eqv     Compares two logical values
Imp     Logical implication
```

VarType(varArgument) returns a value according to what you feed it, as shown in the following table.

| VarType Value | Intrinsic Constant | Description |
|---|---|---|
| 0 | vbEmpty | There is nothing in the variant |
| 1 | vbNull | There is not any valid data in the variant |
| 2 | vbInteger | Contains a standard integer |
| 3 | vbLong | Contains a long integer |
| 4 | vbSingle | Contains a single precision floating point number |
| 5 | vbDouble | Contains a double precision floating point number |
| 6 | vbCurrency | Contains currency |
| 7 | vbDate | Contains date or time |
| 8 | vbString | Contains a string |
| 9 | vbObject | Contains an object |
| 10 | vbError | Contains an error object |
| 11 | vbBoolean | Contains a Boolean value |
| 12 | vbVariant | Contains an array of variants |
| 13 | vbDataObject | Contains a data access object |
| 14 | vbDecimal | Contains a decimal value |
| 17 | vbByte | Contains a byte value |
| 36 | vbUserDefinedType | Contains a user defined type |
| 8192 | vbArray | Contains an array of values |

You can use the intrinsic constants in place of the numeric values to make it easier to read your code. The two following lines are identical.

```
If vartype(varMySchizoVariable) = 8 Then
If vartype(varMySchizoVariable) = vbString Then
```

The following table lists a number of conversion functions and what they do.

| | |
|---|---|
| Cbool(argument) | Converts an argument to a Boolean. Best used to convert a numeric value to True or False, such as non-zero to True and zero to False. |
| Cbyte(argument) | Converts an argument to a number between 0 and 255 if possible; else, converts to 0. |
| Ccur(argument) | Converts an argument to a currency value if possible, using the Regional Settings to determine the number of decimals in the currency value. |
| Cdate(argument) | Converts an argument to a date. |
| Cdbl(argument) | Converts an argument to a double precision number. |
| Cdec(argument) | Converts an argument to a decimal. |
| Cint(argument) | Converts an argument to an integer (truncating, not rounding the decimal.) |
| Clong(argument) | Converts an argument to a long integer (truncating, not rounding the decimal.) |
| Csng(argument) | Converts an argument to a single precision number. |
| Cstr(argument) | Converts an argument to a string. |
| Cvar(argument) | Converts an argument to a variant. |

# Program Construction, Flow and Scoping

Right-click on the Project in the Project Explorer to bring forward the Project Properties dialog. Define the “main” program using the “Startup Object” drop down. This can be a program or a form.

To write a program, right-click in the Project Explorer and select Add, Module.

## The Code Window

The code window can display a single method or pieces from several methods. As you can image, this is pretty handy - being able to see code for more than one procedure in the same window (as long as those procedures are small.)

The two buttons to the left of the horizontal scroll bar in the bottom of the code window allow you to toggle between a single procedure and multiple procs. The right button indicates that you want to see more than one procedure at a time. This button is called the "Method View" button. The left button, Procedure View, limits the display to one procedure at a time.

### Defining a Procedure

You'll also see that code for a particular procedure is bracketed by "Private Sub <name>" and "End Sub" statements, just like we use FUNC and ENDFUNC in Fox. "Sub" is a holdover from the olden days when programmers used to write "subroutines", and VB puts them in for you automatically. If you delete one of those lines, "unexpected results may occur."

### Code Completion

You'll notice that VB has "Code Completion" - a ToolTip will display prompting you regarding the parameters that can be used with the function you're typing.

### Subroutines and Functions

Subroutines execute a process. Functions return a value. They both can take parameters. Parameters can be declared when defined in the Sub or Function.

```
Private Sub ProcessMyFiles1 (Optional sDirectory as String)
‘ code goes here
End Sub

Private Function ProcessMyFiles2(Optional sDirectory as String)
‘ code goes here
ProcessMyFiles2 = iHowManyWereDone
End Function

Call ProcessMyFiles1(“c:\”)
```

Or

```
ProcessMyFiles1 "c:\"

iHowMany = ProcessMyFiles2("d:\")

Private Function ProcessMyFiles3(ParamArray sFileNames())
' code to process each element in sFileNames
End Function

iHowMany = ProcessMyFiles3("A.TXT", "B.TXT", "C.DOC")

' in a standard module:
ProcessMyFiles4

' if "ProcessMyFiles5" is in more than one module
ModuleHerman.ProcessMyFiles5

' in another form
FormTheOtherOne.ProcessMyFiles6
```

## Scoping

Three levels of scoping in VB: global, form/module, procedure.

Public: Declares global.

Private (in Form/Module declarations section): Declares private to form/module.

Static: Will preserve a variable between calls to the function.

```
' iCounter gets reset each time X is called
Function X(iValue as Integer)
 Dim iCounter as Integer
End Function

' iCounter keeps its value between calls to X
Function X(iValue as Integer)
 Static iCounter as Integer
End Function
```

Constants - Unchanging values in a program to avoid "magic numbers."

```
Const nPi as Double = 3
```

Can be Public (all procedures in all modules – not allowed in procedures)
Can be Private (all objects in a module – not allowed in procedures)

## Project Components and File Extensions

The project is stored with a "VBP" extension, and is simply a text file. A VB form is also just a text file, with an extension of ".FRM." A module has an extension of .BAS, and contains code that isn't tied to a form - much like a separate FoxPro procedure file.

# Language Constructs

### Logic Structures

You can do the same things with Visual Basic as you can with VFP in terms of controlling flow within a program module. The syntax is just a little different.

#### *Making a decision*

The statement for making a decision is, as it is with VFP, IF. However, VB's IF is much more flexible, as shown in this code fragment:

```
IF <condition1> THEN
<stuff to do>
ELSEIF <condition2> THEN
<stuff to do>
ELSEIF <condition3> THEN
<stuff to do>
ELSE
<stuff to do>
END IF
```

You'll notice that a "THEN" keyword is required after the IF <condition> expression, and the closing expression is two words, not just one as in VFP. The coolest thing about this is that you can nest multiple IF conditions using the ELSEIF <condition> expression instead of having to build multiple IF/ELSE/ENDIF constructs as you do in VFP. Also take note that each condition can be different!

#### *Choosing between multiple choices*

The SELECT CASE construct is VB's version of DO CASE in VFP. It, too, is a bit more powerful, as shown in the following code fragment:

```
SELECT CASE sFileExtension
 <stuff to do>
CASE "BAT"
<stuff to do>
CASE "SYS"
<stuff to do>
CASE "DBF", "CDX", "FPT"
<stuff to do>
CASE "MAA" to "MZZ"
<stuff to do>
CASE ELSE
<stuff to do>
END SELECT
```

This construct takes as input a string expression, sFileExtension, that contains the extension of a file, and processes it according to what extension it is. Thus, .BAT files get one treatment, .SYS files get a different treatment, and so on. This is different from VFP in that you can use multiple, inconsistent conditional expressions following each CASE statement in VFP, and, thus, you have to use the entire conditional expression Note that you can include multiple values following VB's CASE statement, as in the third CASE statement. The code following that piece will be executed for DBF, CDX and FPT files. In VFP, you'd have to write an expression like so:

```
CASE inlist(cFileExtension, "DBF", "CDX", "FPT")
```

Note that you can also have VB span a series of values, as in the fourth CASE where every extension between MAA and MZZ will be treated the same. CASE ELSE works the same as OTEHRWISE in VFP.

### *Looping*

Somebody in Redmond went bonkers when it came time to assemble looping constructs in VB, because there are five of them. I'll describe the first four and then explain why you shouldn't use the fifth.

The FOR NEXT construct is the basic looping construct, as shown in this code fragment:

```
FOR nIndex = 1 to 10 STEP 2
  <some code to run>
  IF <some bad condition occurred> THEN
    EXIT FOR
  END IF
  <more code could be here>
NEXT
```

You'll see that NEXT is used in place of END FOR in VFP, although many of you are probably using the (sort of new) NEXT keyword in VFP instead of END FOR. (If you're not, you should, because it will cut down on the confusion generated when switching between languages.) You'll also see that the escape route out of a FOR loop is EXIT. Big surprise.

If your index expression is an element in a collection of some sort, you can use FOR EACH much easier.

```
FOR EACH <element> IN <group>
  <some code to run>
  IF <some condition found the right element> THEN
    EXIT FOR
  END IF
  <more code could be here>
NEXT
```

So far, so good. The next two logic structures are pretty similar: DO WHILE and DO UNTIL. In fact, you can probably noodle them out yourself, right? Not so fast!

```
DO WHILE <condition>
  <some code to run>
LOOP

DO UNTIL <condition>
  <some code to run>
LOOP
```

The trick to both of these is that while the above syntax looks comfortable to you, you're most likely not going to see it in experienced VB'ers code. That's because you can also put the WHILE and UNTIL expressions after the LOOP keyword, like so:

```
DO
  <some code to run>
```

```
LOOP WHILE <condition>

DO
  <some code to run>
LOOP UNTIL <condition>
```

Obviously, doing so changes how the loop is processed – much like incrementing a counter at the beginning or the end of a loop. You can use the EXIT DO statement to get out of DO JAIL early, like so:

```
DO WHILE <condition>
  <some code to run>
  IF <some bad condition occurred> THEN
    EXIT DO
  END IF
LOOP
```

The WHILE/WEND construct is a cheap, poorly imitated version of DO WHILE, because you can't use EXIT DO to escape early. If that's not important to you, then you can use WHILE/WEND instead. Keep in mind, however, that if your requirements change and you have to be able to terminate processing early, you'll have to recode your WHILE/WEND to DO WHILEs. Why not save yourself the inevitable hassle now?

Instead of trying to describe, in detail, each VB statement, I've just listed the rest of them, so that you can get a quick idea of what functionality is available to you natively, and, thus, what you'll have to write yourself or live without.

### File Handling

| | |
|---|---|
| Close | Concludes IO to a file opened using OPEN |
| Get | Reads data from an open disk file into a variable |
| Input # | Reads data from an open sequential file and assigns the data to variables |
| Let | Assigns the value of an expression to a variable or property |
| Line Input # | Reads a single line from an open sequential file and assigns it to a string variable |
| Lock, Unlock | Controls access by other processes to all or part of a file opened using OPEN |
| Lset | Left aligns a string with a string variable, copies a variable of one user-defined type to another variable of a different user-defined type |
| Mid | Replaces a specified number of characters in a variant (string) variable with characters from another string |
| Open | Enables IO to a file |
| Print # | Writes display-formatted data to a sequential file |
| Put | Writes data from a variable to a disk file |
| Reset | Closes all disk files opened using OPEN |
| Rset | Right aligns a string within a string variable |
| Seek | Sets the position for the next read/write operation within a file opened using OPEN |
| Width # | Assigns an output line width to a file opened using OPEN |
| Write # | Writes data to a sequential file |

### Program Control

| | |
|---|---|
| Call | Transfers control to a Sub procedure, Function procedure or a DLL |
| Function | Declares the name, arguments and code that form the body of a function procedure |
| GoSub | Branches to and returns from a subroutine within a single procedure |

| | |
|---|---|
| GoTo | Branches unconditionally to a line within a procedure |
| On GoSub, On GoTo | Branches to a line depending on the value of an expression |
| Stop | Suspends execution of a program |
| Sub | Declares the name, arguments, and code that form the body of a sub procedure |
| With | Executes a series of statements on a single object or user-defined type |

### File System

| | |
|---|---|
| ChDir | Changes the current directory or folder |
| ChDrive | Changes the current drive |
| FileCopy | Copies a file |
| Kill | Deletes files from a disk |
| MkDir | Creates a new directory |
| Name | Renames a directory or file |
| RmDir | Deletes a directory |
| SavePicture | Saves a graphic from the picture or image property of a control to a file |
| SetAttr | Sets attributes information for a file |

### Variables

| | |
|---|---|
| Const | Declares constants for use in place of literal values |
| Declare | Declares references to external procedures in a DLL – used at module level |
| Deftype | Set default data type for variables, arguments passed to procedures, and return type for Functiona nd Property Get procedures – used at module level |
| Dim | Declares variables and allocates storage space |
| Enum | Declares a type for an enumeration |
| Erase | Reinitializes the elements of a fixed size array and releases dynamic array storage space |
| Option Base | Declares the default lower bound for array subscripts – used at the module level |
| Option Compare | Declare the default comparison method to use when string data is compared – used at the module level |
| Option Explicit | Forces explicit declaration of all variables in that module – used at the module level |
| Option Private | Prevents a module's contents from being referenced outside its project |
| Private | Declares private variables and allocates storage space – used at the module level |
| Property Get | Declares the name, argument and code that form the body of a property procedure which gets the value of a property |
| Property Let | Declares the name, argument and code that form the body of a property procedure which assigns the value to a property |
| Property Set | Declares the name, argument and code that form the body of a property procedure which sets a reference to an object |
| Public | Declares public variables and allocates storage space – used at the module level |
| Randomize | Initializes the random-number generator |
| ReDim | Reallocates storage space for dynamic array variables |
| Static | Declares variables and allocates storage space – used at the procedure level |
| Type | Defines a user-defined data type containing one or more elements – used at the module level |

### Registry

| | |
|---|---|
| DeleteSetting | Deletes a section or key setting from an application's entry in the Registry |
| SaveSetting | Saves or creates an application entry in the application's entry in the Registry |

### Classes

| | |
|---|---|
| Implements | Specifies an interface or class that will be implemented in the class module in which it appears |
| Load | Loads a form or control into memory |

| | |
|---|---|
| Set | Assigns an object reference to a variable or property |
| Unload | Unloads a form or control from memory |

**N.E.C.**

| | |
|---|---|
| Date | Sets the system date |
| Error | Simulates an error |
| Event | Declares a user-defined event |
| On Error | Enables an error-handling routine and specifies the location of the routine within a procedure |
| Resume | Resumes execution after an error-handling routine is finished |
| Rem | Identifies a comment |
| RaiseEvent | Fires an event |
| SendKeys | Sends one or more keystrokes to the active window as if typed at the keyboard |
| Time | Sets the system time |

# User Interface Components

Earlier I mentioned that there is no hierarchical display of objects in the Properties Window object box. This is because VB doesn't have the same type of containership as there is in VFP - and while confusing, this actually makes life easier in same cases. For instance, if you put a check box on page two of a page frame in VFP, you use the following code to reference it's caption:

```
thisform.pageframe1.page2.checkbox.caption
```

In order to reference the same check box's caption in VB, you simply use the code

```
check2.caption
```

This tells us a couple things. First of all, you will wear down your keyboard less, since you have to do much less typing. Second, every control on a form has to have it's own unique name - regardless of where it lives. Thus, while you can have a "checkbox1" on each page of a four-page tabbed dialog, you'd have "check1", "check2", "check3" and "check4" in VB. Third, the containership, or lack thereof, is going to have some serious ramifications on programming that we'll see later.

## Adding Code to a form

Code windows are one significant departure in behavior from "the Fox way." First, in order to get to them, you can't use the Properties window. It's for *properties*, after all. It doesn't say "Properties and Code Window," does it?

To open up the code window, you again have nine thousand ways to doing so. You can highlight an object in the Project Explorer and click on the View Code button in the project

You can also issue the View, Code menu command, but the easiest way is to just double-click on the form. The left drop down shows the various objects of the form, including "General", "Form", and every control on the form. The right drop down contains all of the events into which you can stuff code - and you're familiar with many of these, such as Activate, Load, Resize, and Unload.

As I just mentioned, when you write code, you don't have to use a fully qualified name for a control - simply the name of the control and it's property or method name. Second, in order to get to the code, you can either press the View Code button or just double click on the form. But here's where it can get confusing. Remember that you can use the icon in the bottom left of the window to display all the code for the form, or just the code for the current object (either the form, or a control on the form). When you double-click on a control that has no code attached to it, you'll get a new module for the click event of that control already built for you.

Otherwise, double-clicking on a control with code in one of it's events will bring up the code for that event.

The code window can contain code for one or more events, for one or more controls. Each "chunk of code" attached to an event has two parts. The first is the declaration of the module, and the second is the termination of the module, like so:

```
Private Sub Form_load()
<your code goes here>
End Sub
```

Each of these "chunks of code" is referred to as a subprocedure, or a function, or, generically, a code segment. The "Sub" keyword means that this is a subprocedure - much like procedures in Fox. The "Private" keyword means that this subprocedure scope is the form - it can be called by code elsewhere in this form, but not anywhere else in the project.

The subprocedure's name consists of the name of the control and the event to which the module is attached. Remember last month's warning - putting code in a module and then changing the controls' name will reflect in the module being orphaned.

The code window also has a section called "General" (it's the first entry in the object drop down at the top of the code window.) You can use this area to create your own subprocedures and functions, much like form-level methods in Fox.

The parens following the name of the control enables you to pass parameters to the code segment. For example, suppose you have a form with a check box and a command button on it. You could use the following code to pass the caption of the control to a form-level function that would display the caption whenever the control was clicked.

```
Private Sub MySub(xmessage)
 MsgBox (xmessage)
End Sub

Private Sub Check1_Click()
MySub (Check1.Caption)
End Sub

Private Sub Command1_Click()
MySub (Command1.Caption)
End Sub
```

In this brief example, you see that the same basic <groan> programming principles apply - calling a subroutine with a parameter. And since MySub is declared as private, you can't go off and call that very valuable subprocedure from another component in the project.

# Controls

Visual Basic is considerably different from VFP in that hardly any VB developers just accept the controls that come "in the box" as all they'll use. The whole idea behind VB was to allow the developer to extend the environment by using additional controls as they desired – truly, very few VB developers toolboxes look alike.

However, there are twenty controls that come with VB and appear automatically on the toolbox's toolbar.

## Manipulating controls

Silly enough, it took me the better part of a day to get the hang of putting controls on a VB form. You don't click on the control in the toolbox and then on the form, and expect the control to be drawn for you like in VFP. Instead, either click on the control, and then click and drag the control on the form – just clicking on the form doesn't do anything. Alternatively, you can just double-click on the control in the toolbox – the control will be placed in the center of the form. If you double-click on a control in the toolbox multiple times, you'll end up with several controls on top of each other in the center of your form.

Moving a control is a little different – the cursor keys don't nudge a selected control or group of controls bit by bit. Instead you have to hold the Ctrl key down in order to nudge them one way or another. You'll also want to take note of the Format, Lock Controls menu option – selecting it will turn all of the sizing handles to white, and you'll not be allowed to use the mouse to move or resize the controls from then on. Evidently accidental moves by novice VBers was a problem at some point in time. You can, however, still use the keyboard to move and resize as you want.

Alignment is one of those "half-full, half-empty" issues. You can use the Tools, Options command to display or hide the grid on a form, define the granularity of grid lines, and to have controls snapped to grid lines or not. You can also use the Format, Align menu command to align a group of selected controls, much as you do in VFP. And if you're like me, you never use the menu command in VFP, preferring to use the Layout toolbar. Unfortunately, if you look for a Layout or an Alignment toolbar in VB, you won't find one. Instead, select the Form Editor toolbar. It looks sparse, but you'll see that each button opens up into a bunch of related choices, which I think is pretty cool. It just took a while to get used to it – it's still two mouse clicks instead of one.

## Labels

The Label control is similar to VFP's label, and has many of the same properties.

## Text and Edit Boxes

The TextBox control is similar to it's VFP counterpart, but also does double duty to serve as VFP's editbox match. The Value property in VFP is called the Text property in VB for this control. You can set the MultiLine property to True and then enter more than one line of text – where you'd use an EditBox in VFP to do the same thing. If you have MultiLine set to True, you can use the ScrollBars property to display them as desired.

## Checkboxes

The CheckBox control is similar to VFP's CheckBox. It can contain a value of 0 (not checked), 1 (checked), or 2 (disabled). You have to be careful about using a value of 2, because once a checkbox's value has been set to 2, you can't tell whether it's checked or unchecked. 2 (Disabled) is also different from setting the Enabled property to False; the former keeps the checkbox's caption enabled (black) while the latter also dims the caption, turning to a fuzzy grey/white combination.

## Command Buttons

The CommandButton is VB's answer to VFP's command button – and a button is a button is a button.

## Option Buttons

The OptionButton sort of maps to an option button in VFP, but there are a couple of significant differences. If you've already placed an option button on a VB form, you may be wondering, "Where's the Option Group control?" And the answer is… There isn't any! All of the option buttons you place on a form – regardless of *where* on the form – are part of the same option group. Blech, you say? Me too. It is a bit easier, of course, to be creative with placement, of course – don't you hate having to enlarge an option group, and then edit the group, and then align the buttons where you want them? Kind of a pain. In VB, this is easier.

But the flip side is that if you want more than one option group on a form, you have to build your own container first, using a Frame control, and then add individual option buttons. Like I said, "Blech."

A group of option buttons (I didn't say "An Option Button Group", did I?) works just like you would expect – selecting one deselects another that had been selected. You need to write code to deselect all of them, like the following fragment (you could put this in the click() method of a command button, for example):

```
Option1.Value = 0
Option2.Value = 0
Option3.Value = 0
```

## Combos and Lists

The ComboBox and ListBox controls are similar to VFP's combo box and list box controls, and I think they're actually somewhat easier to use "out of the box."

The ComboBox control is similar to VFP's combo box, complete with multiple modes and a variety of properties. You can set the Style property to one of three possibilities. 0 (DropDown Combo) means you can enter text or click on the arrow to open the combo. 1 (Simple Combo) means the combo box is always open but you can also enter text if you want. It's a little unusual – you need to resize the combo box to display more than one item – and the result ends up looking like a text box with a list box placed below it on the form. 2 (DropDown List) means that you can't enter text, and you have to open the control in order to select a value.

The easiest way to add items to a ComboBox or ListBox is with the AddItem method, like so:

```
List1.AddItem "A"
List1.AddItem "B"
List1.AddItem "cccccc"
```

You can make a ListBox multi-selectable by setting the Mult-Select property to 1 (Simple) or 2 (Extended). Simple means that you can just keep clicking on items and they stay selected (clicking a second time deselects the item.) Extended means you can use the Ctrl and Shift keys to select a range of items – such as the 3rd, 4th, 5th, 9th, and 12th.

You can determine which item has been selected in a ListBox with the Text property (just as you would use the DisplayValue property in VFP.) If you have Multi-Select set to 1 or 2, the Text property displays the last item selected. In order to determine all those items selected, use the Selected property together with the ListCount property, much as you would in VFP:

```
Dim nItemNumber As Integer
nItemNumber = 0
Do While nItemNumber < List1.ListCount
  If List1.Selected(nItemNumber) = True Then
    MsgBox "Hey, I'm selected " & nItemNumber + 1
  End If
  nItemNumber = nItemNumber + 1
Loop
```

Note that the index of the array that populates the ListBox starts with zero here, not 1 as in VFP.

## Other Controls

The Shape and Line controls allow you to create a variety of lines, circles and rectangles on your forms, much like the same controls in VFP do. The Shape property of the Shape control allows to you to choose between Rectangle, Square, Oval, Circle, Rounded Rectangle, and Rounded Square.

The Image and Picture controls allow you to place graphic files on your form. The Image control is lightweight, but doesn't allow you to overlap controls (and thus, images), and can't receive focus. The Picture control, on the other hand, can be overlapped and can receive focus – which makes it a good candidate to use when creating graphical controls.

The DirListBox control, in combination with the DriveListBox and the FileListBox controls, allow you to build file navigation dialogs quickly and easily.

The Frame control is used as a container, and I'll cover it, together with the Horizontal and Vertical Scroll Bars, the Timer, the OLE control, and the Data control, in another article.

# Building Applications and Executables

### Running and stopping a form

Press F5 or the right-point arrow in the Standard toolbar. The project will execute, running whatever is defined in its hierarchy.

Press Alt-F4 or the square box in the Standard toolbar to stop a running program.

### Building an EXE

Once you've put together the components you want in your project, you can build a simple EXE through the File|Make EXE menu command. You can set various options for this operation in the Project Properties (right-click in the Project Explorer.) These options include versioning, ActiveX properties (such as licensing), version information, Icons, command line and compilation arguments, optimization choices, and so on.

Depending on what type of project you selected when you first created it, some options may or may not be available.

### Types of Projects

When you start a new project, you have 13 choices of what type to build and how to proceed:

Standard EXE
ActiveX EXE
ActiveX DLL
ActiveX Control
VB Application Wizard
VB Wizard Manager
ActiveX Document DLL
ActiveX Document EXE
Addin
Data Project
DHTML Application
IIS Application
VB Enterprise Edition Controls

# VB's Version of Object Orientation

Object Oriented languages distinguish themselves from procedural languages by exhibiting three characteristics: Polymorphism, Inheritance, and Encapsulation, or "PIE".

VB 5 and 6 provide the ability to create what they've called "classes", and allow the user to "instantiate" objects from those classes. This capability has excited many a VB programmer, making them think they were in the big leagues now. Unfortunately, inheritance is not included in the mix.

*"The classes we can create in VB support the concepts of polymorphism and encapsulation, but not inheritance. That's just fine. While inheritance can be useful when structuring collections of classes, it can be a major source of bugs and its implementation requires the use of code that is difficult to understand."*

– John Connell, Beginning VB6 Database Programming, WROX Press.

In other words, inheritance is too hard to use correctly, so it's just as well that it's not in VB.

Nonetheless, the ability to create classes – blueprints for objects – is a major step forward for Visual Basic. If you forget for a moment about inheritance, VB's classes will make your life easier.

# Data

Visual Basic is a programming language that has more and more been used as a front end to database applications. As such, it does not contain a native DML (data manipulation language) as we've seen above in the command and function set, nor does it have a native data engine.

Instead, the mechanism you use to connect VB to data is much like a remote view in VFP – you define a connection, generate a cursor (called a recordset) through that connection, map that recordset to controls, and navigate through the recordset from record to record.

There are a large number of backends you can connect to, and a variety of connection mechanisms. They all ultimately result in a recordset of one form or another. If you're just starting with VB, I'd suggest you skip all of the old methods (although they are still popular today) and being using ADO – ActiveX Data Objects. ADO is Microsoft's stated direction for the future, and you might as well start off on the right foot.

## Connections

One of the controls that ships with VB, but is not a part of its intrinsic Toolbox, is the ActiveX Data Control. You need to add it to Toolbox by right-clicking on the Toolbox, selecting the Components menu option, and selecting the Microsoft ADO Data Control 6.0 (OLEDB) item in the Controls tab list box.

"ADODC" will appear in the Toolbox along with all of the other controls that were already there. Double-click on it to place it on a form.

Right-click on the control (once placed in the form) to bring up its Property Pages (not the Property Window). Click on the Connection String option button and then select Build.

The first tab of the Data Link Properties dialog will display a list of OLE DB providers. Select a provider (it will vary according to which data source you want to connect to), and then fill in the rest of the properties as appropriate for the type of data source you're using. If you select Jet 4.0, for example, for an Access database, you'll then be asked the name of the Access MDB file you want to connect to.

Once you've connected to a database, you'll need to specify what the record source is. This object is the source of the cursor you're going to use. There are four choices in the Property Pages RecordSource tab:

**adCmdUnknown**

```
adCmdText
adCmdTable
adCmdStoredProc
```

Depending on which you choose, other attributes are then available. For example, for the adCmdTable choice, you can then select a specific table in the database. For adCmdText, you can enter in your own SQL Select command.

## Recordsets

Depending on which type of connection and RecordSource, you'll now have a recordset attached to your form. You can manipulate this through the use of properties and methods, much like you deal with other objects. For example, you could create a series of command buttons that

- Determine which record in the recordset is the current record
- Move to the first record in the recordset
- Move to the next record (not the last record) in the recordset

When you drop the ADODC control on the form, you'll give it a name, like adodcNewCustomers, because the recordset it represents are new customers in the Customers table. (You've done a filtered query or otherwise narrowed down the number of records here.) You can then refer to the recordset's properties and methods like so:

```
adodcNewCustomers.Recordset.AbsolutePosition
adodcNewCustomers.Recordset.MoveFirst
adodcNewCustomers.Recordset.MoveNext
```

## Binding Controls to "Data"

However, moving back and forth from record to record is only partially interesting. You will want to display data from the recordset in controls on the form.

In order to bind a text box to a field in a recordset, you'd set the text box's DataSource property (in the Property Window) to whichever recordset you were interested in, such as adodcNewCustomers. (A form can have more than one data control, and thus, more than one recordset on it.) Then you'd select the DataField property drop down. It would be filled with all of the possible choices available from the recordset (since a recordset may just have a few fields from the table or database.) This two step process maps the text box to the recordset.

You may also want to populate a combobox, say, from a recordset that was specifically created for that purpose. The following code would do this.

```
Do While True
 ComboDepartmentNames.AddItem AdodcJustNames.Recordset!cDeptName
 AdodcJustNames.Recordset.MoveNext
```

```
 If AdodcJustNames.Recordset.EOF Then
  Exit Do
 End If
Loop
ComboDepartmentNames.AddItem "All"
```

Note that the identifying separator for a field in a recordset is the bang – thus, recordset!cDeptName refers to the cDeptName field in the recordset.

# Conclusion

Summing up a popular programming language that's been through a half-dozen major revisions is impossible in 75 minutes. There's a great deal that we haven't been able to address in any detail, and even more that wasn't mentioned at all.

Nonetheless, given your experience with Visual FoxPro, we've been able to point out some of the major features in Visual Basic 6.0 as they relate to database programmers, and what some significant differences between VB 6.0 and VFP 6.0 are.

<end>