

Creative's

VISUAL BASIC



Digital India
Power To Empower



Skill India
कौशल भारत - कुशल भारत



INSTITUTE OF INDUSTRIAL AND INFORMATION TECHNOLOGY (IIIT)
SILCHAR-1, PHONE: 03842-262588, Visit us at: www.iiitsilchar.in

Overview	2
IDE.....	4
Projects.....	9
Forms	12
Controls Overview	15
Intrinsic Controls.....	18
ActiveX Controls	21
Code.....	25
Command Descriptions.....	29
Command Summary	46
File Handling.....	48
Printing.....	49
Distribution of Applications.....	52
Menus.....	53
Common Dialogs	57
Using the Mouse.....	60
Drag & Drop.....	62
Graphics Commands.....	64
Error Handling.....	67
Databases	71
Database Controls.....	71
SQL.....	76
Code Librarian.....	78
API.....	87
Objects	98

Overview

It can be very difficult to pick up a new skill, especially when the training material starts off assuming you know more than you actually do! Throughout my tutorials I will continually try to put things into perspective and to explain why a certain capability of VB is valuable and whether it might be valuable to you. This means that my introductions to each of the sections will be a bit long, but the intent is to set the stage so that you will know why you need to know the information in that section. To begin, let's talk about VB and how it was derived from its predecessor, BASIC.

A Definition

Visual Basic is a much-enhanced version of the BASIC programming language and the BASIC Integrated Development Environment (IDE). The bottom line of the enhancement is the VB can create Windows programs whereas BASIC could only create DOS programs. Ok, so the modifications are **very** major, but the idea holds true that Visual Basic is BASIC for Windows.

One of the many significant improvements is that VB provides massive support for easily creating the user interface to your applications. This is accomplished within the VB Integrated Development Environment (IDE), in which you use a mouse to "draw" your application and use the keyboard to type in the code that is to be executed. When I write a VB program, I almost always create the user interface "shell" before I write any code at all. This approach, made so simple by the VB IDE, allows me to evaluate how the user will interact with the program. I can generally create the initial version of the shell in just hours. It's a much better way to program than to create pieces of the interface (and the corresponding code) as you go. It especially is beneficial in that you can demonstrate to the users just exactly what they will see, and you can do it early enough in the development cycle to prevent costly reiterations later in the cycle.

The single largest effect on coding that VB introduced was the concept of an event-driven programming model. In the old BASIC you had to write code to watch for the occurrence of user events (pressing a key, using the mouse, ...). VB performs that function for you, and in fact, the **only** time code will execute in VB is in response to such an event!

And finally, the other major concept that VB has incorporated is the concept of **objects**. Objects provides a way to link together both code and data into a "package" in such a way as to make handling and saving the code/data more intuitively. VB forms are objects, menus are objects, and the so are the intrinsic VB controls. A lot more on this later!

VB has also provided a wide variety of built-in code that programmers once had to handle themselves. Of most significance is the built-in database handling features of VB. It is generally accepted that over half of all VB applications are written to handle databases! You'll find the built-in database features of VB to be very powerful, and that you can tap into them at whatever level of programming skill you possess.

A second area in which VB has begun to provide built-in support is that of Internet access. The VB features are still maturing, but with the tools available you can create very useful applications. I've included Internet topics in the Advanced section of the tutorial because it is such a specialty application, not so much because its concepts cannot be handled by less skilled programmers. Feel free to skip around in the tutorial if you find a section that interests you.

Critical Visual Basic Elements

Although Visual Basic has grown into a fairly complex programming tool, it is still the case that a programmer can pretty much ignore all of the capabilities he doesn't need (or understand) and still create very useful applications.

But, no matter how much a beginner, or how advanced you are, there are still some fundamental areas in which you must be proficient to become a VB programmer.

When you start VB, you will see a group of windows that are know as the VB IDE (integrated development environment). As a programmer you will spend the majority of you time here, so you might as well get used to the IDE and spend a fair amount of time exploring the menu options that the IDE provides. Pay closed attention to the keyboard shortcuts that are available. If you've read my other Beginner sections you'll know that I am a **big** fan of the keyboard and have argued that the #1 productivity tool you have is good typing skills. While in the IDE, you'll find the keyboard shortcuts invaluable in writing your program quickly.

The second aspect of VB which programmers of all skill levels will have in common is the VB language itself. Most of the questions I get through email are focussed on "how do I ..." and the answer is almost always couched in terms of the code that it takes to perform the task. If ever there was less boring reading than the VB language reference manual, I don't know what it is, but it also provides one of the biggest payoffs of studying that I can recommend. At least 90% of every question I answer for visitors to my site **is in the manual!**

Every Visual Basic application will consist of controls, usually a lot of them! In my opinion, the availability of controls (built-in, or controls you can purchase) is the single biggest reason why VB has reached the level of popularity that it currently enjoys. Because controls represent hundreds (if not thousands) of hours of manpower to come up with full-featured, debugged code which you can reuse in your program, controls are easily the most cost-effective, and the most time-effective way that a VB programmer has to add features to his program. Bottom line is that any good programmer must be an expert at handling controls. In my experience, once you've mastered the intrinsic controls, learning new controls becomes very straightforward.

Because VB has moved to the event-driven model of programming, the last critical VB topic I will mention is that of events. Events are not very complicated but the concept is significantly different than the old-style linear programming of the original BASIC. Simply put, when a VB program is started it sits and waits for an event to occur. The event can be a keypress by the user or the movement of a mouse. Either way, the VB programming model is that your program will only react to events. When an events occurs, VB will execute the code associated with that event.

So your job as a programmer is to basically create the code which your program executes in response to those events.

Programming Basics

To toss in a bit of philosophy, I view a programmer's job as getting to the end product with a minimum of effort. When you're paid by the hour you have an obligation to your employer (even if it's yourself) to minimize the cost/schedule of completing the assignment. To that end, I will emphasize over and over again the importance of using existing capabilities (such as your own reusable code, VB-provided controls, or even controls that you purchase).

In light of this optimize-your-time philosophy, I also believe that every VB programmer should strive to become an expert in the following two areas: Databases and Reporting. It's very common for applications to store data and to provide that data in a printed format (by some estimates, 80% of all VB applications use databases). And while this tutorial will discuss various ways to perform these tasks, most programmers will find that the capabilities of VB for creating/editing Access (a Microsoft product) databases and VB's capabilities for reporting that data (as exemplified by the Crystal Reports control or the newer built-in VB reporting features) are the most effective tools you can use for increasing your effectiveness in creating VB applications.

If you want to become a serious VB programmer, you **must** become an expert in these areas. I'm not saying that Microsoft's built-in tools are the only, or best, tools available for creating/editing databases and reporting on them. What I am saying is that tools which provide similar functions will be one of the most commonly used tools of those available to a programmer and that any serious programmer must take the time to develop strong skills with these tools.

One by-product programming strategy of my philosophy is that newbies should not get too anxious to jump into the more advanced features of VB. Take the time to learn the basics, and especially how to apply them with ingenuity. Once you've exhausted the potential of the fundamentals is the time to consider more advanced techniques. In my experience, over 90% of my programs consists of fundamental programming techniques, while only rarely am I compelled to dip into the more complex features that VB has to offer.

VB History (VB3, VB4, VB5 and VB6)

Starting with VB5, Visual Basic became an exclusively 32-bit programming language, suitable for programming only Win9X or NT systems. If you must program for Win 3.x, then you'll have to drop back to either VB3 or VB4, both of which are in pretty short supply. VB4 had the dual ability to support Win3.x as well as Win9X/NT systems but my personal recommendation is that if you need 32-bit system support, go straight to VB6 and if 16-bit is your need then stick with VB3.

The VB Learning Edition is the most affordable, and truth is that you can do a lot with it, particularly if you use the Windows API to augment its capabilities. However, in light of its better database features and its greater variety of controls, I suggest you go straight to the Professional Edition if at all possible. The price is steep, but it really does pay itself back in terms of time savings. If you need the VB Enterprise edition then you should have it paid for by the "Enterprise" which requires it. Individual programmers generally do not need the Enterprise edition.

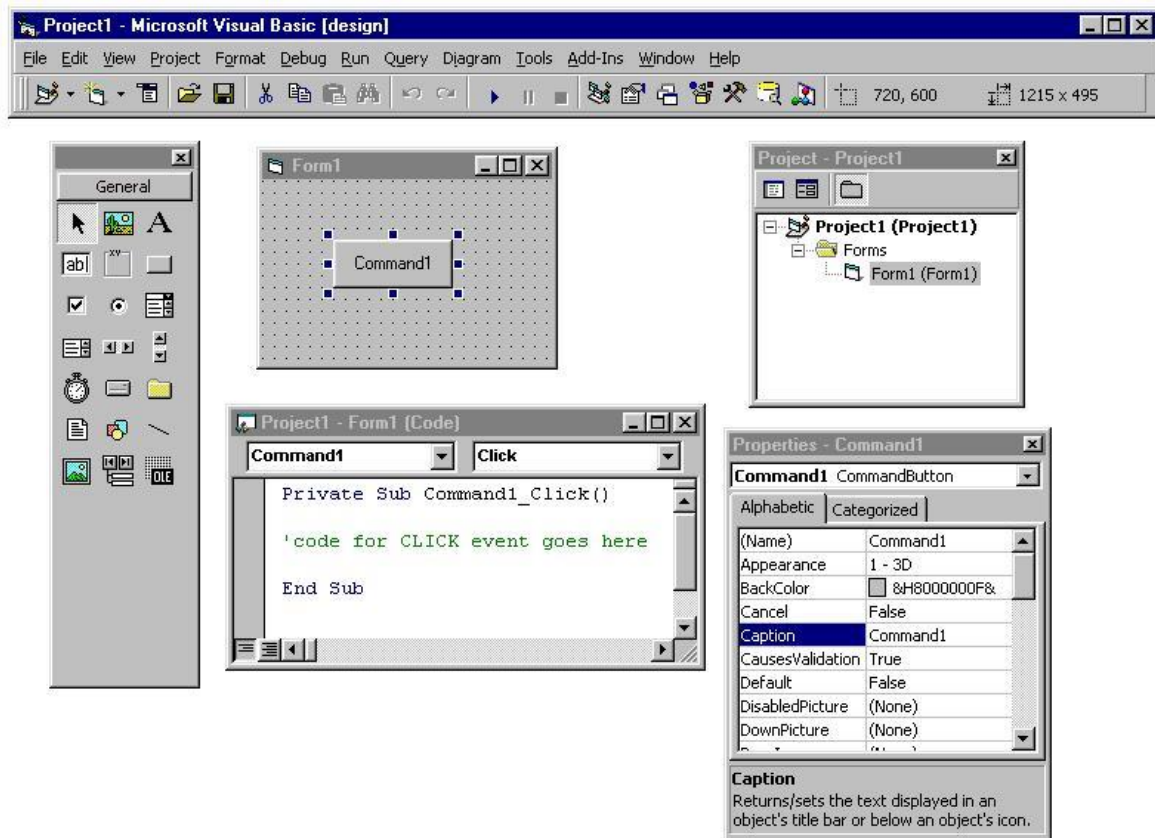
IDE

It's a little known fact, but you can write a VB program from scratch using nothing more than a simple text editor, such as the Edit or Notepad applications which come with Windows. In fact, the Visual Basic project files are exactly that - text files. However, writing a project from scratch would involve a **lot of tedious, detailed** manual entries. To simplify the task, Microsoft has built in to VB a software program to help you write your VB projects. That software, known as the Integrated Development Environment (IDE for short) is what comes to the screen when you start VB and is the topic of this section.

Overview

Like any other Windows application, VB consists of multiple windows which appear at startup. The windows that are displayed when you start VB are collectively known as the Visual Basic Integrated Development Environment (IDE).

When you first start VB all of the windows are locked together in what is called the MDI format. I prefer the SDI format (which you can set in the options menu) which allows each of the windows to be positioned independently on your screen. Here's a sample IDE screen which shows a VB project with one form on which is a single command button.



In particular, VB has the following windows:

- **Menu / Toolbar**

This is the only element of the IDE which is always visible. You use it to select which other IDE elements to view and to add forms or controls to your project. There are many other features which we will discuss later.



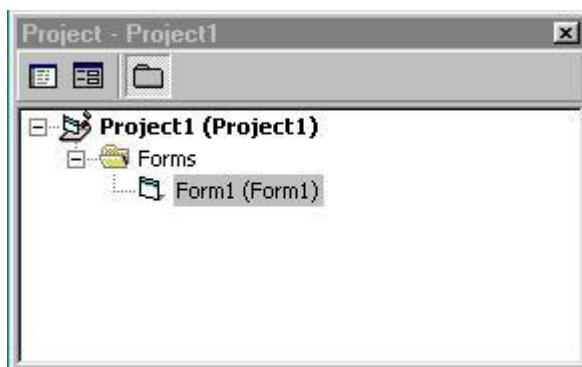
- **Toolbox**

The toolbox is simply a library of controls which you can place on your application. Once you've placed all the controls you need onto your applications forms, you can hide the toolbox to make room for working in the other elements of the IDE.



- **Project Window**

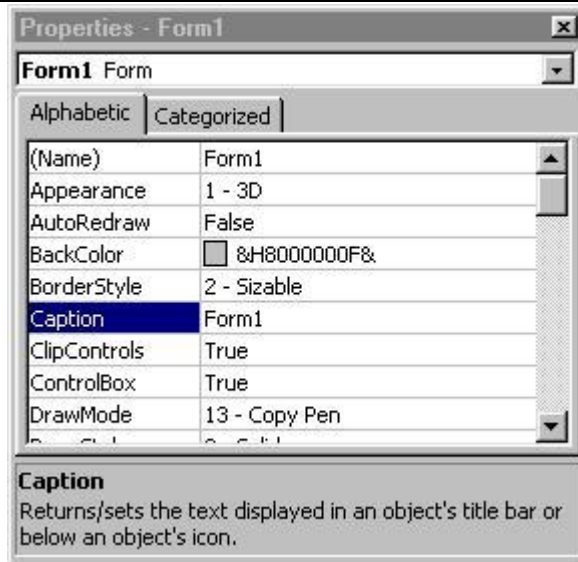
This is simply a list of all the forms which make up your VB project. There are several kinds of forms which we'll talk about later.



- **Property Window**

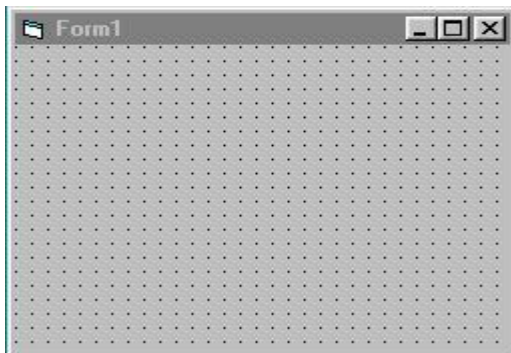
-

We'll talk about controls later, but such things as push-buttons, scrolling text boxes, pictures boxes and other features of most VB applications allow you to enter parameters which define how these controls work. In VB, these parameters are called properties. Some properties can be entered at design time within the IDE, while others must be entered with code while the program is running.



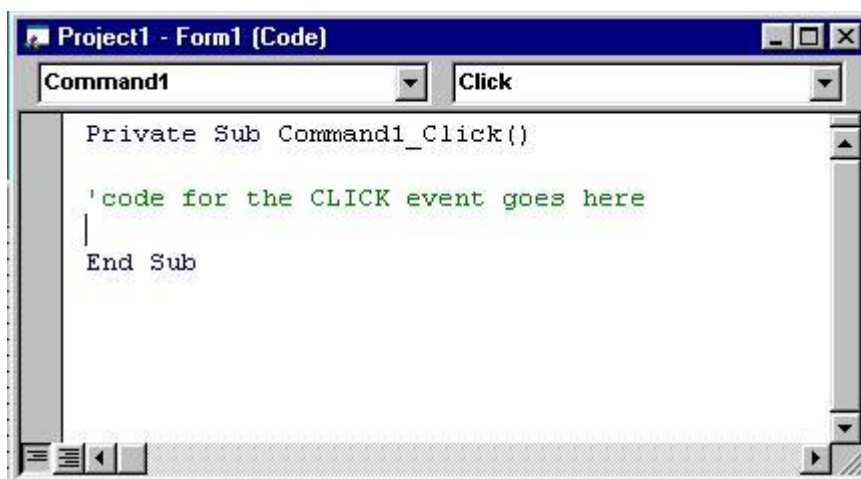
- **Forms**

You add these to your VB application as they are needed. They are the windows which hold the various controls (buttons, text boxes, etc.) which make up your application.



- **Code Window**

Like its name implies, this is where you type in the code that VB executes. Notice that the heading of the window indicates with which event the code is associated.



A VB Session

When VB begins a new project, it starts with a single form which has no controls placed on it. A session in VB would go something like this:

1. Add additional forms (if needed)
2. Set form properties

-
3. Place controls on forms
 4. Set control properties
 5. Write code for form/control event procedures
 6. Create executable (.EXE) or simply RUN program within the IDE
 7. Save project

It's important to understand that each form in VB is saved as a standalone file on your computer. The listing of all forms, and their location on your hard disk, are kept in another file which is called the project file. In VB3 the project file extension was .MAK but in VB6 it is .VBP. This file is just an ASCII text file which provides information about the VB project. There is a variety of other information which is held within the file, but all of it is directed towards describing those files (and controls) which make up the VB project. We'll talk more about the content of the file in other sections of this tutorial. As your skills improve, you'll even feel comfortable about editing the project directly, using a simple text editor such as notepad. I do it occasionally when I want to make a simple change to a program without going through the VB IDE.

Even more useful is that if I want to capture a routine that I know is in a VB project, I can call up the file that has the code and copy it to the clipboard for insertion into my new project. This has the benefit of being very quick and it lets me use search tools that are not a part of the VB IDE.

Your First Application

Ok, it's already time to create your first application. Beginner's are usually shown what's called a "Hello World" application. It's called this because all the application does is display the words "Hello World" when the user presses a button. As useful applications go, it's not much, but it does show what is involved in creating an application.

- **Step 1.**
Start VB. As it VB comes up, it automatically creates a NEW application consisting of only 1 form which has no controls.
- **Step 2.**
Using the mouse, move the cursor over the toolbox and click with the left mouse on a command button control. This selects the button. Now move the cursor to the form and while pressing the left mouse button draw out a rectangular shape with the mouse. Once the shape is drawn and the left mouse is released, a command button appears on the form.
- **Step 3.**
Change the CAPTION property of the command button. Do this by clicking once on the button to select it, then pressing F4 to bring up the Property window. Scroll in the window until you find the CAPTION property. Highlight the CAPTION property and type in "PRINT". The text on the command button will be replaced by what you type.
- **Step 4.**
Remember earlier in the tutorial that we talked about writing code for events? Well, each of VB controls recognize certain events. In this example, the CLICK (left mouse press and release) is the event which corresponds to pressing the button.

VB uses a nomenclature such as command1_click() for event procedures. Since we've not yet discussed procedures this information is a bit premature. For now, just take on faith that VB provide a place to write the code which will be executed when the button is clicked.

To get to the location where the code will be placed, DOUBLE-CLICK on the button. A new window, the CODE WINDOW, will appear with the cursor in place, ready for you to type the code. Type the following line of code:
PRINT "Hello World"

- **Step 5.**

Your program is now complete and ready to run. Within the IDE you simply press F5 to run the program. Do so now.

Your application will appear as a single window in which there is a single button labeled "Print". Press the button with your mouse and you will see the words "Hello World" appear in the upper left corner of the window.

That's it! You've just written your first VB application.

To return to the IDE, click on the "X" button at the top right of the form. Later we'll discuss better ways to exit from a program.

Projects

The end result of most VB programming efforts is an executable program (one that has a .EXE extension). This file is compiled from the actual text files which make up a VB project. VB doesn't put the entire project into a single file. Instead, it allows the programmer to break up a project into several smaller files. Each of these files can be used in more than one VB project. The group of files used to compile the application is called a VB project and is the topic for discussion in this section of the tutorial.

Project Definition

A typical VB application might consist of more than one forms, each of which may have multiple controls. In VB, the information about each form (it's own properties as well as those of the controls that are on the form) is saved into it's own file. In fact, there are several types of files which VB creates (we'll cover them all later in the tutorial).

VB also saves a "project" file which contains the list of files which VB loads when the "project" is loaded. The project file includes other information, such as the filenames of controls which are not intrinsic to VB (they exist as separate files with .OCX extensions).

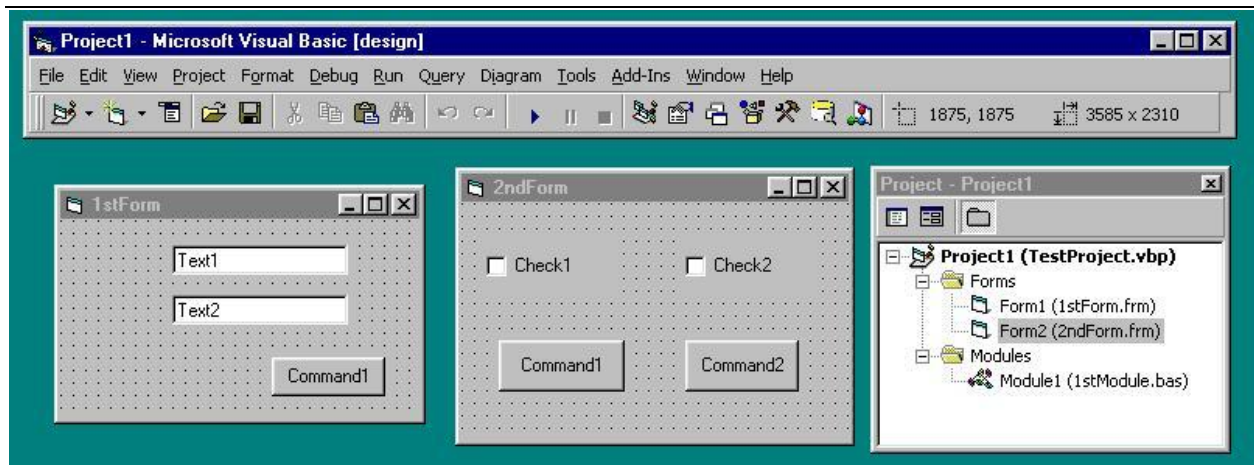
The project file is a simple ASCII text file and may be edited with any text editor, including NOTEPAD (which comes with Windows). Once you learn the format of the project file you may have reason to edit it directly but in general, most programmers do their editing within the VB IDE.

The project file is saved with an extension of .VBP (Visual Basic Project). Other files saved by VB include the following extensions:

- .FRX : graphics associated with the form
- .BAS : code not associated with control events

The good thing about splitting a project into many files is that you can use a file (form or otherwise) in one or more projects. If you make a change to the one file, all using projects see the change.

In the following VB IDE example, you see a project with two forms, each with multiple controls. The filename of each form is displayed in the project window, as is the name of a single .BAS file. Together, all three files make up the VB project.



Here is the content for each of the files. You will note that the files are in ASCII text. By inspection, you can guess what a lot of the line items are for.

- **Project File**

```
Type=Exe
Form=2ndForm.frm
Reference=*\G{00020430-0000-0000-C000-000000000046}#2.0#0#..\WINDOWS\SYSTEM\STDOLE2.TLB#OLE Automation
Form=testform.frm
Startup="Form1"
Command32=""
Name="Project1"
HelpContextID="0"
CompatibleMode="0"
MajorVer=1
MinorVer=0
RevisionVer=0
AutoIncrementVer=0
ServerSupportFiles=0
VersionCompanyName="Personal"
CompilationType=0
OptimizationType=0
FavorPentiumPro(tm)=0
CodeViewDebugInfo=0
NoAliasing=0
BoundsCheck=0
OverflowCheck=0
FlPointCheck=0
FDIVCheck=0
UnroundedFP=0
StartMode=0
Unattended=0
Retained=0
ThreadPerObject=0
MaxNumberOfThreads=1
```

- **Form 1**

```
VERSION 5.00
Begin VB.Form Form1
Caption = "2ndForm"
ClientHeight = 2280
```

```

ClientLeft = 5805
ClientTop = 2055
ClientWidth = 3735
LinkTopic = "Form1"
ScaleHeight = 2280
ScaleWidth = 3735
Begin VB.CheckBox Check2
Caption = "Check2"
Height = 495
Left = 2160
TabIndex = 3
Top = 360
Width = 1215
End
Begin VB.CheckBox Check1
Caption = "Check1"
Height = 495
Left = 240
TabIndex = 2
Top = 360
Width = 1215
End
Begin VB.CommandButton Command2
Caption = "Command2"
Height = 495
Left = 2160
TabIndex = 1
Top = 1320
Width = 1095
End
Begin VB.CommandButton Command1
Caption = "Command1"
Height = 495
Left = 360
TabIndex = 0
Top = 1320
Width = 1215
End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Explicit

```

- **Form 2**

```

VERSION 5.00
Begin VB.Form Form2
Caption = "Form2"
ClientHeight = 1905
ClientLeft = 6600
ClientTop = 2025

```

```
ClientWidth = 3465
LinkTopic = "Form2"
ScaleHeight = 1905
ScaleWidth = 3465
Begin VB.CommandButton Command1
Caption = "Command1"
Height = 375
Left = 2040
TabIndex = 2
Top = 1320
Width = 1095
End
Begin VB.TextBox Text2
Height = 285
Left = 1080
TabIndex = 1
Text = "Text2"
Top = 720
Width = 1695
End
Begin VB.TextBox Text1
Height = 285
Left = 1080
TabIndex = 0
Text = "Text1"
Top = 240
Width = 1695
End
End
Attribute VB_Name = "Form2"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Explicit
```

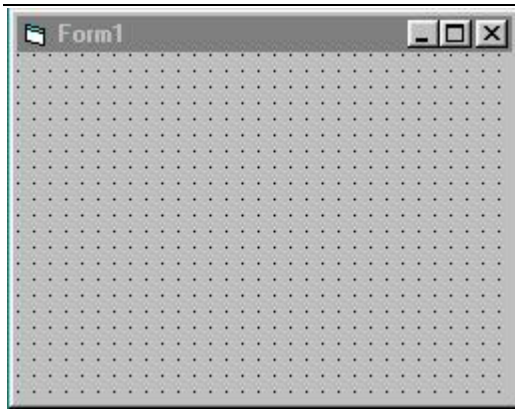
As you go through more of the tutorial, the content of the files will make more sense, but even now you can see that the information is not that difficult to decipher.

Forms

Visual Basic forms are windows. It's an important piece of data because it ties the concept of a form in with everything you already know about Windows applications. These rectangular shaped areas of the computer screen are called windows and the whole strategy of the Windows Operating System is to manage the display of those windows (while running the code that generates them or which performs calculations in the background). Since you've already been exposed to other Windows programs, then you already intuitively understand the concept of a form (window)! This section provides additional details about how VB handles forms.

What is a Form?

Here's a simple Visual Basic form. It looks just like any other form that you use in Windows applications. The header area has a caption, the control menu, and the minimize/maximize/close buttons. The the large area of the form is called the client area.



Don't be shocked, but all Windows/NT programs consist of one or more windows. In its simplest form, a window simply consists of a rectangular area of the screen. Anything that appears inside that area is considered to be part of the window. However, you can have one window contained inside another. Control objects, which are also implemented as windows, will be framed by the form window to which it belongs. As an operating system, Windows 9.X/NT controls the display of the various, possibly overlapping, windows on the screen.

In Visual Basic, the basic building block of an application is a form, which is simply a window. The VB IDE can insert forms into your project, and then you can resize the forms as well as change other properties of the form.

However, controls (checkboxes, textboxes, ...) are also windows. A form is distinguished from a control in that only forms can exist as standalone objects. When controls are used, they must be placed in a form. Ok, there are a few exceptions such as the printer object or the screen object which are not considered part of any form, but are part of a VB program. I'll talk to these special "system" objects later in the tutorial.

Not to confuse the issue, but controls can also be placed inside of other controls. When this happens the parent control is known as a container. Likewise, forms are containers but are the highest level of container there is in a windows application. Forms are always parents of controls, never the other way around.

There is one exception which I will not cover in these tutorials, and that is a special form called an MDI form. In this special case, an MDI form is always contained within a parent form. This is exactly the same type of parent/child relationship which you see in Word. Each new Word document is contained in its own window, but is always framed within the larger window that is the Word application.

The MDI (multiple document interface) forms can be very useful in applications where multiple files/images/documents need to be open at the same time. Other than this brief mention, I will not cover MDI forms in these tutorials.

Properties / Events / Methods

Now is a good time to bring up the 3 categories of information which may be used to describe any object, including forms. Forms, like any object, have properties which you may set. The properties range from the caption that the form displays to the physical size of the form. Later on this page I list all of the properties/events/methods that a form recognizes.

Likewise, a form may recognize certain events. All forms recognize the same events, but there are controls which recognize a broader range of events than forms. Events range from a simple keypress by the user to the click of a mouse button.

Then, finally, forms and controls also support various actions that may be taken. The actions are known as methods, and may include such tasks as moving the form, loading it into memory, or refreshing the form to redraw graphics which may have been overshadowed when one form was placed on top of another.

Remember that even though this part of the tutorial is focussing on forms, that Properties/Events/Methods apply to **all** objects in Visual Basic.

Given that there are over 20 controls available to you in the VB Pro edition, you might be concerned that learning all of the possible properties, events, and methods could be an overwhelming task. However, it's not at all that bad. Here's a very helpful piece of information that makes your task easier: **all forms, controls, or objects share many of their properties, events, and methods!**. Re-read what I just wrote! It's a very important piece of information and it means you can reuse what you learn about one control to help you learn about other controls. I've created a **control summary chart** (available in [Excel 97](#) and [Excel 5.0](#) formats. The chart gives the complete list of VB controls (provided in VB Pro) and lists their properties, events, and methods. The chart is listed in such a way that you can see the common items, as well as those which are unique to that control. You'll see that many controls have no unique items at all! In my chart I show 41 common properties, 20 common events, and 7 common methods. Please note that not every control uses all the common items! Some common items may be shared by only 2 or 3 controls.

I highly recommend that you look over the chart and become familiar with **all** of the items on it. I regularly get questions at my site where the programmer could have performed a desired task by simply setting a property of the control, **if he had just known about it!** Just having the chart gives you the ability to look and see if a control supports a feature that you need.

Detailed descriptions and sample code for using the items can be found in the VB HELP file. There is also a Microsoft book call "The VB6 Language Reference" which gives additional detail for each of the items.

As a prelude to the larger chart, here's a simple listing of the entire set of properties, events, and methods that are recognized by a form. Don't shy away from looking at this list in detail because you **will** use every one of these over and over again! Forms are particularly critical to the VB programmer because they are the fundamental building block for **every applications**.

Properties	Events	Methods
Name	Click	Refresh
Appearance	DragDrop	Drag
BackColor	DragOver	Move
BackStyle	GotFocus	SetFocus
BorderStyle	KeyDown	ZOrder
Caption	KeyPress	OLEDrag
CausesValidation	KeyUp	ShowWhatsThis
Container	LostFocus	
Enabled	MouseDown	
Font	MouseMove	
ForeColor	MouseUp	
Height	OLECompleteDrag	
HelpContextID	OLEDragDrop	
hWnd	OLEDragOver	
Left	OLEGiveFeedBack	
MaskColor	OLESetData	
MouseIcon	OLEStartDrag	
MousePointer	Validate	
OLEDropMode		
Parent		
RightToLeft		
Style		
Tag		
Text		
Top		
Visible		
WhatsThisHelpID		

Width

You'll note that forms, like most objects tend to have many properties, and that the number of events is much larger than the number of methods. A control can have well over 100 properties but normal count is usually around 50-70. You'll find that the name of most of the properties to be very self-explanatory (i.e., caption, name, fontsize, enabled, ...).

On the other hand, controls are not likely to have more than 25 or so events, and rarely has more than 10 methods. There are exceptions, but the generalization gives you a feel for what is involved with most controls.

The [control summary chart](#) lets you quickly get to details of which controls supports which item.

Controls Overview

Quick! How many controls come with VB? Can you list them? Well, it's not as easy as you might think. The documentation that comes with VB is a bit vague on exactly what is available to you. Even worse, I've found multiple places in the documentation which don't even say the same thing! Finally, not all that's on the CD-ROM gets installed on your PC, so you can't simply use the Components dialog box to determine every OCX control that VB has to offer! Because of that, I've created this part of the tutorial. I still have a separate section for Intrinsic (built-in) and ActiveX (separate OCX files) controls.

Introduction

When I was writing the sections on the Intrinsic and ActiveX controls, I realized that the documentation for the controls was somewhat confusing. The MS Programmers Guide, the MS Component Tools Guide, and the VB HELP files weren't consistent in their description of what controls were available, nor were they clear as to which controls come with the Learning and Professional editions of VB.

Intrinsic

Controls

In the next section of the tutorial, I cover the 20 controls which are built in to VB. These controls are called intrinsic controls. Every one of the intrinsic controls are available to every VB program you write. When you create a "New" VB project, all 20 of the intrinsic controls will show up in the Toolbox. This is true for all versions of VB.

ActiveX

Controls

Microsoft uses this terminology to refer to any control which exists as a separate file whose extension is OCX. To use an ActiveX control in your VB program, the control must be registered in the Windows Registry. Usually, the OCX control installation software handles the registration for you (such as VB does to register the controls it provides).

If you have a control which has not been registered by other software, you can register it yourself using the free program provided by Microsoft. The program, REGSVR32, comes with all version of Windows and is usually found in the Windows folder. To use it, simply type in:

```
REGSVR32 control.ocx
```

where the "control.ocx" is the filename of the control that you want to register.

IDE and Controls

Here's a key point to remember. Just because you register a control **does not** mean that you can use the control in one of your projects. Registration only assures you that the control can be used by an application at run-time. Whether or not a control can be used at design-time (within the IDE) depends on the control.

There are many free controls which can be used freely at design-time. However, commercial controls require that you install them using a password before you can use them within your project at design time. If you simply copy an OCX to your system and register it with REGSVR32, don't be surprised if you get an error message when you try to put the OCX on a form!

Intrinsic Control List

The following list shows the 20 intrinsic controls that come with all version of VB. I have a separate tutorial section to discuss them further:

 checkbox	 frame	 OLE Container
 combobox	 horizontal scrollbar	 optionbutton
 commandbutton	 vertical scrollbar	 picturebox
 data control	 image control	 shape control
 dirlistbox	 label	 textbox
 drivelistbox	 line	 timer
 filelistbox	 listbox	

Beyond the intrinsic controls, what you see and what you can install from the CDROM depend on which version of VB you've purchased. Not only that, but VB doesn't automatically install every possible OCX onto your system. Some OCXs (which can be used in the IDE) can be found on the VB CDROM. I'll show you where in just a minute!

Learning Edition ActiveX Control List

The next list shows the OCX controls which come with the VB Learning Edition. I show you which ones get automatically installed, and which ones you must manually install.

 Common Dialog	CMDLG32.OCX	automatically installed
 DataComboBox	MSDATLST.OCX	automatically installed
 DataList	MSDATLST.OCX	automatically installed
MSFlexGrid	MSFLXGRD.OCX	automatically installed

Professional Edition ActiveX Control List

These are the OCX controls which come with the VB Professional Edition. I show you which ones get automatically installed, and which ones you must manually install. In a later tutorial section I also discuss which controls are the most useful to programmers.

 ADO Data Control	MSADODC.OCX	automatically installed
 Animation Control	MSCOMCT2.OCX	automatically installed
 Communications Control	MSCOMM32.OCX	automatically installed
 CoolBar Control	COMCT332.OCX	automatically installed
 DataGrid Control	MSDATGRD.OCX	automatically installed
 DataRepeater Control	MSDATREP.OCX	automatically installed
 DateTimePicker Control	MSCOMCT2.OCX	automatically installed
DBGrid Control	DBGRID32.OCX	automatically installed
DBCombo Control	DBLIST32.OCX	automatically installed
DBList Control	DBLIST32.OCX	automatically installed
 FlatScrollBar	MSCOMCT2.OCX	automatically installed

	Grid Control	GRID32.OCX	automatically installed
	ImageCombo Control	MSCOMCTL.OCX	automatically installed
	ImageList Control	MSCOMCTL.OCX	automatically installed
	Internet Transfer Control	MSINET.OCX	automatically installed
	ListView Control	MSCOMCTL.OCX	automatically installed
	MAPI Controls	MSMAPI32.OCX	automatically installed
	Masked Edit Control	MSMASK32.OCX	automatically installed
	MonthView Control	MSCOMCT2.OCX	automatically installed
	MSChart Control	MSCHART.OCX	automatically installed
	Multimedia Control	MCI32.OCX	automatically installed
	MSHFlexGrid Control	MSHFLXGD.OCX	automatically installed
	PictureClip Control	PICCLP32.OCX	automatically installed
	ProgressBar Control	MSCOMCTL.OCX	automatically installed
	RemoteData Control	MSRDC20.OCX	automatically installed
	RichTextBox Control	RICHTX32.OCX	automatically installed
	Slider Control	MSCOMCTL.OCX	automatically installed
	StatusBar Control	MSCOMCTL.OCX	automatically installed
	SysInfo Control	SYSINFO.OCX	automatically installed
	SSTab Control	TABCTL32.OCX	automatically installed
	TabStrip Control	MCSOMCTL.OCX	automatically installed
	Toolbar Control	MSCOMCTL.OCX	automatically installed
	TreeView Control	MSCOMCTL.OCX	automatically installed
	UpDown Control	MSCOMCT2.OCX	automatically installed
	WinSock Control	MSWINSCK.OCX	automatically installed

Making Your Own Choices

I've shown you which controls are available but I've yet to tell you which controls are of any real use to you. You'll find that in applications you write, you will use a good sprinkling of the intrinsic controls. In my experience, about 90% of the controls on my forms have come from VB's intrinsic control list.

Once you select the intrinsic controls you need, you'll add to your project a few of the ActiveX controls which provide a specific feature you want in your application. You are very unlikely to have an application which uses all of the available ActiveX controls. You'll often find that adding 6-10 ActiveX controls will be the most that any application requires. If you're like me, you will have 3-4 very versatile controls which you use a lot, and a very few others which you use on a regular basis.

From the available ActiveX controls, I use the Toolbar, ImageList, and Common Dialog controls extensively. The Treeview, and Tab Control round off my list of personal favorites from the ActiveX control list. I probably should make more use of the Progress bar but most of my applications don't include tasks which make the user wait long enough to warrant the insertion of a Progress Bar. Likewise, using the StatusBar would certainly help my programs be more user-friendly, but I just don't seem to generate the enthusiasm to include it on my applications. I also depend on third party OCXs. I find that I use the Formula One spreadsheet and the Crystal Reports reporting controls very often. Other than that I have a few specialty OCXs which I use, particularly in the graphics areas.

Intrinsic Controls

VB comes with 20 built-in controls. In this section of the tutorial I provide a few comments about each one, trying to give some useful pointers on the use of each control. I strongly suggest that you review my **control summary chart** - available in [Excel 97](#) and [Excel 5.0](#) formats. The chart gives the complete list of VB controls - along with their properties, methods, and events. The spreadsheet will help you get a "big-picture" overview of the VB controls. I also strongly suggest you read the HELP file content for each of the properties/events/methods for these controls. If you don't know the item exists, then you won't know when to apply it in your applications!

Introduction

All controls are not equally useful. Some you will use on every application you write. Others you will use only when you have a special need for the features the controls offer.

When you start VB, you'll always find the intrinsic controls displayed in the toolbox. The controls are built-in to the VB files and do not exist in an external file (with a .OCX extension) the way the ActiveX controls do. In the toolbox, each of the controls has its own distinctive icon.



The Most Useful Intrinsic Controls

These nine intrinsic controls are pretty much used on every VB application I've written. Start your learning with these and then branch out. Further down on this page I have a brief comment on each of the controls.

 Command Button  Image

 CheckBox	 PictureBox
 TextBox	 ListBox
 Label	 ComboBox
 Option Button	

The Rest of the Intrinsic Controls

The other eleven intrinsic controls are also valuable but I find myself using these less often than the others. Also, you'll find that you use fewer of these within an application than you do of the nine that I listed as the most useful controls.

 ADO Data Control	 Vertical Scroll Bar
 DirListBox	 Line
 DriveListBox	 Shape
 FileListBox	 OLE
 Frame	 Timer
 Horizontal Scroll Bar	

Database Features

I've put the discussion of databases elsewhere in the tutorial, but you should know right now that several of the intrinsic controls can display or edit data directly out of a database. With VB, the ADO Data Control is used to access the database information and to distribute it on to the other intrinsic controls which can handle database information. VB uses the terminology "databound" to describe controls which have built-in features for handling database access.

Comments on Each Control

-  **Command Button**
 This one works just like you expect. Press the button and it executes a block of code.
-  **CheckBox**
 Typically this is used for turning on/off some particular feature of your program.
-  **TextBox**
 This is the standard way of letting a user edit information. To make it convenient for the users, learn about the .SELLENGTH and .SELSTART properties which highlight text in the the textbox.
-  **Image**
 Use this to display a picture. Use it over the PictureBox because it takes less operating system resources.
-  **PictureBox**
 While it can display pictures, it also acts as an area on which you can print text and graphics. Use it for home-grown graphics or print previews.
-  **Label**
 As the name suggests, this is used to label other controls. It's pretty passive and you'll seldom use its items other than the .CAPTION property.

-
-  **Option Button**
If you use it, you'll use it in groups. VB handles the feature that only 1 option button can be selected at a time.
 -  **ListBox**
This is the first of the intrinsic controls to introduce methods common to some of the more complex controls. The use of a ListIndex which starts at 0 (not 1) is a confusing factor that you must watch in your code.
 -  **ComboBox**
Whereas a listbox takes up space on the form, the combobox control minimizes the use of valuable form real estate. It has 3 modes of operation some of which allow you to keep your users from entering bad data.
 -  **ADO Data Control**
If you're not accessing a database, then you don't need this one. If you are accessing a database you have to have this control to act as the interface to any other databound control. The exception is that VB offers ways to access databases directly from code, but for modest display/edit applications the ADO Data Control is very effective.
 -  **DirListBox / DriveListBox / FileListBox**
You'll almost always use these in combination with each other. Read the HELP file for how to synchronize them to work together. Often, however, you will use the CommonDialog Control instead of these.
 -  **Line**
It can be used as a static display, or you can animate it with the .VISIBLE property and the .MOVE method.
 -  **Shape**
When the line is not enough, this one supports rectangles and ellipses/circles. As with the line control, use its items to create animation.
 -  **OLEContainer**
If you want to put objects on your VB application which come from other applications already on your machine (such as Word, Excel, ...) then this control is very useful. For my needs, I don't like making the assumption that my users have the application (in a specific version) on their machine to make distribution of my application go smoothly. I avoid this one whenever possible!
 -  **Frame**
It's just a container - it can hold other controls. There are two very good reasons to use it. If you want multiple groups of option buttons then place each group in a frame and each group will operate independently. If you want to manipulate controls as a group (i.e., positioning) then put them in a frame and you can handle them all at one time.
 -  **Horizontal/Vertical ScrollBar Controls**
Basically you let use the slider value of a scroll bar as the input for other code that you write. These are normally used in conjunction with other controls.

ActiveX Controls

Introduction

When you start VB, **none** of the ActiveX controls are displayed in the Toolbox. Only the intrinsic controls are displayed, so you must manually insert the ActiveX controls into the Toolbox as you need them. To do so, right-mouse click on the toolbox and go to "Components", select the controls to put on the toolbox and press "OK".

The Most Useful ActiveX Controls

Visual Basic: Institute of Industrial and Information Technology(IIIT): Council of Education & Training
Page 2

	Coolbar		Progress Bar
	ImageList		SSTab
	Internet Transfer Control		Tab Strip
	ListView control		ToolBar
	Multimedia control		

The Specialty ActiveX Controls

Some of the VB controls are very excellent tools, but they simply aren't needed in many applications. This list of controls are ones that would be on my list of the best controls except that they are just not needed that often:

	Communication control		RichTextBox
	Data Repeater		SysInfo
	MAPI Session		MAPI Messages
	Winsock		MSChart

The Rest of the ActiveX Controls

Of the remaining ActiveX controls, I find some useful but mostly they sit un-used on my PC. It doesn't mean they don't perform their features well, it just means that my own applications don't find the need for the features very often.

	Animation		MonthView
	DateTimePicker		PictureClip
	FlatScrollBar		Slider
	ImageCombo		UpDown
	MaskedEdit		

Comments on Each Control

One of the things you'll notice is that I use a few controls a lot, and don't make a heavy effort to use every available control. I've thought about it a lot and wonder if VB programmers out there are like me, or if everyone uses their favorites over and over? You'll have to decide for yourself which approach makes sense. In the comments that follow I provide some of the logic which pulls me to one of these controls, or what it is that keeps me away!

- 
Animation Control
 Personally, I don't do games and I find animation in business programs to be of marginal value (unless the animation is of real-time generated data). So, this is one of the controls I seldom use. Interestingly enough, I get a lot of questions from newbies on how to do animation and I steer them to this control.
- 
Communications Control
 There really are some excellent reasons to become familiar with the MCI control. Programs like Laplink are hugely successful, primarily over their ability to work through the serial and parallel ports of a PC. I believe that 98% of all programmers never have that need. For the remaining two percent, however, this control can be a god-send.
- 
CoolBar Control
 The concept is great - give your program a look and feel like that of other programs that your users are familiar with. In practice, most of our users have been exposed to so many windows programs that an Explorer-like interface or a Win3.1 application interface are

all so easily understood that unless your user is picking between your program and a second one, I don't believe the look-and-feel will buy you any more users. Take advantage of it if you want, or just use what you're familiar with.

-  **DataRepeater Control**
I tend to do things manually. Any control which automatically performs tasks for me tends to be ignored so long as I can get the job done with tools I'm already familiar with. This is a habit I really need to break. As I've said before, a programmer's job is to get the job done quickly and economically. Under those guidelines, controls such as the data repeater should be high on my list. I'll work on it!
-  **DateTimePicker Control**
So far, I've never written a program which needed a calendar. I've done many where a date was entered but usually the date was the current date and that was easy to provide to the user. I can see where some programmers could put this to good use.
-  **FlatScrollBar**
The only thing that would entice me to use this is that it can be made to take on both horizontal and vertical orientations. That makes it pretty versatile. Otherwise, I'm not interested.
-  **ImageCombo Control**
This solves a problem which a lot of VB programmers have asked for, putting graphics in a list. It's more effort to program than a simple text list, but if you want the graphics then it's the way to go.
-  **ImageList Control**
I use the Toolbar control a lot! Since the Toolbar gets its images from this control, then that means I have to give this one a thumbs up! I need it, I like it, and I use it all the time.
-  **Internet Transfer Control**
From what I've read, this is much improved over the version that came with VB5. Many of my users are corporate employees from engineering areas. Their needs rarely require an over-the-next solution so my experience on this one is minimal. However, since it is the **only** internet control in VB, then by definition you will use it if you have needs for Internet applications.
-  **ListView Control**
The four controls - the listbox, the common dialog control, the TreeView and this one (ListView) can pretty well handle display of data in simple or heirarchical methods. It's a useful control, but not one that you cannot live without.
-   **MAPI Controls**
This is an excellent example of a control which is very specialized, so much so that no one I know uses it. I have no doubt someone does, but I just haven't met them yet.
-  **Masked Edit Box Control**
You'd think that this control would be one of the most popular there was and that Microsoft would have made it an intrinsic control. But, it just didn't turn out that way. The earlier versions of the control gave it a very bad reputation and it doesn't seem to

have recovered from it. It's not exactly the easiest to work with and the actual operation isn't as trouble-free as I'd like. I don't use it but I think about it a lot!

- **MonthView Control**

Calendars in a program out to be very useful. Every PIM has one so why haven't my own applications had them? Maybe if I'd had this control in VB5 I'd be further along the learning curve.

- **MSChart Control**

Yep - when I need a chart I use this one. Actually I don't like it all that much. It has a huge number of properties and I never seem to get all of them working together the way I want. Even so, it comes with VB and it can be made to do the job, so I use it.

- **Multimedia Control**

As I mentioned, I mostly do engineering-oriented applications. The use of .wav and .avi files would be fine, but for my needs I would want to create the .wav/.avi files myself and VB has no means of doing so. If you can use stock files from someone else, then this control is really a valuable tool. It's the only control in VB that does what it does, so it must also be the best one!

- **PictureClip Control**

Multimedia presentations and applications really are great. But as all artists know you can't just create a quality image in seconds. Having this control available to house images is fine, but where do all those images come from? Like with the Multimedia Control, this one is valuable to folks who have access to images that already exist and which meet their own business needs.

- **ProgressBar Control**

This is the one where I am really delinquent. I owe it to my users to apply this control much more often than I do. I recommend that you use it whenever possible. The techniques of changing the cursor to an hourglass is good, but a progress bar is much better!

- **RichTextBox Control**

This is very much a specialty control. For 99% of my user inputs, a simple text input is just fine. I've gotten a lot of email from users who are using this control so I know that the lure of better looking text must be hard to resist. However, I resist it and don't really have any reason to recommend this one to you unless you are writing some kind of word processor or if you really need to display formatted text.

- **Slider Control**

Yawn. Well, maybe not. There are times when adjusting something to get a value is needed. But I could do that with the intrinsic scrollbars already. This control is easier to use but I wish Microsoft had spent their money getting a totally new feature rather than a better old one.

- **StatusBar Control**

This one is an excellent way to give your user messages about the status of the application or about the status of a user request. I don't know exactly why I don't use it more, but I'm certain that my bad attitude will change. I recommend you learn how to use this one and that you apply it to virtually every program you write.

-
-  **SysInfo Control**
Based on questions I get from users this one should be a big winner. There are all kinds of reasons why a programmer needs to know about the PC on which his .EXE is running. This control has made a good start in providing that kind of data to an application. I'd rather have seen it built in to VB as functions but until then you can use this control to get data that otherwise would require the use of the Windows API.
 -  **SSTab**
 -  **TabStrip Control**
 -  **Toolbar Control**
The Common Dialog box is easily the most useful ActiveX control, and the Toolbar control takes second place. The control is reasonably easy to use and I've never had any trouble with it - it works as advertised! I highly recommend you learn to use this one.
 -  **TreeView** **Control**
Except for the lack of a built-in ability to save its content, this control has a lot of application potential. I've used it or one of its predecessors in every version of VB since VB3. Learn to use it right and it can provide a very strong user interface for the display of ordered data. If you can't figure out how to save a tree structure, email me and I'll let you in on the code I came up with.
 -  **UpDown Control**
Pretty simple, but actually pretty useful. It should have been intrinsic too, but it isn't. You won't have to spend much time on it to become an expert so just learn it and then put it on the shelf until you need it. You may not need it often but when you need what it does, you'll be pleased that it is available.
 -  **WinSock Control**
I don't use it. However, lots of my visitor do ask questions about it and it seems to be a pretty popular control. As soon as I bite the bullet and get more into Internet programming I suspect this one will move up the list of favorites.

Summary

As I said, I tend to use a few controls often and a lot of controls very seldom. If I was an expert in the controls I use infrequently, would I tend to use them more often? It's the old problem of not knowing what you don't know. I plan to work on stretching my control selection to give all of my less favorite controls a better chance. In a year, we'll see if my list of favorites has changed!

Code

In this section I'll cover the topic of coding - the use of the Visual Basic language. As part of this section I'll try to show you which elements of the language you need to know the most, and which elements work together. Even more than you saw with some of the intrinsic controls, there are groups of the VB language which are almost always used together because they cover different aspects of a problem.

Overview

It's worth noting that when managers talk about programmers, one of the common metrics used to describe performance is "lines of code per month". There's all kinds of debate about how good a metric this is, but the fact is that the metric is used!

It's not that you don't get credit for novel algorithms, or that you won't be a hero to fellow programmers when they see how you solved a problem with 10 lines of code that took them 100. You'll get that credit (and mental satisfaction, too!) but looking at the big picture it's clear that the volume of code you can crank out will be the visible result of your efforts!

Let's walk again through the elements of writing a VB application. The percentages at the right are an estimate of how much time a programmer might spend on the various phases of the program. For short programs, a greater percentage would be spent on the concept and user interface phases. For larger programs, more time will likely be spent on the later phases. However, in either case you can see that the coding section is one of the key areas in which a programmer will be spending his time.

- Concept / Requirements Definition (10%)
- User Interface Design (20%)
- Coding (40%)
- Test & Debug (20%)
- Customer Acceptance / Evaluation (10%)

Get the picture? If you want to be a great programmer, you have to know how to code!

There's another aspect to coding that you need to be aware of, but it's a little harder to explain. In the electronics manufacturing business there's a saying that 90% of the cost of a product is determined in the design. Can you see the link? Writing a program is essentially a design task, aimed at solving a problem. Like most things, there are many design approaches for every problem. Some solve the problem by grinding out an answer. Some are very elegant. Others are so complex that even the designer has a hard time keeping up with the convolutions of the approach.

Remember that in my [Training/Advice](#) I harped on the idea that a programmer's job was to economically provide the result for which he is being paid. A customer will be more impressed by a job done on time, on budget and which meets spec than he will be for a project with overruns because of bells and whistles (or unusual code) which do not improve the utility of the application to him or his employees.

You'll get this kind of philosophy to programming throughout this tutorial.

Let's see if we can tie all this together now.

1. Great programmers must be coding experts
2. Great programmers understand that there are many approaches to a problem
3. Great programmers work efficiently. Remember my [Rule 4!](#)

As I go through the rest of the tutorial I'll try to show how these principals can be applied to real life decisions and solutions.

Language Grouping - According to Beene

My overriding approach to learning is take what I read and try to piece it together with everything else I've learned. I think of all knowledge as one big puzzle and it's important to know where each piece of information goes in the puzzle in order to understand it's importance. In programming, knowledge is certainly a tool and as my daddy used to say, "It's the poor carpenter who blames his tool!".

The immediate value of this pearl of wisdom is shown in the following chart.

- [Visual Basic Command Grouping Chart](#)

Beginner's who read through the Microsoft manuals, particularly the Language Reference manual, are often intimidated by how darn many commands they find. An alphabetical listing is

simply no help in getting your arms around the problem. Back in the DOS days, several of the BASIC manuals groups commands much like what you see in the table above. This one is one that I did myself, while **reading through the entire Microsoft VB Language Reference!** The value of such a grouping is twofold. If I had it to start with then I could have "bitten" off commands in small chunks. The small doses would consist of commands which act together to let me perform categories of tasks (file I/O, user input, loops, string manipulation, etc.). Secondly, by creating my own grouping, I forced myself to think about the code and to understand not only what the command can do, but how it can be used with other commands to get a job done!

This brings me to another philosophical point. You noticed that I mentioned having **read** the entire Language Reference? On my Beginner's page you may be noted that I also have read the Programmer's Guide front to back not once, but three times? I am a staunch believer that reading material can only be absorbed little by little. You read something once and you get 70% of it. Read it again and you get up to 90%. Third time brings you up to 95%. Better yet, write your own tutorial and you'll get some of that last percentage .

If you want to learn code, then you **have to read the manual** over and over til it makes sense to you. Each time you learn something, you're better able to learn something else. That's why rereading a manual is so valuable. The skills you pick up in the first reading make it easier to understand the parts you didn't understand the first time. I encourage you to read, read, read and while you're doing it, take notes and later summarize what you've learned. It sounds tedious, but then coding is not exactly all glitter either!

All of you have seen the acronym RTFM, right? Well the popularity of this phrase just backs up what I've been saying. You'll get no sympathy from me for misunderstanding a topic unless you can show me where you studied the topic. Of course, it's okay to not understand the manual's explanation of a topic (Microsoft's manual are good, but they are definitely not great!) but you owe it to the person you're asking questions of to have done the leg work yourself **before** drawing on their experience to help you out.

Variables

I know you have been getting anxious to get to the **coding** part which I keep referring to, but there's one more lesson you need before we get there. The lesson is about variables. Simply, a variable is a name you use in your program which refers to a value. You've probably already seen variable names like **x**, or **i**, or **name**, or

You've probably already heard that values can be represented in many ways, such as integers, floating point values, strings, objects, etc.

Here's the most important list of variable types you'll run across in VB:

Integer 1, 2, 3, ... with the largest value = 32767

Long Like integers, but with the largest value about 2E9

Single 0.2, 5.7, 24.12 ... any number with a decimal

Double Like Single, but with the largest value about 2E308

Strings Not numbers at all, but letters or special characters

This isn't the whole list, but it's definitely the ones you'll use 99% of the time in your applications. I'll bring up the others as we need them in the tutorial.

This list is **not** just an exercise in useless knowledge. Later on you'll see how the use of a particular variable type can be of use in speeding up your program or of how using the wrong variable type can actually result in a wrong answer in some calculations! What you don't know can really hurt you!

File I/O

Often, books on VB don't cover this topic until much later in the book. I'm covering it early because it influences how I've grouped the VB language into the chart above.

A fundamental thing to understand is that computer files are usually stored in two basic ways, ASCII and binary.

1. **ASCII** is just an abbreviation which refers to storing information as string data. Each letter or number has a value (for example, the letter 'a' has the ASCII value 95) which you can find in VB Help by searching on ASCII. The value 12 is stored as a two character string, '1' and '2'. This takes two bytes of hard disk space.
2. **Binary** describes a way of storing data more efficiently than ASCII. The value 12 would be stored as a single byte value represented by the binary string 00001100. This approach holds for all numbers although larger numbers may take more than a byte of storage. Strings cannot be stored this way and are always stored as one byte per character.

Visual Basic has commands which can read or store in both storage methods. The storage method you use can make a big difference in how simple the code can be in your application. I won't cover all the details now, but will bring it up as needed in the tutorial. For now, here's the summary:

1. ASCII storage is most useful for storing strings. When **both** numbers and strings are stored, ASCII is normally used. VB commands for ASCII file access are usually the least complex I/O commands, but are not as flexible as binary file I/O commands.
2. Binary storage is more useful for storing numbers. VB commands let you read one or more bytes of binary data at a time and it provides more flexibility in moving around through the file.

There are, of course, other schemes for storing data. Many programs such as Word, Excel, and PowerPoint have their own storage schemes. This is also true of database program files such as those created by Access or dBase. This tutorial won't discuss proprietary formats except to note that they exist.

Printing

VB has commands which allow you to print information, either to the computer screen or to a printer. For the printer, VB creates an empty "canvas" to which your commands write, then dumps the "canvas" contents to the printer on demand or at the conclusion of a program. For me, the single most tedious aspect of coding is that of printing reports. I **strongly advise** that whenever possible, that you make use of reporting tools such as the Crystal Reports control that comes with Visual Basic. Crystal Reports provides a very fast method of creating printed reports against data contained within Access databases, which is native database format supported by Visual Basic.

Procedures

One fundamental aspect of VB which will become second nature to you is the idea of procedures.

When we talk about VB commands, such as "Print", or "Open", we are talking about a command which tells VB to run a section of code which performs the Print or Open function. The code is contained in the various VB files that were installed on your machine, such as DLL files that get distributed with your applications.

With VB, like most languages, you can create your own "commands" but we call them procedures instead. The two types of procedures VB supports are called Subroutines and Functions.

1. **Functions:** These procedures are a section of code which is run when the Function is called, and a single value is returned to the calling line. For example, when you use the square root function in VB:

x = SQR(25)

This is a VB function called SQR. It passes the value 25 to the function and the function returns the square root and places it into x.

2. **Subroutines:** These procedures are just like a Function, but no value is returned.

It's important to note that procedure calls include a list of values (or variables) which are passed to the procedure. The procedure is allowed to change those variables when it is executed - but you make the decision by the choice of your code within the procedure.

Okay, it's coding time!

Okay, let's begin to pick apart the subject of coding. You'll remember that I described coding as something you put into an event procedure that VB will execute when the event occurs? Well, that's true but you can also add code to your application which is not contained in an event procedure, but which **must** be called from within those event procedures.

Here's how it works. If you double-click on an object in the IDE you'll be presented with the code window for that object. The upper left dropdown list shows the list of objects in your application and the upper right list shows the events that each object supports! It's an excellent way to explore the events your objects can support. If an event has had code added to it, the event list makes the event title **bold**.

If you look at the top of the left dropdown list, you'll see a listing that is labeled "General". Remember those Function or Sub procedures we discussed earlier? Well you'll find all of them in this "General" section. VB automatically collects all of the user-defined function and sub procedures into this one area.

So, back to a higher level of discussion: When you enter code, you first think to put it in the events of your project objects. If you have special function or sub procedures to write (mostly because it will be used in multiple locations throughout your application) then you create a Sub or Function to put the code. Sub/Function procedures work identically except VB controls when an object event procedure is executed, whereas the code you write in the event procedures can specifically call out for your custom Sub/Functions to be executed.

By the way, you can also cause an object's event procedure to be executed at any time. For example, if you have a button (Command1), then its click procedure is command1_click. If you type this procedure name in as a line of code, then it will execute, **just as though you have clicked the button with your mouse!**

Helpful Tip!

Visual Basic has some built-in variables which can make your code very easy to read, and more importantly, are much easier to remember than the actual value of the variable. Here's how it works:

Suppose you want to set the .CHECKED property of a checkbox to its checked state? If you look into HELP you'll see that you must set the .CHECKED property to a value of "2" for it to be checked. Easy enough? Well, consider across all the controls that there are hundreds of properties to remember. VB comes to the rescue with built-in variables. In our example, you could set the .CHECKED property to the built-in variable VBCHECKED, which VB will recognize as a "2". As you would guess, there is also a VBUNCHECKED. Both of these are visually self-explanatory, whereas if you used an integer (2 or 1, in this case) you might trouble figuring out which state the code is intended to create.

VB has a lot of these easy-to-remember constants. When you read HELP for the syntax of a command/function, you will find the list of built-in constants which can be applied. **Get in the habit** of using these built-in constants to avoid the hassle of memorizing the actual values themselves!

Command Descriptions

In the earlier lesson you were introduced to coding, and reviewed a chart which groups VB commands into useful categories. In this section of the tutorial, I take each one of the commands and provide the syntax and a brief description of what the command does. For details, you can turn to the VB HELP file. My intent is to allow you to skim quickly over the commands to

determine which one can help out in your situation. This page is kind of long, but I wanted to keep it all on one page to make it easier for you get scroll through it. I've also summarized the contents of this page in a [command summary chart](#).

Operators

Here are the VB operators used to perform mathematical operations on one or more variables. Aside from the normal multiply/add/subtract and divide, you will find the AND, OR, Not Equal, MOD and Integer Division operators very useful.

- / - Normal division
- \ - Integer division (truncates the answer)
- ^ - Exponentiation operator
- * - Multiply
- + - Plus
- - - Minus
- = - Equal
- > - Greater Than
- < - Less Than
- <> - Not Equal
- >= - Greater than or equal
- <= - Less than or equal
- AND - Defines a boolean value that is the AND of two values
 - result = expression1 AND expression2
- OR - Defines a boolean value that is the OR of two values
 - result = expression1 OR expression2
- XOR - Defines a boolean value that is the exclusive OR of two values
 - result = expression1 XOR expression2
- NOT - Defines an opposite boolean value
 - A = NOT B
- EQV - Performs a logical equivalence on two expressions (result is true if both expressions are true)
 - result = expression1 EQV expression2
- IMP - Performs a logical implication on two expressions
 - result = expression1 IMP expression2
- IS - Determines if 2 variables reference the same object

-
- result = object1 IS object2
 - LIKE - Determines if one string matches a pattern
 - result = string LIKE pattern
 - MOD - Returns the integer remainder of a division
 - i = 27 MOD 5

Math

VB also provides built-in functions which can act on variables. Most are self-explanatory. In my experience, the VAL, RND, and ROUND functions are among the most valuable, so be sure to pay close attention to them!

- Round - Rounds a number to a selectable number of decimal places
 - result = round (tempvariable,2)
- Val - Returns the numerical content of a string
 - result = Val ("123.4")
- Int - Returns an integer by truncating (different than Fix)
 - i = int (tempvariable)
- Fix - Returns an integer by truncating (different than Int)
 - i = fix (tempvariable)
- Hex - Returns the hexadecimal value of any number
 - temp\$ = hex (tempvariable)
- Oct - Returns the octal value of any number
 - temp\$ = oct (tempvariable)
- Tan - Returns the tangent of an angle
 - tempvariable1 = tan (tempvariable2)
- Rnd - Returns a random number between 0 and 1
 - tempvariable1 = rnd
- Randomize - Initializes the Rnd function so it gives different answers each time
 - randomize
- Sgn - Returns the sign of a number
 - i = sgn (tempvariable)
- Sin - Returns the sine of an angle
 - tempvariable1 = sin (tempvariable2)
- Cos - Returns the cosine of an angle

-
- tempvariable2 = cos (tempvariable)
 - Abs - Converts a number to a positive value
 - i = abs (tempvariable)
 - Sqr - Returns the square root of a number
 - tempvariable1 = sqr (tempvariable2)
 - Log - Returns the base 10 logarithm of a number
 - tempvariable1 = log (tempvariable2)
 - Atn - Returns the arctangent of an angle
 - tempvariable1 = atn (tempvariable)
 - Partition - Sort of an oddball function but segregates values according to ranges
 -
 - Type Conversions - A variety of conversion functions
 - CBool, CByte, CCur, CDate, CDb1, CDec, CInt, CLng, CSng, CStr, CVar

Strings

In my experience these functions are used more than just about any of the other VB built-in functions. The FORMAT, MID, and INSTR functions are incredibly powerful and I use them extensively. If you don't understand what they are, they are **worth the time to figure out!** The LEN and CHR functions are also valuable as are the variations on the trim and case functions.

- Left - Returns the left n characters of a string
 - temp\$ = left\$ (teststring\$, 4)
- Right - Returns the right n characters of a string
 - temp\$ = right\$ (teststring\$, 4)
- Trim - Removes leading and trailing spaces of a string
 - temp\$ = trim\$ (teststring\$)
- LTrim - Removes only the leading spaces of a string
 - temp\$ = ltrim\$ (teststring\$)
- RTrim - Removes only the trailing spaces of a string
 - temp\$ = rtrim\$ (teststring\$)
- UCase - Makes all characters upper case
 - temp\$ = ucase\$ (teststring\$)
- LCase - Makes all characters lower case
 - temp\$ = lcase\$ (teststring\$)

-
- Mid - Returns n characters from a string, starting a any position
 - temp\$ = mid\$ (teststring\$, 1, 4)
 - Len - Returns the length of a string (how many characters it has)
 - temp\$ = len (teststring\$)
 - LSet - Positions a string inside another, flush to the left
 - temp\$ = lset (teststring\$)
 - RSet - Positions a string inside another, flush to the right
 - temp\$ = rset\$ (teststring\$)
 - Format - Returns a string formatted according to a user-defined format
 - temp\$ = format\$ (teststring\$, "####.0")
 - String -
 - temp\$ = left\$ (teststring\$, 4)
 - Chr - Returns the string representation of a number
 - temp\$ = str\$ (32)
 - Asc - Returns the ASCII code of a single character
 - temp\$ = asc ("A")
 - Space - Returns n spaces
 - temp\$ = space\$ (15)
 - Instr - Determines if one string is found within a second string
 - i = Instr (starthere, string1, string2)
 - InStrRev - Determine if one string is found in a second, starting at the end
 - i = InStrRev (string1, string2, start)
 - StrComp - Compares two strings
 - result = StrComp (string1, string2)
 - StrConv - Converts the case of a string's characters
 - StrConv (string, vbuppercase)
 - StrReverse - Reverses character order in a string
 - StrReverse (string1)
 - Replace - Replaces each occurrence of a string
 - Replace (bigstring, searchstring, replacementstring)

-
- FormatCurrency - Returns a string using a currency format
 - FormatCurrency(var1, 2)
 - FormatDateTime - Returns a date or time expression
 - FormatDateTime("3/2/99", vbShortTime)
 - FormatNumber - Returns a number formatted according to a variety of options
 - FormatNumber(var1, 2)
 - FormatPercent - Returns a number formatted as a percent
 - FormatPercent(var1, 2)

Arrays

Every programmer eventually uses arrays. Mostly they're pretty easy to understand. Take note, however, that you can resize an array with REDIM without losing the data. For details, see the PRESERVE keyword in the HELP entry on REDIM. If you use the LBound/UBound in your code instead of hard-coding the dimension of the array, you can later change the size of the array without touching your code!

- Option Base - Determines whether the lowest range of an array is 0 or 1
 - option base 1
- Erase - Erases all values of an array
 - erase (arrayname)
- Dim - Creates an array
 - dim arrayname(25)
- Redim - Resets the bounds of an array (has option to save values)
 - redim arrayname(28)
- UBound - Returns the upper dimension of an array
 - i = ubound (arrayname)
- LBound - Returns the lower dimension of an array
 - i = lbound (arrayname)
- Filter - Returns a subset of an array based on a filter
 - Filter (inputarray, searchstring)
- Array - Yes, there is a function called array. It returns an array that has been filled with data from a list. It allows you to put the actual data values in the code to avoid having the user input it or to avoid having to read it from a file
 - ArrayName = Array (10, 20, 30)
- Join - Concatenates strings within an array

File Handling (Generic)

While VB is working on a better approach (FileSystemObject), the built-in file handling statements are still the only way to access data other than through the VB database capabilities. Your skills in this area can make or break your ability to work with various formats. The OPEN/CLOSE statements are critical to success, but the LOF, EOF, and LEN functions are used even more often! It's also a given that you'll use the DIR function regularly.

- Dir - Returns a filename that matches a pattern
 - temp\$ = Dir ("*.*)")
- CurDir - Returns the current directory
 - temp\$ = CurDir
- Mkdir - Creates a directory
 - mkdir ("newdirectoryname")
- ChDir - Changes the current directory to a new location
 - chdir ("newdirectoryname")
- ChDrive - Changes the current drive
 - ChDirve "A"
- Rmdir - Removes the indicated directory
 - rmdir ("directoryname")
- Freefile - Returns an unused file handle
 - i = freefile
- Open - Opens a file for access, locking it from other applications
 - open "filename" for input as #1
- Close - Closes a file so that other applications may access it
 - close #1
- LOF - Returns the length of a file in bytes
 - i = lof (#1)
- EOF - Returns a boolean value to indicate if the end of a file has been reached
 - statusvariable = eof (#1)
- Name As - Renames a file
 - name "filename1" as "filename2"
- Kill - Deletes a file
 - kill "filename"

-
- FileAttr - Returns attribute information about a file
 - i = int (tempvariable)
 - GetAttr - Returns attributes of a file or directory
 - i = GetAttr("c:\windows\temp")
 - SetAttr - Sets the attributes of a file
 - SetAttr pathname, vbHidden
 - Reset - Closes all disk files opened by the OPEN statement
 - Reset
 - FileDateTime - Returns data file was created or last edited
 - FileDateTime (filename)
 - FileLen - Returns length of file in bytes
 - FileLen (filename)
 - FileCopy - Copies a file to a new name
 - FileCopy sourcefile, destinationfile
 - Lock - Controls access to a part or all of a file opened by OPEN
 - Lock #1
 - UnLock - Restores access to a part or all of a file opened by OPEN
 - UnLock #1
 - Width # - Set the output line width used by the OPEN statement
 - Width #2, 80

File Handling - ASCII-specific

While VB is working on a better approach (FileSystemObject), the built-in file handling statements are still the only way to access data outside of a data base. Your skills in this area can make or break your ability to work with various formats. The OPEN/CLOSE statements are critical to success, but the LOF, EOF, and LEN functions are necessary to build useful code.

- Line Input - Reads an entire line of ASCII text
 - line input #1, tempvariable\$
- Write - Puts data in a file, with separators for the data
 - write #1, tempvariable\$
- Print - Puts data in a file with no separators
 - print #1, tempvariable\$
- Spc - Used in a print statement to move a number of spaces

-
- Print #2, var1; spc(15); var2
 - Tab - Used in a print statement to move to TAB locations
 - Print #2, var1; Tab(20); var2

File Handling - Binary-specific

VB also support features which allow you to access a file on a byte-by-byte basis. The good thing about it is that you have more control, the bad thing is that you may have to write more code. Generally, a programmer will use the option (ASCII or Binary access) according to the **least** code he has to write. For binary access the Get/Put are equivalent to the Line Input and Print functions used in ASCII text file access. The big difference between the two is that binary access will read (Get) an exact number of bytes of data, and the reading can start at any byte within the file.

- Get - Reads data from a file
 - get #1, anyvariable
- Put - Puts data into a file
 - put #1, anyvariable
- Seek - Moves the current pointer to a defined location in a file
 - seek #1, 26
- Input
 - input #1, anyvariable
- Loc - Returns current position with an open file
 - i = Loc(#2)

Declarations

I probably get more questions about the functions in this section than about any other group. In general, the concepts are pretty simple, but the details of getting it exactly right can cause even experienced programmers trouble. Focus on understanding Dim/ReDim/Public/Private/Sub/Function/Type and Set. However, they're all useful at times, so bear down and commit these to memory. I'll try to add more text and tips on these than I have on the others.

- Dim - Used to define a variable as a certain type
 - i = dim i as integer, r as single
 - You can use the Option Explicit to make sure that VB forces you to declare every variable you use. DIM is that simplest way to declare a variable
- ReDim - Used to change the dimensions of a dynamic array
 - redim arrayname(37)
 - Don't be afraid of this one. You can use ReDim to create an array whose size grows by 1 every time you want to add a number to it. Then, the UBound tells you how many numbers you've added.

-
- Static - Establishes a procedure variable which keeps its value between calls
 - static i as integer
 - For example, if you want to keep track of how many times you've been in a procedure, set a counter as STATIC and increment it by one for each visit to the procedure. It will never go away until the program is terminated.
 - Public - Creates a variable which can be accessed outside its own procedure
 - public i as integer
 - Even if you're the only programmer writing code in your application, use of Private vs Public will help catch errors if you inadvertently try to access an out-of-scope variable
 - Private - Creates a variable that can be read only in its own procedure or module, according to where the declaration took place.
 - private i as integer
 - Use this as often as possible to avoid unnecessary exposure of your variables to coding mistakes.
 - Sub - Defines a procedure which can execute a block of code
 - Sub NewProcedure (var1 as integer, var2 as string)
 - Be sure to check out HELP for how to handle Sub arguments. There are more questions and mistakes made concerning the use of arguments than just about anything else I've seen.
 - Function - Declares a procedure which can return a value
 - Function NewFunction (var1 as integer, var2 as string) as SINGLE
 - This is actually the most versatile of the Sub/Function procedure types. It can do anything a Sub can do as well as returning a value for use in an expression.
 - Call - Transfers control to a Sub or Function (is optional)
 - Call Procedure 1
 - Since the use of CALL is optional, forget you ever saw it
 - CallByName - Executes a method of an object or set/returns a property
 - CallByName(form1,procedurename,vbMethod)
 - The really cool thing about this is that you don't have to hardcode a procedure call. Just use a string variable with the name of the procedure to call.
 - Option Explicit - Instructs VB to force an explicit declaration of all variables
 - Option Explicit
 - You're borderline stupid if you don't use it to catch typing errors. Set up the VB IDE to automatically include this in all projects.

-
- Option Compare - Instructs VB on how to make string comparisons
 - Option Compare Binary
 - This can add case-insensitivity for those times when you don't want to hard-code it
 - Option Private - Prevents a module's content from being referenced outside a project.
 - Option Private Module
 - Generally doesn't apply to most VB applications. If you find a good use for it let me know.
 - Property Get - Declares how to get the value of a property
 - Property Get Name()
 - You won't use this much until you get into creating classes of your own
 - Property Let - Declares how to assign a value to a property
 - Property Let Name()
 - You won't use this much until you get into creating classes of your own
 - Property Set - Declares how to set a variable reference to an object
 -
 - You won't use this much until you get into creating classes of your own
 - Set - Assigns an object reference to a variable
 - Set X = form1.txtInputFromUser
 - Very useful for making code more readable or simply to cut down on how much typing you have to do!
 - Let - Precedes assignment of a value to a variable
 - Let i = 3
 - It's optional, no one uses, so forget you ever saw it
 - Type...End Type - Creates a user defined part type which consists of standard VB data types
 - type anytypename
 - one as string
 - two as integer
 - three as boolean
 - End Type

-
- This is a really excellent way to keep several kinds of data under one variable name. Plus, you can PUT or GET a user-defined type with a single line of code.
 - Const - Creates a variable whose value is fixed
 - const anyname
 - Basically, use this to give easy to remember names to values. For example, suppose you use the value 37.2 a lot in your code, then if you put `CONST MyAge = 37.2` in your code you'll be able to insert the `MyAge` where the 37.2 should have gone. Easier to type and easier to read. Also, you can change the value of the constant by changing only the declaration line of code, rather than searching out every place the value was used!
 - Declare - Used to define a procedure that exists in another file
 - declare functionname (arg1 as integer, arg2 as string) as integer
 -
 - ArrayName = Array (10, 20, 30)
 - Implements - Specifies a class to be implemented in a module
 -
 - Friend - Allows procedure to be callable from modules outside the class
 -
 - GetObject - Return a reference to an ActiveX component
 -
 - CreateObject - Creates and returns a reference to an ActiveX object
 -
 - GetAutoServerSettings - Returns information about the state of an ActiveX component's registration.
 -
 - Enum - Declares a type for an enumeration
 -
 - Event - Declares a user-defined event
 -
 - TypeName - Returns the type of data in a variable
 -
 - VarType - Returns the type of data in a variable
 -

-
- DefType - Sets the default data type of variables
 - DefInt A-Z
 - IS - A variety of data type or status checking options
 - IsArray, IsBindable, IsBroken, IsDate, IsDirty, IsEmpty, IsError, IsMissing, IsNull, IsNumber, IsObject, IsReady, IsRootFolder

Date/Time

These functions are pretty self-explanatory so I've not added any extra comments to them.

- Date - Gets the current date
- Time - Gets the current time
- Now - Gets the current date and time
- Timer - Returns the number of seconds since midnight
- DateAdd - Adds a time interval to a date
- DateDiff - Returns how many time intervals there are between two dates
- DateSerial - Returns the month/day/year
- DateValue - Returns the date
- Year - Returns the current year
- Month - Returns the current month (integer)
- MonthName - Returns the text of the name of a month
- Day - Returns the current day
- Hour - Returns the current hour
- Minute - Returns the current minute
- Second - Returns the current second
- TimeSerial - Returns a date with the hour/minute/second
- TimeValue - Returns the time
- WeekDay - Returns the current day of the week (integer)
- WeekDayName - Returns the text of a day of the week

Miscellaneous

In this list you'll find some of the features of VB about which I get a lot of email questions! The MsgBox is easily the **most used** of the bunch. It handles all of the "Y/N" queries to your user so get to know it well. Also, the **DoEvents, Shell, and Command** functions are indispensable in certain occasions so make sure you know when they should be used.

- MsgBox - A built-in dialog box that gives a message and allows a user input

-
- i = msgbox "Read this!", vbokonly, "Test Message"
 - DoEvents - Allows VB to complete pending tasks
 - doevents
 - Shell - Executes a 2nd program from within the current program
 - shell "notepad.exe"
 - Note - VB does not wait for the Shell'd program to quit before executing the next line of code!
 - Command - Gives any text that followed a VB .EXE execution command
 - temp\$ = command
 - Environ - Returns the system environmental space content
 - temp\$ = environ
 - Beep - Makes the computer beep once.
 - beep
 - InputBox - A built-in dialog box that allows entry of a text string
 - inputbox "Input a value!", 5
 - AddressOf - Provides an entry point for an external program to use a procedure
 - AddressOf (procedurename)
 - AppActivate - Activates an applications window
 - AppActivate (windowtitle)
 - RaiseEvent - Fires an event declared at module level
 - RaiseEvent ProcedureName
 - Load - Load an object
 - load form1
 - Unload - Unload an object
 - Unload form1
 - LoadPicture - Load a picture into a control property
 - form1.picture = loadpicture (filename)
 - SavePicture - Save a picture to a file
 - SavePicture(form1.picture,filename)
 - LoadResData - Load the data from a resource file
 - LoadResData(index,format)

-
- LoadResString - Load a string from a resource file
 - LoadResString(index,format)
 - SendKeys - Send keys to another app as though they were from the keyboard
 - Sendkeys {DOWN}
 - QBColor - Returns a value corresponding to the original QB values 0-15
 - form1.backcolor = QBcolor (12)
 - RGB - Returns a color value by inputting the red, green, and blue parts
 - form1.backcolor = RGB (12,128,256)
 - Me - Refers to the current object, usually the active form
 - print Me.caption

Registry

I've never quite understood why Microsoft got away from the use of an INI file. The ability to use a simple text editor to resolve problems with a program's settings was a key feature about INI files. Also, no matter how Windows crashed, the INI file was protected. Whining aside, VB has made it incredibly easy to access values in the registry. The following VB functions are simple to use and there's hardly any excuse for not taking advantage of them. One thing to remember is that the registry save **strings** so if you're saving or reading numeric information then may have to do some string manipulation with the results.

- GetSetting - Get a value from the Registry
 - temp\$ = getsetting "TestApp", "SectionName", "KeyName", "defaultvalue"
- GetAllSettings -Returns a list of key settings and their values
 - GetAllSettings(appname,section)
- SaveSetting - Save a value into the Registry
 - savesetting "TestApp", SectionName, KeyData
- DeleteSetting - Deletes an entry from the registry
 - deletesetting "TestApp", "SectionName", "Keyname"

Loops and Conditional Decisions

While the event-driven model of VB has taken out a lot of the need for controlling the flow of your application, don't think for a second that you can get by without being an expert on these features of VB. Virtually every single procedure you'll ever write will have one or more of these in it. The concepts are simple, so take the time to become a master of each one! The **For...Next** and the **Select Case** statements are the two most used, so concentrate on them first.

- If..Then..Else - Performs code based on the results of a test
 - If A>5 Then Print "A is a bit number!"

-
- For...Next - Loops a specified number of times
 - For i = 1 to 5: print #1, i: next i
 - For Each ... Next - Walks through a collection
 - For Each X in Form1.controls: Next X
 - While...Wend - Loops until an event is false
 - while i < 5: i = i + 1: wend
 - Select Case - Takes an action based on a value of a parameter
 - select case i
 - case 1 : print "it was a 1"
 - case 2 : print "it was a 2"
 - end select
 - Do...Loop - Loops until conditions are met
 - do while i < 5 : i = i + 1 : loop
 - IIF - Returns 1 of two parts, depending on the value of an expression
 - result = IIF (testexpression, truepart, falsepart)
 - Choose - Selects and returns a value from a list of arguments
 - Choose (index, "answer1", "answer2", "answer3")
 - With - Executes a series of statements on a single object
 - With textbox1
 - .Height = 100
 - .Width = 500
 - End With
 - End - Immediately stops execution of a program
 - End
 - Stop - Pauses execution of a program (can restart without loss of data)
 - Stop
 - Switch - Returns a value associated with the first true expression in a list
 - result = Switch (testvalue1, answer1, testvalue2, answer2)
 - GoTo - Switches execution to a new line in the code
 - GoTo Line1

-
- GoSub ... Return - Switches execution to a new block of code and then returns
 - GoSub Line1
 - On .. GoSub - Branch to a specific line of code then return at the next Return statement
 - On Number GoSub Line1, Line2, Line3
 - On .. GoTo - Branch to a specific line of code
 - On Number GoTo Line1, Line2, Line3

Special Values

There are some keywords in VB which take on special meaning. Their use can be confusing at times, but you'll get used to the terminology as your programming experience grows.

- True - A logical (Boolean) expression. In VB, its value is -1
 - X = TRUE
- False - A logical (Boolean) expression. In VB, its value is 0
 - X = FALSE
- Nothing - Disassociates an object variable from an actual object
 - Set X = Nothing
- Null - Indicates that a variable has no valid data
 - X = Null
- Empty - Indicates that a variable has not yet been initialized
 - X = Empty

Error Handling

Try as I might, I cannot create error free code! So, I turn to these VB features to help me figure out what went wrong.

- On Error - Enables an error-handling routine
 - On Error GoTo Line2 (if error occurs, go to line2)
 - On Error Resume Next (if error occurs, continue executing next line of code)
 - On Error Goto 0 (disables error handling)
- Resume - Used to resume execution after a error-handling routine is finished
 - Resume
 - Resume Next
 - Resume Line1

-
- CVerErr - Returns an error type variable containing a user-specific error number
 - X = CVerError(13)
 - Error - Simulates the occurrence of an error
 - Error 23

Financial Calculations

For those folks who want to use VB for performing routine investment calculations, VB provides a variety of functions. Personally, I use them very infrequently, but I suspect they are used regularly by a lot of programmers. I've never gotten a single question in the mail about these functions!

- DDB - Returns the depreciation of an asset for a specific time period
- FV - Returns the future value of an annuity
- IPmt - Returns the interest payment of an investment
- IRR - Returns the internal rate of return on a cash flow
- MIRR - Returns a modified internal rate of return on a cash flow
- NPer - Returns a number of periods for an annuity
- NPV - Returns a present value of an investment
- PPmt - Returns the principal payment of an annuity
- PV - Returns the present value of an annuity
- Rate - Returns the interest rate per period for an annuity
- SLN - Returns the straight-line depreciation of an asset
- SYD - Returns the sum-of-years' digits depreciation of an asset

Command Summary

Fact is, that few of the VB commands stand alone. More often than not, a command supports only one aspect of a topic. For example, the OPEN command opens a file so you can read it, whereas the LINE INPUT command reads one line of data at a time. And so it is in all of the areas of capability which VB supports. This page provides my own interpretation of VB command grouping. The benefit to you is that when you have a task to do you can quickly see which commands to consider for the task!

Don't confuse VB commands with control properties/events/methods! In this section I cover only those VB language elements which **are not** associated with specific VB objects.

----- Strings -----			----- Math -----			----- Logic -----		----- Arrays -----	
Left	Mid	String	Hex	Rnd	Sqr	Mod		Option	Base
Right	Chr	InstrRev	Oct	Sgn	Abs	And		Erase	
Trim	Len	Asc	Val	Sin	Atn	Or		Dim	
LTrim	LSet	Space	Round	Cos	Log	Xor		ReDim	
RTrim	RSet	Replace	Int	Tan		Not		UBound	
UCase	Format	StrComp	Fix	Randomize		Eqv		LBound	
LCase	Instr	StrReverse		Partition		Imp		Filter	

FormatCurrency	StrConv	Round	Like	Array
FormatDateTime		CInt...	Is	Join
FormatNumber				
FormatPerCent				

Files / Folders			Binary	Sequential	Error Handling
Dir	FileDateTime	FreeFile	Get	Line Input	On Error
ChDir	FileLen	Open	Put	Write	Resume
MkDir	FileCopy	Close	Seek	Print	Resume Next
ChDrive	Lock	Reset	Input	Spc	Resume LineX
CurDir	UnLock	Lof	Loc	Tab	CVErr
Name	GetAttr	Eof			Error
Kill	SetAttr	Width			
RmDir	FileAttr				

Declarations			Date/Time	
Sub	Call	Set	Date	MonthName
Function	CallByName	Let	Time	WeekDayName
Dim	PropertyGet	Event	Now	Day
ReDim	PropertySet	ArrayName	Timer	Hour
Static	PropertyLet	Implements	DateAdd	Minute
Public	Option Explicit	Friend	DateDiff	Second
Private	Option Private	Enum	DateSerial	TimeSerial
Const	Option Compare	TypeName	DateValue	TimeValue
Declare	Type...EndType	VarType	Year	WeekDay
Is	GetObject	DefType...	Month	
	CreateObject			
	GetAutoServerSettings			

Miscellaneous			Registry	Values
MsgBox	Beep	Load	GetSetting	True
DoEvents	InputBox	UnLoad	SaveSetting	False
Shell	AddressOf	SavePicture	DeleteSetting	Empty
Command	RaiseEvent	LoadPicture	GetAllSettings	Nothing
Environ	Load	LoadResData		Null
RGB	Me	LoadResString		
QBColor	SendKeys	AppActivate		

LOOPS AND CONDITIONAL			
IIF	For i=1 to j Step k	If_Then_Else	Select Case
End	-		Case
Stop	Next i	If_Then	Case IS cond
Switch		ElseIf_Then	Case _ TO _
GoTo...	While cond	Else	Case Else
GoSub...Return	-	EndIF	End Select
On...GoSub	Wend		
On...GoTo			
Choose	Do While Until cond	For Each	
With	-	-	
Exit Function	Loop While Until cond	Next	
Exit Sub			
Exit Do			

File Handling

The usual model for all computer programs is that a user inputs data and the computer saves it. Okay, there's a lot more than that to a computer program, but the point is that virtually every program is written to save data that is entered or data that is calculated by the software itself. If you want to be an expert VB programmer then you'll need to know a lot about how VB handles files - what is possible and what is not, plus how to do something in the smallest code or in the least amount of time. To do this you need to understand the file handling features which VB offers.

Sequential (Text) Files

In VB, Microsoft has separated the file handling capabilities from the database handling features. With databases, the file manipulation is pretty much transparent to the user. However, in non-database applications the programmer has to handle virtually all aspects of reading or editing the data contained in a file.

Because of their universally standard format, simple text files (also called sequential files) are often used as the storage method for information. The old DOS Edit program and the newer windows NotePad programs both create simple text files.

To simplify the handling of text files, Microsoft is working on a new set of features that will be implemented in an ActiveX object (the FileSystemObject). Until then, file handling will continue as the manual procedure that it is today.

The two statements you **must** use to access a text file are OPEN and CLOSE. Here's a quick example:

```
OPEN "filename" for INPUT as #1
CLOSE #1
```

In this example nothing was done but opening the file and closing it.

Here's a little more useful example:

```
OPEN "filename" FOR INPUT as #1
WHILE NOT Eof(1)
LINE INPUT #1, temp$
WEND
CLOSE #1
```

In this example, the code reads one line at a time, walking through the file until the end-of-file (EOF) is true. The line of data (minus the carriage return / line feed characters) is put into the string variable temp\$. In your own applications you can save each line into an array, or concatenate them into a single string variable, or any other processing you want to do.

This next example shows how the text can be read as a single string variable and put into a textbox control.

```
OPEN "filename" FOR INPUT as #1
WHILE NOT EOF(1)
LINE INPUT #1, temp$
alltext$ = alltext$ & temp$ & vbcrLf
WEND
textbox1.text = alltext$
CLOSE #1
```

Note that I had to add back in the carriage return / line feed to maintain the same line break as was in the original file. If you want the file to be read simply as a long string, replace the vbcrLf with a space (" ") to keep separation between the word at the end of the line and the word which starts the next line.

The examples so far were to read a text file. Here's an example for writing to a text file:

```
OPEN "filename" FOR OUTPUT AS #1
```

```
For i = 1 to 10
```

```
PRINT #1, i
```

```
NEXT i
```

```
CLOSE #1
```

In this case, each PRINT operation goes to a different line in the text file (which means a crlf character is inserted into the byte stream). The PRINT statement provides a variety of options for putting the data into the output file. You can put all the data in one line, separate each piece by one or more characters, or format the numbers as they are written. The bottom line is that whatever is written to the file is **written as text** and when it's read back your program must convert it to numbers as needed.

One more example will be worthwhile. Suppose the data consists of columns, and you next to extract the numbers from those columns. How do you do it? Here's how it could be done for the case of having 3 numbers per line, 1 in each ten columns:

```
OPEN "filename" FOR INPUT AS #1
```

```
WHILE NOT Eof(1)
```

```
LINE INPUT #1, temp$
```

```
Number1 = VAL (mid$(temp$, 1, 10))
```

```
Number2 = VAL (mid$(temp$, 11, 10))
```

```
Number3 = VAL (mid$(temp$, 21, 10))
```

```
WEND
```

```
CLOSE #1
```

Binary (Data) Files

Binary files are created by storing variables (both string and/or numeric) without any added formatting as can be done with sequential file. An integer variable can be written, followed by a string, and followed by another integer. However, the programmer has to keep track of which variable data is placed where!

One of the really great features of binary file access is that you can often avoid using the VB database features. This is important because the distribution files for database access are **HUGE** and I hate having to put them in my application distribution files.

Another especially nice feature of binary files is that you can PUT a complete user-defined variable in a single statement, and recover it just as easily. This greatly simplifies data storage in many cases.

Let's get right into some examples. To put data into a file, use this:

```
OPEN "filename" FOR BINARY AS #1
```

```
PUT #1, var1, var2, var3
```

```
CLOSE #1
```

In this example, all we did was write three variables to the file. To read them back, use this:

```
OPEN "filename" FOR BINARY AS #1
```

```
GET #1, var1, var2, var3
```

```
CLOSE #1
```

You'll notice that we didn't have to format the PUT or GET statements in any way.

Printing

Let's amend the computer program "model" which I gave in the tutorial section on file handling. A computer program basically consists of a user who enters data, a file in which the data (or calculated results) are stored, and a paper copy of the data. The printed copy is usually formatted in such a way as to allow the user to make sense of the data. The point of this short essay is that most users expect to be able to get a hard copy (meaning a paper copy) of their data. They want it stored, but they also want to be able to take the results away from the computer for subsequent review. Hence - printing! An important fact is that how well the data is presented to the user can

make or break the success of the application. Learning to use the printing tools of VB is critical to your success as a programmer.

Introduction

For those of you who remember BASIC printing, the news with Visual Basic is both (mostly) good and (not much) bad. With BASIC and before the advent of the proportional fonts of Windows it was easier to figure out where your print statements were going to put the output. Since every character took up the same width, you only had to count positions (80 per line) as part of figuring out how to code print statements.

The bad news with DOS BASIC was that you had to print one line at a time. You couldn't print line 40, then go back to line 10. The really good news is that in Visual Basic, an object known as the printer is available to you. The printer is like a work space to which you can direct the output of print statements or the output of graphics statements. You can send text/graphics output to the printer object in any order you want, then when you're done you can tell Visual Basic to print the material to a physical printer.

The other good news about Visual Basic is that you can also "print" inside an applications window (a form or a picture control) and Windows will handle the movement of the printed output as you move windows around or as you move other windows in front of the printed material. With DOS BASIC you had to handle all of that yourself!

The (almost) bad news with Visual Basic printing is really a Windows issue. Although there are non-proportional fonts (same width for each letter) in Windows, they really don't look professional. So most folks program the print statements to use proportional (different width for each letter, to enhance to display of the text) fonts. The downside to this is that since each character is not the same width, you have to specify the X/Y coordinates of the print location. VB has some tools to make it a little easier than I make it sound, and the conveniences of the printer object really do outweigh the inconvenience of handling proportional fonts.

Printing Surfaces

Ok, I've mentioned that in VB you can "print" to more than just a physical printer. In fact, you can print to a form, to a picture control, to the DEBUG object (more on this in another section) and also to any physical printer connected to your PC or to any shared network printer to which your PC is connected.

Here's one of your first opportunities to see methods at work. Do you remember we described methods as being commands within VB which act on a control? Well, printing (by which I mean the output of text as well as the output of graphics) is a method that you use with an object. Here are three examples:

```
form1.print "test"  
picture1.print "test"  
printer.print "test"
```

All three of these do the same basic thing, which is to print the word "test" to an object. The first two print on-screen to a form (form1) and to a picture control (picture1). The last prints to the printer object that I mentioned earlier.

In every case where a print method is used, the printing takes place at the current X/Y position of the object. There are several ways by which VB determines the current X/Y position of an object (this isn't the same as the .Top or .Left properties of an object, which determine the positioning of a window on your screen). However, you can always override VB by using code to set the current X/Y position. Here's the code for it:

```
form1.currentx = 25  
picture1.currenty = 200  
printer.currentx = 844
```

Notice that there are two properties .CurrentX and .CurrentY. You can set them using the statements I just gave, or you can use them to return the current X/Y position as follows:

```
i = form1.currentX
```

```
j = form1.currentY
```

One complicating factor in VB that you'll just have to get used to is the concept of an object's scale. You can set the scale of an object to pixels, twips, inches, centimeters, points, character, and millimeters. The default value is twips (1440 twips is an inch). Depending on the application, you can make a big difference in the code if you select a scale to match the type of application you're writing.

For example, when you're doing graphics work it may be much better to deal in pixels. In general, however, I've found that only the pixel, twips, and inches scales are necessary to meet my own programming needs.

One final comment about the difference in printing on forms and picture controls versus printer on the printer. With forms/controls, the printing takes place immediately. With the printer, none of the information is sent to the printer until you use the .EndDoc method, which tells VB you're done printing and to send all of the information on the the physical printer.

I go into more detail about printing graphics in the next section of the tutorial.

Printing Tips

Now that you have some general ideas of how printing works, lets do some examples. I've picked some of the examples because they provide tips on how to do some of the most typical printing tasks you're likely to need.

Print at a specific location

To print at a specific location, such as 1 inch down and 4 inches over, you must set the currentx and currenty properties accordingly. If you'll first set the scalemode to inches you won't have to do any conversions.

```
printer.scalemode = vbInches
```

```
printer.currentx = 4.0
```

```
printer.currenty = 1.0
```

Print multiple lines

Each subsequent time you execute a print statement, VB resets the X/Y coordinates to a position corresponding to the "next line down". In other words, it moves the current position to the far left and one line down. You can override this if you choose, but here's a simple printing of four lines one after the other:

```
printer.print "line 1"
```

```
printer.print "line 2"
```

```
printer.print "line 3"
```

```
printer.print "line 4"
```

Each line will print at the left side of the printer, and the printout will be four lines deep.

Print in the middle of a line

To printer in the middle of a line, you'll need to figure out the width of the text you're about to print, then begin printing 1/2 that difference between that width and the width of the printer object. This involves the use of the method textwidth, which gives the length (in the current scalemode) of a text string.

```
printer.currentx = (printer.width - printer.textwidth("print string")) / 2 printer.print "print string"
```

Print in the middle of a page (top to bottom)

This case is just like the one before, except you also position halfway between the top and bottom of the page an make use of the textheight method.

```
printer.currentx = (printer.width - printer.textwidth("print string")) / 2
```

```
printer.currenty = (printer.height - printer.textheight("print string")) / 2
```

```
printer.print "print string"
```

Print in columns

If you want to print starting in a column, then reposition currentx for each item to be printed.

This example prints at 1, 3, and 5 inches.

```
printer.scalemode = vbInches  
printer.currentx = 1  
printer.print "column1 data";  
printer.currentx = 3  
printer.print "column2 data";  
printer.currentx = 5
```

Notice that the print statement ends with a ";", which directs VB to leave the currentx/currenty values positioned at the location following the current print operation.

Print as right justified

Left justification is the normal method automatically happens with justification taking place at the current value of .CurrentX. For right justification, you want to start printing at a position so that the length of the text will result in an end-of-print action at justification position.

```
printer.currentx = rightjustificationposition - printer.textwidth("print text")  
printer.print "print text"
```

Summary Hopefully that examples give you an idea of the basic concepts of printing. Head over to my **control summary chart** - available in [Excel 97](#) and [Excel 5.0](#) formats - to see a complete listing of the properties and methods available to you.

Finally, just in case you don't go looking right away, please remember that you **can set the orientation** of the printer with the .orientation property. This lets you switch between landscape and portrait very easily within your code.

Distribution of Applications

There are programmers out there who program solely for themselves. They write code/applications and no one but them ever sees their applications. However, most of you are like me and we want to revel in the accolades that our friends and peers heap on us when we provide them with exactly the right program to meet a need that they have expressed! Ah.. but how do you give them the program? One of VB's weakest features has always been it's ability to create an installation program that allows you to distribute your great application to your users. This section introduces the distribution aspect of programming.

Introduction

If you want to run your VB program, you have two options. You can open the VB project within the VB IDE and run it there. Or, you can compile the project into a .EXE file and run that. To give (or sell) your program to other people takes more effort. You cannot give them a copy of VB6 (it's illegal). If they already have VB6 (or the same version as you have), then you can simply give them the .EXE file and they can run the .EXE file just fine.

Usually, however, you cannot depend on your users having a copy of VB on their machines. That being the case, you not only have to give them a copy of the .EXE file, but also a copy of any VB required files which they don't have on their machine. Microsoft allows you to distribute any such files as are required by the user to run your distributed .EXE program. Note that the distributed files **can only** be used in the run time program. Adding a distribution will not necessarily allow a user who has VB on his PC to use those files (OCXs) in the IDE. More on that later.

The fact that a VB program requires the presence of other files to operate correctly is really bad news because those extra files can be very large. As a worst case example, I've written a VB VRML application called [WinPlace](#) whose distribution file measures almost 10MB! That's a lot of diskettes, or else a lot of download time over a modem. Many potential users of WinPlace won't try it simply because it will take too long to download the installation files!

Every VB program requires the use of a 1.4MB file, msvbvm60.dll. That means that no matter how small the .EXE file is that you create, that your minimum size for distribution files is 1.4M. Often, a VB .EXE file is only a few hundred KBytes (100K - 400K), but at those sizes you're automatically going to require at least two distribution diskettes.

Another "gotcha" that awaits within VB is that the addition of a single feature can cause an easy multi-megabyte increase in the size of your distribution files. For example, add in any database features to a VB program and you'll see the distribution file list go **way up!**. It's hard to tell ahead of time how big your distribution file size is, so you pretty much have to create the distribution files to know for sure.

In this section of the tutorial I'm not going to walk through an actual setup file creation. The capability in VB is known as the "Package and Deployment Wizard" and it's reasonably straightforward to use, although you have very little control over the details of the installation procedure.

Third Party Help

I use the Package and Deployment Wizard for some of my applications, but I also use the InstallShield software as well as the Wise Installation System. The latter two are shrink-wrap programs which are generally much more powerful than the built-in capabilities of VB. However, these also each cost several hundred dollars. For your applications which mostly go to friends, or to a limited set of users, don't hesitate to use the built-in VB Packaging and Deployment Wizard. If you want a wider range of options on how to control the installation process, check out one of the two above.

Menus

Everyone knows what a menu is, the question is how can you make them in VB? Well, it turns out to be very simple. VB has a built-in menu editor that you will use and it's pretty much a no-brainer. The only time menus should give you any trouble is when you want to do some of the more sophisticated tasks such as adding menus on the fly or such as providing popup menus. Considering that virtually every application I've ever written makes use of menus, I was glad to see that Microsoft made it so easy to do.

Menus Are Controls!

You already have been exposed to menus. The most well known example being the File/Open menus used by most Windows programs for opening a file. What you may not know is that each of the menu selections File and Open are independent controls which VB can create for you using the menu editor. We'll get to that in just a second.

The concept that a menu selection is really a control is a very valuable piece of information because as you will see, the menu controls support properties and events, just like any other controls you're used to seeing. What this means is that you can use the experience you have in using controls to help understand menu operation.

What menus don't support are methods. If you didn't read through my control summary spreadsheet you might not have noticed it, but controls **may or may not** implement all three of the categories - events, methods, and properties. There's no rule that says you have to support them all, and in some cases it's just not necessary. Menu controls are like that. You'll see that properties and events are all you need.

Before we get into the menu editor, in which you can set properties of a menu control, let's look at the complete list of menu control properties. It's not very long:

- Caption - the visible text you see
- Name - the name used in code to refer to the menu control
- Checked - determines whether a small check mark is displayed to the left of the menu control
- Visible - determines whether a menu control can be seen
- Enabled - if FALSE, the text is grayed out and cannot be selected

- Index - used only when the menu control is part of a control array
- Shortcut - a key sequence that will invoke the menu
- HelpContextID - determines which topic in the help file will be displayed
- NegotiatePosition - works in conjunction with OLE embedment to determine which menu controls are displayed
- WindowList - determines whether a menu control maintains a list of the current MDI child windows

The event list for Menus controls is very short because Menus support only one event, the click event:

- Click - occurs when a menu item is selected

The bottom line for using menus is that you use the Menu Editor to create the menu structure and to set the properties for each menu. Then you **add code to each Click event** to perform whatever function you choose in response to a user selection of the menu items.

Menu Editor

Here's the Menu Editor window:



As you can see, the editor has two general sections. In the top half, you set the properties (you saw a complete list of the properties above). In the bottom half you create the hierarchical structure of the menu.

Remember that menus are only associated with a form. No other control has a menu. VB provides the built-in ability to manage the display of all of the menu items. You only have to create the structure and let VB handle it from that point on.

Also note that each and every form in your application can have its own menu. The menus can be identical (same properties) or they may be different. It's entirely up to you.

Now, let's talk about each of the properties and see if there's some guidance on what to use for the properties.

- **Caption**
Simply use the shortest name you can. Users hate long captions because they take up too much space on the screen and reading them slows down using the menu. Also, try to use

a caption that doesn't have the same first letter as any other menu caption. This will allow you to use the first letter of the control caption as the shortcut - it makes it much easier for the user to remember!

- **Name**

While it can be anything, remember that the menu event will bear this name. I always use a name that starts with mnu so that I can recognize the event procedure as belonging to a menu control. For example, the File menu caption usually is associated with a menu name of mnuFile.

- **Checked**

Menu items are either checked, or not. You can check it from within the menu editor or by using code. For example, this code will cause a menu item to be displayed with a small checkmark to its left:

```
mnuFileOpen.checked = TRUE
```

- **Visible**

If you want to prevent a user from having access to a menu item, simply set the visible property to FALSE. This will keep the user from even knowing that the menu item ever existed.

```
mnuFileOpen.visible = FALSE
```

- **Enabled**

To allow the user to see the menu, but not to select it, set the enabled property to FALSE:

```
mnuFileOpen.enabled = FALSE
```

- **Index**

If you give a menu (or any control) a non-zero index value, that makes it a member of a control array. Control arrays are covered elsewhere in the tutorial, but the bottom line is that you can use the Load function to add new menu items at run time, but **only** if there is a menu control array created during design time. This isn't something you'll do a lot of, but at least be aware that it is possible. On the other hand, there are lots of times where non-menu control arrays are very useful, so keep the idea of control arrays in your mind and we'll talk about how to apply them elsewhere in the tutorial.

- **Shortcut**

Most users want to be able to invoke a menu item from the keyboard. This is the property that defines the shortcut key strokes. When a shortcut is defined, you can invoke the menu item from the keyboard, **no matter how deep** in the menu structure the item is that you are calling.

- **HelpContextID**

Like any control, this allows you to refer to a specific topic in the application's HELP file. Pressing F1 while the menu item is highlighted will call up this topic. This is a useful technique, but mostly applies to HELP files you've written yourself. It speeds up how quickly a user can drill through the HELP file to get to information about a menu item. In my own applications, I seldom use this feature. Lazy perhaps, but it's not that terrific a feature to warrant maintaining the synchronization between an application and a continuously changing HELP file.

- **NegotiatePosition**

When you embed an OLE object which has its own menus you have to give instructions

on how to blend the form's menu and that of the embedded object. Because I don't depend on my users having specific OLE-enabled applications (such as Word, Excel, or others) I rarely use this feature. When you need it, it's great, I just don't expect most programmers to need it.

- **WindowList**

Only for use in MDI applications, where you have multiple MDI windows open at one time. This is another feature which is really great if you need it, but in my own applications I rarely need it. In this case, even though I don't use it much myself, you might want to give MDI windows a second look because of the convenience they offer to users. Microsoft changed version 6.0 of VB to default to MDI windows in its IDE because of user inputs which supported the change. I still set my own IDE to the single window interface but you may find your own tastes running differently.

Other Menu Tips

To end this section of the tutorial, here's a few areas where special handling of menus might make your program more user friendly.

Previously Opened Files

You've seen applications which provide a list under the File menu, of files which have been previously opened. There are two basic ways to do this, either by dynamically adding controls (remember my comment about control arrays?) or by creating the list during design and simply setting the caption and visible properties to reflect the most recently used file names.

When I write applications, I use Registry Settings to store the most recent file names and then set caption/visible properties of 4 or 5 pre-built menu items.

Feature-Activated Menu Items

One of the user-friendly techniques you should put in your applications is the practice of setting the visible property of menus to FALSE if the user is not allowed to use the menu. For example, if a program allows a user to make a selection of objects in the window then you might make visible a menu item to delete the selected items. However, if there are no items selected, then you should set the visible property of the delete menu item to FALSE because the user has nothing to delete.

The alternative is to set the enabled property of the unusable menu items to FALSE. The user can still see the menu item but it will be grayed out and will not work correctly.

Unless you have specific reasons to keep it visible, my preference is to set the visible properties to FALSE so as not to clutter up a user's screen.

PopUp Menus

You should use this capability of VB as much as possible. PopUp menus are a means of providing a user with context sensitive menus - which are menus that provide only the options which are likely to be used according to where on the screen the mouse is located. Typically, the popup menu is activated by using the right mouse.

First of all, any menu that is to be a PopUp menu must not be visible. Once it's visible setting is set to FALSE you can use it as a PopUp menu. Plus, you can have multiple PopUp menus on the same form.

Secondly, the invisible menu must have sub-menus. You call the invisible menu and it's visible sub-menus are displayed.

Here's the basic code for a PopUp menu. This example assumes that you have a menu called "mnuTestPopUp" whose visible property is set to FALSE. Put this code in the MouseUp event of a form (form1 in this example).

```
If Button = 2 then  
    form1.PopupMenu "mnuTestPopUp"  
End If
```

You can put this code in the MouseUp event of any control, but the line of code must read "form1.PopupMenu" because only forms have menus.

HELP Menu

When you create a program, the IDE allows you to define the .hlp file which is displayed by WinHelp when you press F1. But be sure to include a Help menu on the top of your form. Users will expect it. In that menu, you should include at least two selections - "Contents" and "About". The contents should go to the contents section of the help file, but the about selection is normally used to call up a small window which gives version and author information about the program. Also, put the Help menu at the far right of the menu selections. Your users are used to seeing it there and they will expect to find it in the same place as all other Windows programs that they use.

You'll notice that I have used the phrase "users will expect it" several times. Remember that as a programmer you want users to feel comfortable using your program and you don't want to force them to use a non-standard interface unless there is a clear advantage to doing so. Bottom line is that for your users' sakes, you should stick with Windows standards whenever you can!

So, there you have it. If you have any trouble with menus or need some more examples, the VB HELP file has some pretty good code examples to help you along.

Common Dialogs

The whole point of a visual language like VB is that it makes certain parts of programming (usually the user interface) much easier on the programmer. The idea is to build in features which programmers use over and over again. In the case of common dialogs I'm happy to say that Microsoft has really done us a good turn. I've written my own dialog windows (windows which provide information or request inputs from users) and I've used the built-in common dialog windows that VB provides. It's a no-brainer -- use the common dialog unless there's no other way!

Common Dialogs Mean Less Code!

In my beginner's section I state that a programmer's job is to get to the customer results as quickly and efficiently as possible. Common Dialogs are a great way to abide by this strategy. VB provides a single .OCX file which contains the code for all 5 of the windows known as the common dialogs. Without further ado, here are the five:

- **Open**
- **Save As**
- **Color**
- **Font**
- **Print**

Before we get into using each one, remember that the common dialog windows **do not do anything automatically**. All they do is provide a way of asking your user some questions and returning the answers to your application. You must write the code to use the answers!

For example, if your application displays the Open common dialog and your user selects a file name to open, absolutely nothing will happen. You have to write the code that performs the open functions. The Open common dialog box only provides an easy way for you to ask the user which file he wants to open.

You could write a completely custom version of the Open common dialog, customizing it to your specific needs. However, this can take considerable time and introduces code which you must maintain. It's far better to use the proven common dialog window than to invent something new every time!

Calling Up the Common Dialog Windows

To use the common dialogs you must first add the Common Dialog OCX to your toolbox. This is

done by right-mouse clicking on the toolbox, selecting Components, and then checking the Microsoft Common Dialog Control 6.0. Select OK for the selection to take effect.

The code for displaying a common dialog box couldn't be simpler. Simply use code to invoke the appropriate Method of the common dialog object as follows:

- CommonDialog1.ShowOpen

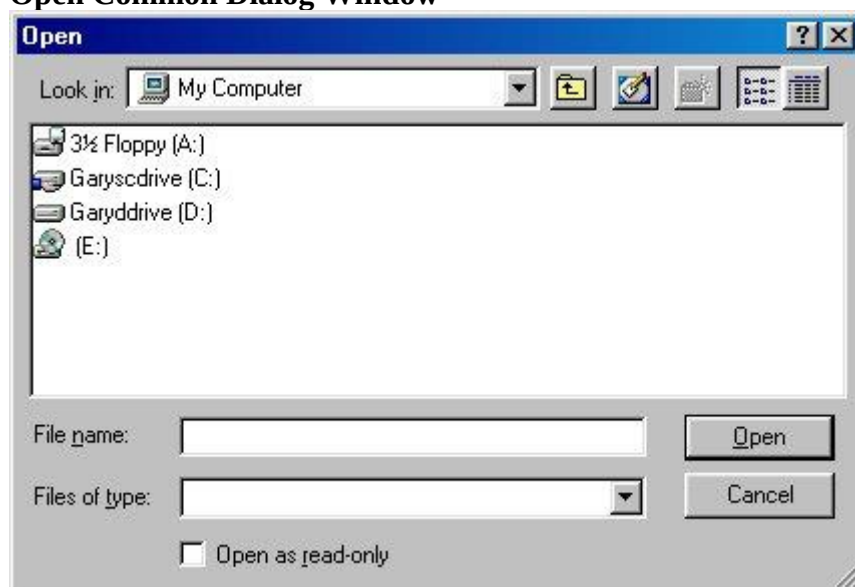
The common dialog object also supports the .ShowSave, the .ShowColor, the .ShowFont and the .ShowPrinter methods which cause the display of the corresponding common dialog window.

A few other points are in order. If you look at my Control Summary Chart you'll see that the common dialog control has properties which you can set in code **before** you display the dialog windows. In this tutorial I don't discuss all of the possible settings. However both the VB HELP files and the VB 6.0 Programmer's Guide provide sample code.

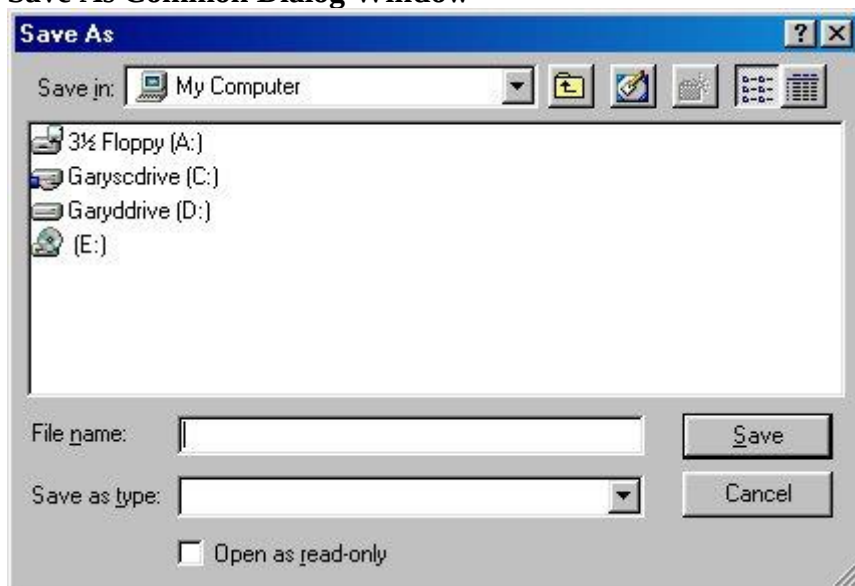
Be sure to take note of the fact that by default, when a user selects "Cancel" in a common dialog window that a program error will occur. If you don't override this by setting the .CancelError property, you'll have to include error detection code in the procedure which invokes the display of the common dialog window.

Also take note that the common dialog control allows you to control the display of various aspects of the window by setting the .Flags property. You really should read through the HELP file's discussion of all of the options you can control using the .Flags property. There's a lot of power in that one property and you can add some powerful customization of your dialog windows if you use them fully.

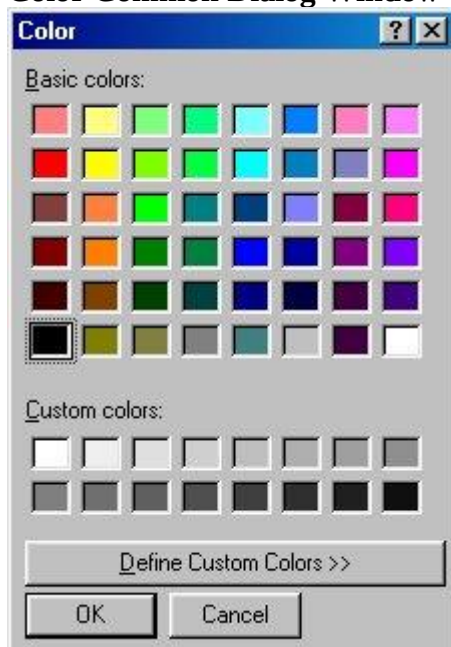
Open Common Dialog Window



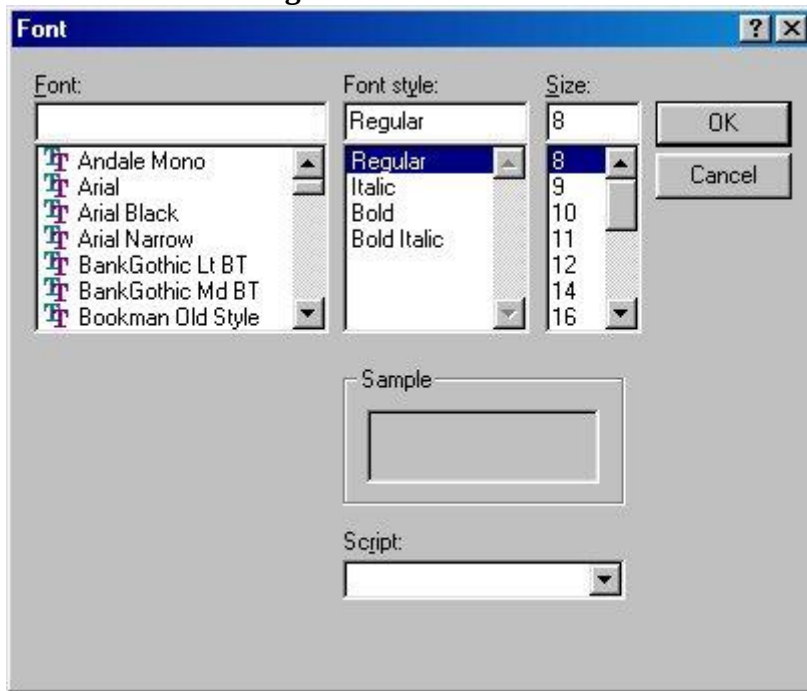
Save As Common Dialog Window



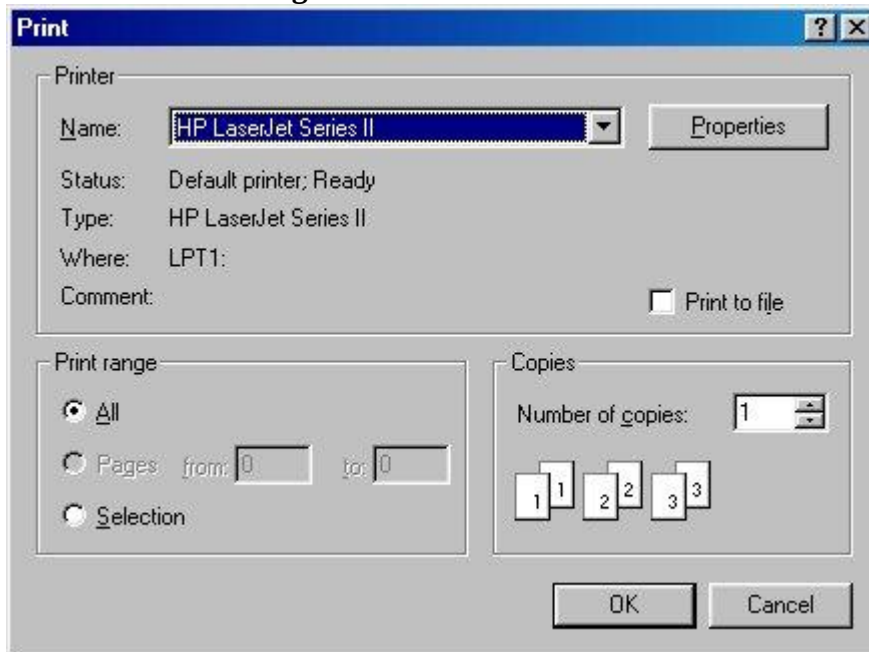
Color Common Dialog Window



Font Common Dialog Window



Print Common Dialog Window



And last but not least, a common error seen when using the Font dialog window is due to not setting the .Flags property. That property must be set (check HELP for allowed settings) before the Fonts dialog window will display correctly. You'd think Microsoft would have given it a default value, but as it is you must set the property!

Using the Mouse

If you're expecting to have to drain your brain to understand VB's mouse support features, then you're going to be disappointed. Using the mouse within a program is actually very simple and in the next ten minutes you'll have been exposed to most of the things you'll ever need to know!

Mouse Overview

There are actually two parts to being a "mouse" expert. The first part is simply using mouse movements, including button clicks, in your program. You'll find that handling mouse

movements is very simple. Any VB programmer can learn to respond to the movement of the mouse or the click of a mouse button.

The second part is using the built-in drag-and-drop features of VB. I cover drag and drop in the next section of the tutorial.

Mouse Movement and Mouse Button Clicks

Incredible as it may sound, you only need to understand 3 events to become a MouseMaster (ok, it's not as useful a title as an MSCD Engineer, but the title comes free with completion of this tutorial!). The key to this simplicity is that **a mouse is not an object**. It has no properties and it has no methods. The recognition of mouse activity (move/click) is actually built into all of the other controls! If you look at the controls that come with VB (or commercial controls) you'll see the following 3 events:

```
Form_MouseDown (Button As Integer, Shift As Integer, X As Single, Y As Single)
Form_MouseUp (Button As Integer, Shift As Integer, X As Single, Y As Single)
Form_MouseMove (Button As Integer, Shift As Integer, X As Single, Y As Single)
```

It doesn't take much examination to realize that the arguments for each mouse event are exactly the same. This greatly simplifies our discussion here and makes your learning process much simpler!

First, let's make sure we know what is going on inside VB. If you press and release a mouse button, **BOTH** the MouseDown and MouseUp events will be executed, **for the control over which the mouse cursor is positioned**. Make sure you understand that. Mouse events happen for the control under the mouse. This is true even if, after pressing a button, you move the mouse of a control before releasing the button. In that case the MouseUp event still takes place for the control over which the mouse button was pressed.

One thing to note about the mouse events. While the MouseMove event will happen by moving a mouse, it will also happen when you click a mouse button.

Remember also, that clicking a mouse button or moving a mouse **does not automatically make anything happen** inside your program. Unless you write code that executes inside the mouse up/down/move events, your program will ignore the mouse actions.

As you can see in the arguments of the mouse events, there are only 4 variables to understand. Let's tackle each one now.

X and Y

These are the simplest of the arguments and are exactly what you think they are - the position of the mouse **within the control (or the form, if the mouse cursor is not over a control)**. The only "trick" to remember is that X and Y are given in coordinates as defined by the .ScaleMode property of the control or form. Check out the graphics part of my tutorial for information on the ScaleMode property of an object.

The X and Y properties are simply information that you can use in your program. For example, if you were to draw a circle inside a picture control, and wanted to know if the mouse click took place inside the circle, you would examine X and Y to see if it falls within the bound of the circle. What you do with the information is up to the code that you chose to put into the event procedures.

In my experience, the X and Y are typically used to identify where to take an action, such as where to create a graphic.

Button

The Button arguments actually can tell which of the three possible mouse button(s) are pressed. The possible values of Button are 0 through 7. The following code shows how you would use Button to figure out what is being pressed:

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
Select Case Button
    Case 0 'no button is pressed
    Case 1 'only left button is pressed
```

```
Case 2 'only right button is pressed
Case 3 'only left and right buttons are pressed
Case 4 'only middle button is pressed
Case 5 'only left and middle buttons are pressed
Case 6 'only right and middle buttons are pressed
Case 7 'all three buttons are pressed
```

```
End Select
```

```
End Sub
```

If you simply want to know if a particular button is being pressed, and don't care about the state of any other button, then you can use this code:

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single,
Y As Single)
```

```
    If Button AND 1 then Print "Button 1 is pressed"
```

```
    If Button AND 2 then Print "Button 2 is pressed"
```

```
    IF Button AND 3 then Print "Button 3 is pressed"
```

```
End Sub
```

In all my programs I use the press of a single button for my user interface. I can't think of a single instance where I found it necessary to have my user press two buttons at once.

Here's one more point you should know. The MouseMove event **does not** take place for each pixel that the mouse is moved. Instead, the number of events that takes place depends on the operating speed of your PC and how loaded the processor is. The bottom line is that while the frequency with which MouseMove events are generated is reasonably high, you cannot depend on the event to take place any specific number of times, nor at any specific interval of time.

Shift

As I mentioned above, I very rarely require that my user press multiple mouse buttons to accomplish a task. I also never force him to press a key at the same time as he presses a mouse button. However, there are a number of programs which do this (Windows 9x, for example, uses CTRL-Mouse to identify a copy command). VB supports this capability by providing the Shift argument. Like the Button argument, the Shift argument also has 8 values as follows:

0 - Neither the SHIFT, CTL, or ALT keys are pressed

1 - SHIFT key is pressed

2 - CRTL key is pressed

3 - ALT key is pressed

4 - SHIFT and CRTL keys are pressed

5 - SHIFT and ALT keys are pressed

6 - CRTL and ALT keys are pressed

7 - All three keys (SHIFT, CTRL, and ALT) are pressed

A point to remember is that just because you press one of these keys while also pressing a mouse doesn't mean that anything will automatically happen. You must put the code in your events to determine what action to take based on which keys are pressed.

One last point about using the mouse before we tackle drag and drop features. You saw that the Button and Shift arguments take on values from 0 to 7. As I have mentioned before I strongly suggest that you use the built-in constants that VB offers to help make your code more readable.

For example, the code:

```
If Button = vbLeftButton
```

is more understandable than

```
If Button = 1
```

For testing the Button argument, use the vbLeftButton, vbRightButton, and vbMiddleButton constants.

Drag & Drop

You see it in just about every Windows application, so how can you do it in your VB application? Fortunately VB provides built-in tools to make it easy for you. While the code you write to respond to a drag and drop operation can be fairly complex, the mechanics of a drag and drop operation are fairly simple and logical. VB especially handles well the graphical tasks of a

Drag and Drop

Ok, a short description first. Put the mouse cursor over an object on a form, then press and hold the left mouse button. Without releasing the button, move the mouse cursor to a new location. Then release the mouse button.

Was that fun? Either way, what you just did was a drag and drop operation. How your program responded to what you did was entirely up to the programmer of the application. In programs which support drag and drop operations, the cursor changes to an icon (indicating a drag operation is taking place) and when the mouse is released the icon reverts back to a normal cursor (indicating that the drop operation is now over).

VB offers both an automated and a manual way to allow users to perform a drag and drop operation. In either case, just like with common dialog windows, a drag and drop operation is just a method of having a user give you instructions/data, which you must use in code before anything happens.

The visible, outward sign of a drag and drop operation is the appearance and then disappearance of an icon during the operation. However, if you don't have code in the appropriate event, absolutely nothing else will take place in your application.

Drag and Drop Events

By way of clarification, a drag and drop event takes places between two objects. At one object a drag is initiated and at the second object a drop takes place.

When the drag begins, the first object's does not experience an event. VB silently handles the creation and movement of the drag icon. When the mouse is released, signalling a drop operation, the DragDrop event of the receiving control is fired and any code in that event is executed. If the DragDrop event has no code, then nothing else happens.

A second event is possible, which is the DragOver event. After a drag is initiated and at the moment the drag icon is moved over another control, that control's DragOver event is fired. It's only when the mouse button is released that the DragDrop event takes place on the receiving control.

The dragdrop can be very useful, users know how to use it and users see it in most of the popular shrinkwrap applications. With that background, how can you not put it into your own application? Well, first of all, not every application can be made easier to use by simply tossing in some drag and drop features. In my own experience, I use drag and drop in only 1-2 out of ten programs I write. It has to add significant value to be worth the time it takes to program.

There are only two event to worry about, the DragDrop and the DragOver, so it won't take long to cover them both. In this example, the events of a form (Form1) are examined:

```
Private Sub Form1_DragDrop(Source As Control, X As Single, Y As Single)
Private Sub Form1_DragOver(Source As Control, X As Single, Y As Single, State
As Integer)
```

Both events share the common arguments Source, X, and Y, whereas the DragOver events includes the additional argument State.

- **Source**

This is a reference to the control which is being dragged. For example, if you drag a command button onto a form, Source refers to the command button. You would use Source.caption to refer to the caption of the command button. You can access other properties or methods of the command button the same way, by using the argument in place of the actual command button's name.

- **X/Y**

X and Y are the position at which the drop took place. You may use that information in a

variety of ways, but if it is used at all, it is typically used to reposition the control from which the drag and drop operation began.

- **State**

An integer value corresponding to whether the dragged object is entering or leaving the control or has simply moved its location within the control.

Drag and Drop Properties

The only two properties of a control which affect a drag and drop operation are:

- **DragMode** There are two settings - automatic or manual. If automatic is chosen, then all you do is drag a control (hold down the mouse while moving the cursor) and VB will automatically create the drag icon. If manual is selected, then you must invoke the .Drag method of the control for to begin the drag operation (including the display of the drag icon).
- **DragIcon** VB provides a default icon (a small outline image) but you may also define the icon to be used while the drag is taking place.

Drag and Drop Methods

- **Drag**
The only method needed is appropriately called Drag. To use it, simply use code such as:
- `command1.drag`
This will initiate a drag operation but as in the automatic case you still must write any code that is executed when the drop action occurs.

Graphics Commands

In VB there are a limited number of graphics features, but mixed with the judicious use of the Windows API a programmer can create some pretty fair graphics applications. When VB falls short, there is also no shortage of third party OCXs which can step in and add to the capabilities of VB.

Visual Basic Graphics

In VB, graphics capabilities are usually associated with drawing lines, boxes, or otherwise manipulating the display. Graphics are usually created **on** something, which in the case of VB is either the printer, the screen, or the picture control. All three represent a surface to which graphics can be applied. No other control supplied with VB supports the graphics features. I'll first walk through the graphical methods and properties which VB supports and then discuss them one at a time to help you understand how they play together. Here are graphics methods which VB supports.

- **Circle** - as you expect, draws a circle or an ellipse
- **Cls** - clears all graphics
- **Line** - draws a straight line
- **PaintPicture** - draws the contents of a graphics file
- **Point** - returns the color of a point
- **Print** - used for printing text
- **Pset** - sets the color of a specific pixel

Likewise, here are the graphics properties which VB supports.

-
- **AutoRedraw** - Determines if graphics are automatically redrawn if something moves in front of them
 - **ClipControls** - Determines how repainting of graphics is handled
 - **DrawMode** - Sets the appearance of a graphics method
 - **DrawStyle** - Sets the line style for output from graphics methods
 - **DrawWidth** - Sets the line width for output from graphics methods
 - **FillColor** - Sets the color used to fill circles, lines and boxes
 - **FillStyle** - Sets the pattern used to fill circle, lines and boxes
 - **FontTransparent** - Determines if the font is printed with a transparent background
 - **Palette** - Sets the image containing the palette to use for a control
 - **PaletteMode** - Determines which palette to use for the controls on a object
 - **RightToLeft** - Indicates the text display direction.
 - **ScaleHeight** - Sets the height of the client area
 - **ScaleLeft** - Sets the starting value of the left of the client area
 - **ScaleTop** - Sets the starting value of the top of the client area
 - **ScaleWidth** - Sets the width of the client area
 - **ScaleMode** - Sets the units of the scale

Scales

One of the first things about objects you need to understand is the concept of scales. The obvious question to ask about an object or graphic on a screen is "How big is it?". The answer to the questions can depend on the units chosen for the measurement. Inches, centimeters, millimeters, and pixels are all common units of measurement so you won't be surprised to know that VB supports all of these. Their definition should be self-explanatory.

VB also supports other units of measuring the size of something on the screen. Specifically, it supports:

- **Points**
Points are the same as you're used to seeing associated with a font. 72 points make an inch. Typically point sizes for typed text range from 10-12 for most documents.
- **Twips**
Microsoft has also introduced a unit known as a twip, where there are 1440 twips to the inch. The twip gives more resolution to measurements than the point (1 twip = 1/20 of a point), making possible more accurate graphics but still with units tied back to the printing industry's standard unit of measurement, the point. There are two properties available which help in converting units, the `twipsperpixelX` and `twipsperpixelY`. Both can be used to convert from pixels to twips. Both forms and the printer object support these two properties.
- **Characters**
Uses fixed twips (120 twips per horizontal character and 240 twips per vertical character)

definitions. This doesn't seem to get much use, but if you have some great places where it provides an advantage, then I'd like to hear about it.

- **User**

In this case, the name "user" means user-defined. This is one of the most powerful units you can use as a scale. Here, you define the number of units within the client area (top to bottom is height, left to right is width). The advantage of this is that you can **use any size window** with the exact same code! Sizing the window simply sizes the graphics that you create.

ScaleMode

The ScaleMode property simply identifies which units are to be used. You can set it to any of the units I've already discussed above. Once you pick a scalemode, VB adjusts the size properties of the client area to match the selection.

A point to remember is that when you create graphics, you'll be creating them in the client area of the window. In general, you'll be using an X/Y position within the client area to direct VB to place graphics. The bounds of the client area are defined by the properties

ScaleWidth and ScaleHeight.

VB also lets you pick the top/left starting coordinates of the client area, using the properties **ScaleTop and ScaleLeft**. This can be useful if you know that your graphics equation only apply to an area some distance away from the coordinate origin (X=0 and Y=0).

Using Graphics Methods

As you saw up above, there are only 7 graphics methods. It's really just 6 if you consider that the PRINT method is in a class of its own. Since I already covered printing in an earlier tutorial, let's just walk through each of the remaining 6 graphics methods and see how to use them. For this exercise, I assume we have a picturebox named Picture1 on a form.

- **Circle**

In this code, I simply draw 10 circles, progressively moving the center to the right starting from the coordinate of X=500.

- ```
For i = 1 to 1000 Step 100
```
- ```
    picture1.circle (i + 500, 1000), 400, vbGreen
```
- ```
Next i
```

The circle syntax can be pretty simple:

`picture1.circle (XCenter, YCenter), Radius, Color`

There are other things you can do as well, so check out HELP for a listing of all the arguments that are available.

- **Cls**

It doesn't get any simpler:

- ```
picture1.cls
```

The .cls methods will erase all graphics which have been drawn on the object, in this case the picturebox.

- **Line**

Similar to the use of the circle method, this example simply creates 10 vertical lines, positioned from left to right.

- ```
For i = 1 to 1000 Step 100
```
- ```
    picture1.line (i+500,500)-(i+500,4000), vbBlue
```
- ```
Next i
```

The line method can also draw boxes by simply adding an extra argument to the code as follows:

```
For i = 1 to 4000 Step 400
 picture1.line (i+200,500)-(i+400,3000), vbBlue, BF
Next i
```

---

The BF added at the end of the line of code simply tells VB to draw a filled box instead of a line.

- **PaintPicture**

- **Point**

- `i = picture1.point (100,500)`

Here, the color of the point located at X=100 and Y=500 is assigned to the variable i. In my own experience I haven't had any reason to use this method.

- **Pset**

This is the most basic drawing tool VB has to offer. With PSET you can set the color of any point within the client area of the drawing surface. For example, the next code randomly picks a color and randomly picks a coordinate at which to set the color. For no good reason I limit the number of points plotted to 1000

- `For i = 1 to 1000`
  - `'pick color value from 0 to 15 (the QBASIC standard color set)`
  - `iColor = (Int(15*Rnd)+0)`
  - `'pick X from 0 to ScaleWidth`
  - `X = (Int((Picture1.ScaleWidth * Rnd) + 0))`
  - `'pick Y from 0 to ScaleHeight`
  - `Y = (Int((Picture1.ScaleHeight * Rnd) + 0))`
  - `'now plot the data`
  - `picture1.pset (X,Y), QBColor(iColor)`
  - `Next i`
  - Note: To get a value from n to 15, replace the "0" with the value of n
- With PSET you can do almost any graphic you can imagine, limited only by the difficulty in controlling one point at a time. Also, the PSET method is not nearly so fast as some of the higher order graphics methods.

### **The Bottom Line**

In case it hasn't struck you yet, VB is not a graphics powerhouse. Its tools are very simple and there's just not very many of them. However, all is not lost!

You'll find that to create graphics of any sophistication will involve the use of the Windows API (Application Programmer's Interface), which are very powerful and very fast. However, even though the API are fairly powerful, they are not designed to be a graphics programmer's toolkit. For that purpose, Microsoft has created another set of DLLs, known as DirectX. DirectX was written for the more professional programmer who is looking to create the more complicated, higher performance applications. You can get DirectX for free from Microsoft.

DirectX works essentially the same way as do the Windows API. However my own experience is very limited so all I can do is to point you off to the DirectX API and let you experiment on your own.

### **Error Handling**

The bottom line is that all of us make mistakes. VB is pretty clever about finding syntax errors within the IDE, but a number of the possible errors simply cannot be discovered until you try to run the program. This section discusses some of the available error detection and correction techniques.

---

### **Users Hate Errors!**

If you make an error in VB, and you haven't written the code needed to respond to the error, do you know what happens? Well, VB simply puts up a message box saying there was an error. If you click on the OK button in the message then the VB program **shuts down** without saving anything that your user might have typed in. Let's be perfectly clear - This is not good! Your users will not like it and they **will** tell you about it.

---

It's amazingly hard for a programmer to figure out how a user will mis-use the program. Actually, the term "mis-use" is not correct because the user will simply do whatever the application allows him to do. If the programmer didn't anticipate a particular input that can crash the program or if the programmer didn't anticipate a particular combination of entries that could also crash the program, then shame on the programmer!

At a minimum, the programmer must allow for the possibility of an unknown error occurring and providing a means within the code to respond to the error, giving the user options to save (or not) any data already entered.

Contrary to the contempt some programmers have for non-technical users, it is not acceptable practice to release programs which lose the user's data if he doesn't follow the user's manual. This is especially true because often a user's manual is ambiguous, or written in techno-speak which the user might not understand.

It's also not acceptable to include in the user's manual an instruction that says "Don't do this or the program will fail!". If the programmer knows of an input that will crash the program, it's his responsibility to add the code to take care of the possible error condition.

In case you don't catch my drift here's my philosophy in a nutshell - It's the programmer's job to create a program which can respond to any possible user input without crashing and losing the user's data. You can whine all day long about how unskilled the users are, but you might as well accept that some user somewhere will get careless and make a weird combination of inputs that you didn't foresee. Under those conditions it is your job to protect the user from his own actions. Fortunately, VB has the features which let you provide this level of protection.

### Errors

When you make an error in VB, such as a divide by zero, VB responds by stopping execution of the program and will look for instructions in the code on what to do next. If you don't include those instructions, the program will simply shut down. As I've said - This is not good!

To respond to errors, VB provides the following code which turns on its error correction options:

```
On Error GoTo LabelX
```

What this means is that if an error occurs, then the program will resume execution at the line which is labelled "LabelX". Of course, you can use your own label name to identify where execution will begin.

In case you don't remember, a label is a short word/phrase which ends in a colon character, such as "BeginHere:". Any line can have a label. It goes at the start of the line and does not interfere with any additional code on that line.

One thing to remember is that the On Error statement **only** turns on error trapping for the procedure (Sub or Function) in which the code is found. You must use the On Error statement in every Sub/Function for which you want error trapping to occur.

Here's a very sample bit of code to explain how error trapping works:

```
Sub TestProcedure()
 On Error GoTo Label5
 Value = 10 / BottomNumber
 Exit Sub
Label5:
 If BottomNumber = 0 Then BottomNumber = 1
 Resume
End Sub
```

In this example, the On Error statement tells VB that if an error occurs, begin executing the code which starts at the line labelled "Label5". In that code (for this example), if BottomNumber is 0, an error will occur and execution starts with the "If" statement. That statement will set the denominator to 1 if it was zero. The "Resume" statement will tell VB to shift execution to the same line which gave the error in the first place! Now that the denominator is changed, no error occurs and the Exit Sub procedure will execute next.

There are some nuances to take notice of in this example:

### Resume

The use of the statement "Resume" can take 3 forms:

- 
- **Resume**  
When VB hits a Resume statement in the error code, it will move execution to the same line which caused the error, attempting to re-execute the line again. Be careful here, because for this to work, your error code must have fixed the problem that caused the error.
  - **Resume Next**  
When VB hits a "Resume Next" statement in the error code, it will move execution to the **line following** the line of code which caused the error. This is usually done when the error code does not, or cannot, correct the problem.
  - **Resume LabelX**  
In this case, VB is directed to a specific line within the current procedure. It can any line you chose, usually chosen based on the specific error that occurred. Generally, this approach is frowned on because it makes the code a little less readable, but if it fits your needs then I wouldn't hesitate to use it.

### **Exit**

Like an "Exit For" or an "Exit Do" statement, your code can direct VB to leave a procedure at any time. In the example above, the "Exit Sub" is used to leave the Sub procedure before the code following the line "Label5" is executed. This is a very typical way of preventing the error code from being executed.

If you want your application to take no action at all when an error occurs then you simply don't put any code behind the error label. In those cases, an "Exit Sub" is not necessary because the "End Sub" will give the same result.

### **Err Object**

To help out programmers, VB6 now provides a built-in object called "Err". On occurrence of an error, VB fills the properties of the Err object with information that uniquely identifies the error and with information that you can use in your code to figure out what actions to take as a result of the error. A bit further in this tutorial, I list the complete list of the Err object's properties and methods. It has no events.

The Err object's properties are reset to zero or zero-length strings ("" ) after an Exit Sub, Exit Function, Exit Property or Resume Next statement within an error-handling routine. Using any form of the Resume statement outside of an error-handling routine will not reset the Err object's properties. The Clear method can be used to explicitly reset Err.

The Err object is a recent VB feature. In older versions the Error statement was available (still is, for backward compatibility) and was used to have VB simulate an error. The Err.Raise method is now the preferred approach.

For test purposes, you can even cause VB to simulate an error. To do so, use the Raise method as described below.

The full list of Err object properties is:

- **Description**
- **HelpContext**
- **HelpFile**
- **LastDLLError**
- **Number**
- **Source**

And, it's methods are:

---

- **Clear**

Provides an immediate clearing of all Err object properties. Strings are set to null and numeric values are set to zero. You might choose to use it following the use of the statement "On Error Resume Next" since the error information would no longer apply

The Clear method is called automatically whenever any of the following statements is executed:

- **Any type of Resume statement**
- Exit Sub, Exit Function, Exit Property
- Any On Error statement

The value in clearing the Err properties is to make sure that any code which depends on those properties doesn't see information from a previous error which has already be responded to by code.

- **Raise**

### **Error Handling Heirarchy**

Last, but not least in this tutorial, is the discussion on error handling heirarchy. You should be aware that the code of a procedure might call a second procedure, which in turn can call even a third procedure. The question is how are errors handled in the second or third procedures?

If, in a procedure, an On Error statement is executed, then that error handler (the code at the error handling label) is considered to be "enabled". If the On Error statement is executed then when an error occurs, the error handler in that procedure is executed.

In a situation when a chain of called procedures exists (i.e., Procedure One calls Procedure Two which calls Procedure Three), then when an error occurs VB will move up the chain of procedures looking for the first enabled error handler it can find. If no enabled error handler exists, VB will give an error message and terminate the program (remember, this is bad!).

### **IDE Debugging Tools**

All of the discussion so far focused on how to handle errors in an application while the program was running. The VB IDE has some very clever tools which help you track down the source of the error. I don't go over these in this tutorial but I point them out and suggest you learn to use them in your own projects.

- **Breakpoints**  
Easily the most common method of debugging code. When a line of code is selected as a breakpoint, the program stops temporarily without losing the values of variables. Once stopped, you can check/change values or even re-write code as needed and then restart the program using those changes.
- **Step Into/Over/Out**  
A close second, the Step options are valuable because it gives you the ability to bypass blocks of code which you know to be good.
- **Windows - Local/Immediate/Watch**  
Provides a more sophisticated way to monitor specific variables and to take action based on the state/value of those variables.
- **Single Line Execution**  
Once you have stopped a VB program, you can execute the code one line at a time with this feature. It's a very methodical way to move through your program and confirm that any logic/redirection code works as intended.

### **Bottom Line**

Put error handling code in any application which you distribute to users. For your own purposes you can choose to do without it but don't expect your users to pay the price of your decision to deny them protection from errors. The VB IDE has some excellent tools to help you analyze and

---

debug code. You can prevent spending hours of time searching for errors simply by understanding and using the built-in IDE capabilities.

## Databases

I've read in print that over 80% of all VB programs require the storage of user/application generated data. That's a pretty big percentage but based on my own experience it's probably about right. The real question is how many of those use database formats such as Access and how many of those use less complicated formats? There are major tradeoffs involved in the selection of a data storage scheme. In this section I'll talk about how to use the database features of VB, but I'll also talk about when those features are overkill and when simpler methods of data storage can be substituted for the VB database methods.

---

## Data Storage

Let's get to the bottom line quickly on databases in VB. Microsoft has gone to great pains to add a lot of excellent methods for accessing data that is stored in not only Access databases, but other database formats as well. For beginner's they've done an excellent job! But even as I give praise to Microsoft for it's efforts, I can (and will) give you a few reasons why **not** to use VB's database features. But even with its negatives, I still chose to use database capabilities in a lot of my applications!

The huge penalty you pay for implementing those features is an enormous growth in the size of the program that you must distribute. Simply by putting in a single data control on a form, your distribution file size grows by over 5MBytes. Not only that, but database handling within VB is not anywhere near as fast as users might like it to be. The penalty you pay for features is a corresponding reduction in performance (i.e., speed)! But while the penalty is not trivial, it is not so great as to prevent VB programmers everywhere from using the VB data controls extensively. Because of the negative aspects of using the VB database features, I look hard at my programs and **do not** automatically use the database handling capabilities of VB until I'm certain that the handling of the data by other means is not practical.

Another **big negative** of an Access database (or any other proprietary format) is the inability to edit the file directly. If the file were simple text, you could edit it in a variety of methods. In the proprietary format of Access you basically have to own another Microsoft product to edit the contents of the database.

### Database Access - To Control or Not to Control!

VB has made it especially easy for beginners to utilize databases in their VB projects. With only one control (the data control) you can enable your program's other controls to display and edit data from the fields of that database. It really is that easy! VB also allows you to combine code and controls to quickly create very powerful applications.

VB also allows you to work with databases entirely in code, using no controls at all. The advantages to this are a greater level of control and faster performance. The penalty is that you have to do operations manually (in code) that the controls might have automatically done for you. I'll also cover this in more detail later.

### Database Formats

While I don't use it all that much myself, VB also offers significant flexibility in the types of databases it can access. The software industry has agreed upon a standard way of accessing databases, known as ODBC (Open DataBase Connectivity). What this means to you is that using VB you are not limited to those databases which come from Microsoft. Even though the Microsoft Access database format is an industry leader, VB nonetheless includes the ability to access other formats such as FoxPro, dBase, and others. You can even treat the contents of an Excel file or a text file as a database!

### Database Controls

Database controls are ideal if you want to get in, get the job done, and then get out. Microsoft has done an excellent job of making it easy for a beginner to utilize existing databases in their

---

applications. In this section we'll discuss the controls available to the database programmer. VB falls short, however, in helping beginners create databases from scratch. Note that I don't say VB doesn't have powerful database support, because it does. But beginners can easily get lost in the maze of detail that surrounds VB database creation and manipulation.

---

### What is a Database?

While there are a lot of concepts you need to know to become an expert at working with databases, here are three definitions with which you should start:

- **Database**  
"A set of data related to a particular topic or purpose. A database contains tables and can also contain queries and table relationships, as well as table and column validation criteria."
- **Table**  
"A table is a collection of data, arranged in rows and columns. For example, you might have a table for author information called authors. Each column would contain a certain type of information, such as the author's last name. Each row would contain all the information about a specific author: first name, last name, address, and so on."
- **Recordset**  
"A logical set of records. where Microsoft defines a record as "A set of related data about a person, place, event, or some other item. Table data is stored in records (rows) in the database. Each record is composed of a set of related fields (columns) - each field defining one attribute of information for the record. Taken together, a record defines one specific unit of retrievable information in a database."

In case you didn't catch the word "logical" in the definition of a recordset let me point it out. It is very important! In dealing with databases, VB creates a temporary object called a recordset, which contains the data (rows and columns) from one or more tables in the database. The recordset **is not the database, it is just a working copy** of some or part of the database tables. Again, pay attention to the "some or part" comment. A recordset does not necessarily contain the entire contents of a table within a database file. You can use SQL queries (discussed elsewhere in my tutorial) to limit the recordset to selected records. Finally, you should understand that the records in a recordset are in no particular order unless you write code to order them.

In your VB code you can manipulate the contents of the recordset but the actual database file will not be changed until you or the user take action to save the changes!

One final set of definitions which you should learn has to do with the types of recordsets which VB can create for you. There are actually 5 types of recordsets you can create, having to do with where the data can come from, how you can navigate through the records, and whether you can edit the data in the recordset.

The type of recordset you use **is very critical** to get good performance from your database application. Some recordset types take much fewer resources than others. Also, some recordset types result in a 10X faster performance. The default values VB uses to open recordsets are often not the best choice! Learn the differences and make a specific selection in your own programs.

Here are the five types of recordsets:

- **Table-type Recordset**  
Basically a complete table from a database. You can use to add, change, or delete records. This is the simplest in concept since it matches a complete table in the database, but is not necessarily the best choice in many applications.
- **Dynaset-type Recordset**  
The result of a query that can have updatable records. You can use to add, change, or

---

delete records. It can contain fields from one or more tables in a database. Provides worse performance than a Table-type recordset.

- **Snapshot-type Recordset**

A read-only set of records that you can use to find data or generate reports. Can contain fields from one or more tables in a database but can't be updated. Uses a minimum of resources and provides fast performance.

- **Forward-only-type Recordset**

Identical to a snapshot except that no cursor is provided. You can only scroll forward through records. This improves performance in situations where you only need to make a single pass through a recordset.

- **Dynamic-type Recordset**

A query result set from one or more base tables in which you can add, change, or delete records from a row-returning query. Further, records other users add, delete, or edit in the base tables also appear in your Recordset.

While they all have their place, it's my experience that the first three are the most useful.

### **Accessing Databases**

Remember that there are a variety of database products on the market, and that many vendors have their own proprietary formats in which the data is stored. In the case of Microsoft's Access database program, the method of accessing the database is through software which Microsoft calls its Jet Database Engine. Other vendors have their own access methods.

Many companies, including Microsoft, have provided their databases with the ability to be read by software which uses a standard database manipulation method known as ODBC (Open Database Connectivity). While a programmer can choose to let the VB database-aware controls handle the details of accessing or manipulating databases (recordsets), it is also possible to use code to create and edit databases.

### **DAO (Data Access Object)**

The first method VB offered for using ODBC to create/edit databases was DAO. It allowed the programmer to create a variety of objects in code, objects whose properties and methods could be used by programmers to create/edit the databases. This worked for Microsoft Access databases as well as any other ODBC-compatible database.

Microsoft also offered a second data access interface to ODBC which had some improved features over DAO, but which also lost some of the more general capabilities of DAO. This approach, called RDO (Remote Data Objects), primarily targeted the more complex aspects of stored procedures and result sets. It is generally not used by most programmers.

### **ADO (ActiveX Data Objects)**

Microsoft is in the process of replacing the ADO/RDO features with the newer ADO. ADO has a flatter model (meaning fewer objects) and is intended to be simpler to use. In its HELP files, Microsoft states that "ADO definitely represents the future of data access technology"

Personally, I would like the choice of database access technology to be transparent to me.

However it's obvious that the Microsoft approach is still under flux and that programmer's will need to be aware of multiple approaches for some time. This holds true because the ADO features have just been introduced and have not yet matched the DAO features of prior versions of VB.

For beginners, or any programmer who simply wants to put a data control on a form and then display database information in other controls, the choice of ADO/DAO/RDO is not critical because the controls handle it for you. The advantage in manipulating databases in code is that you have a far greater degree of control over what actions are performed against the databases.

### **VB Data Control**

VB provides two controls which makes the link to the database file and which creates the

---

recordset that is exposed to the rest of the controls in your application. The two are identical in concept but differ in the flexibility they offer to the programmer. I'll discuss the common features first and then cover some of the differences.

The concept of a data control is pretty simple. You set the `.DataBaseName` property to tell the data control which database file to read. Then you set a `.RecordSource` property to tell it which table within the file to make available as a recordset to other controls. As I've mentioned earlier, you can also create recordsets which are combinations of one or more tables by setting the `.RecordSource` property to an SQL statement.

For any controls which support databases, you set the `.DataSource` property to the data control and the `.DataField` property to the specific field within the table/recordset that you want bound to the control.

That's it! At that point the bound controls will display the information from the chosen field of the recordset. If you make changes to the data then the database can be changed to reflect the changes by either using the data control to move to a new record or by executing the `.Refresh` method of the data control.

Ok, perhaps I simplify things a bit too much because there really are other properties which a beginner must also be familiar with. But, it really can be pretty simple. You can create entire database applications with essentially **no coding**, just the setting of control properties.

If you didn't read my tutorial on SQL statements, then you've missed one of the most useful database capabilities that VB has to offer. I won't repeat all of that section again, but in a nutshell, you can set the `.RecordSource` to a command which tells the data control to create a recordset that is a selected subset of the fields in one or more tables. And there is lot more that you can do with SQL to reduce the amount of coding that you have to write yourself. Every VB programmer should be familiar with the use of SQL to create/manipulate recordsets. SQL can even be used to make mass changes to a recordset, including search and replace actions. Take the time to read my SQL tutorial section and to review the VB documentation on SQL. You'll use it the rest of your career!

Finally, we get to the actual list of the two data controls I mentioned.

- **Data control (intrinsic version)**

This is the original, intrinsic version of the control. It supports the JET database engine and can satisfy most beginners' needs.

- **ADO Data control**

This is the latest version of the data control, implemented as an ActiveX control.

In line with keeping your distribution files to a minimum, I suggest you use the intrinsic control whenever possible. I haven't seen enough information about the ADO Data Control to convince me that it's worth incorporating a separate ActiveX control into my applications.

### **Data Bound Controls**

Once you have the recordset available from the data control, you'll find VB to have a rich variety of controls which can access the recordset. There are 7 intrinsic controls and 16 ActiveX controls which you have available to use. Each has its place and over time you'll find yourself using most of them.

With the proliferation of controls, some of which are similar, Microsoft has created some confusion. In particular, there are multiple versions of the combobox, list, and grid controls. Let's try to address them first and then we'll get into the rest of the databound controls. If you really don't want to worry about selecting between multiple options of the controls, simply go with the latest version (DataList, DataCombo, and DataGrid) and don't worry about the nuances between the various versions of the controls.

### **ComboBox**

Here are the three versions that are available with VB, and some comments on how to decide which one to use.

- 
- **ComboBox**  
This is the original, intrinsic, version of the control. Use it whenever possible to keep down the size of your application.
  - **DataComboBox**  
This is Microsoft's most recent rendition of a combo control.
  - **DBCombo**  
Left over from VB5, you can forego using this one in favor of the newer DataCombo control.

### List

Here are the three versions that are available with VB, and some comments on how to decide which one to use.

- **ListBox**  
This is the original, intrinsic, version of the control.
- **DataList**  
This is Microsoft's most recent rendition of a list control.
- **DBList**  
Left over from VB5, you can forego using this one in favor of the newer DataList control.

### Grid

There are actually four versions of a grid control that come with VB, but only three of them can be databound. Here are the three versions and some comments on how to decide which one to use.

- **DBGrid**  
The oldest version that came with VB5. You'll have to find it on the VB CDROM because it doesn't get automatically installed.
- **FlexGrid**  
Also a VB5 version of the grid control. It does not support editing of the bound data.
- **Hierarchical FlexGrid**  
Newest version of the VB5 FlexGrid and still does not support editing of the bound data.

### Other Data-Aware Controls

There are 8 intrinsic controls which can be bound to fields in a recordset:

- **checkbox**
- **combobox**
- **data control**
- **image control**
- **label**
- **listbox**
- **picturebox**
- **textbox**

There are also 16 ActiveX controls which can be bound to fields/recordsets:

- 
- **ADO Data control**
  - **DataComboBox**
  - **DataGrid**
  - **DataList**
  - **DataRepeater**
  - **DateTimePicker**
  - **DBCombo**
  - **DBGrid**
  - **DBList**
  - **ImageCombo**
  - **MaskedEdit**
  - **MonthView**
  - **MSChart**
  - **MSHFlexGrid**
  - **MSFlexGrid**
  - **RichTextBox**

### **Database Reporting**

In the tutorial section on printing I lavished praise on how VB has made printing much easier. In the area of creating and printing reports, VB6 has taken even greater steps to simplify the programmers coding burden. There are now two options available to you which can **greatly** simplify the amount of coding it takes to print data that resides in a database.

In a prior version of VB a free version of a program called "Crystal Reports" was provided. It included an OCX you could distribute with your applications. A Pro Version was available commercially.

The magic of Crystal Reports was that you could create the report visually, in an IDE similar in concept to the VB IDE. Once the report was created you could use code to tie a database to it and then use code within your application to print (or preview) the report! The version that came with VB wasn't as flexible as the Pro version but it meant that you could create a professional-looking print report literally in a matter of minutes. Making changes under this concept is an order of magnitude easier than it would be to make changes in code which manually created the equivalent reports. I consider it an absolute winner that VB programmers should take advantage of.

With VB6, Microsoft brought out their own version of a report writer. They include Crystal Reports on the VB CDROM but it is not automatically loaded on your PC when VB is installed. You can install it at any time by heading over to the **\common\tools\vb\crysrept** directory of the VB CDROM and running the installation program.

I've yet to use the built-in report writer. Once I do, I'll update this section to provide a comparison.

### **SQL**

To selectively display records in a recordset, the feature to use is SQL - Structured Query Language. By creating a query (a text string which tells VB what to include in a recordset or

---

what actions to take against the data in a recordset) you can greatly simplify the code you have to write in an application that utilizes databases. You can even write SQL queries which will modify many records in a single operation. Once I understood the basics, the use of SQL hit me like a revelation. It's easily one of the top 5 features of VB's database handling capabilities!

---

## Introduction

VB has very poor documentation of its support for SQL. What SQL is or how it is used is a mystery for most beginners simply because it is so darned hard to find out any information on the topic! Those few references to SQL that exist in the VB documentation are very short examples and there is virtually no discussion on how to create your own queries.

I assume that part of the reason for the Microsoft approach to SQL is that it sells a product called SQL Server, which is a very powerful database interface. VB offers a limited set of the SQL Server features but Microsoft doesn't highlight those capabilities, preferring instead to sell its larger, more profitable product.

Considering the power of SQL statements I'm very surprised that Microsoft doesn't highlight the features more than they do. However, the fact is that beginners have to look hard to find help so that's where this tutorial comes into play.

## Sample SQL Queries

Looking at the positive side of things, I think you'll find that SQL is so intuitive that just by seeing a few examples you will gain a fair understanding of what is going on. Before I get into some of the details about using SQL, here are some examples that should help you get the feel for an SQL query.

```
"Select * From Title Where [Year Published] < 1889"
"Delete From Titles Where [Year Published] < #1/1/1889#"
"Select Name, Picture From Authors Where Date_of_Birth = #2/1/1947#"
"Select * From Employees"
"Select [First Name], [Last Name] From Employees"
"Select Employees, Department, SupvName From Supervisors, Employees Where
Employees.Department = Supervisors.Department"
"Select Distinct [Last Name] From Employees"
"Select [Last Name], Salary From Employees Where Salary > 2100"
"Select * From Orders Where [Shipped Date] = #5/12/93#"
"Select [Product Name], Sum ([Units in Stock]) From Products Group By
[Product Name]"
"Select * From Employees Order By [Last Name], Asc"
"Select [Last Name], [First Name] From Employees Order by 2 Asc"
"Select [Last Name], Salary From Employees Order By Salary, Desc, [Last Name]"
```

Three things to note about the examples:

- "\*" is used to denote all fields
- Dates are enclosed by pound signs, like this: "#2/1/1947#"
- Fields with multi-part names which include spaces are enclosed in brackets: [ ]

Now that you've read some of the examples, how do you use them? Simply set the RecordSource property of a data control to an SQL statement such as those above and refresh the control like this:

```
Data3.RecordSource = "SELECT * FROM Agency ORDER BY [City]"
Data3.Refresh
```

Just make sure that any references to fields match those contained in the actual database. Doing so will create a recordset whose content will match the constraints described by the SQL statement.

Considering that there are entire books on this subject, I can hardly expect to do it serious justice but there are some basics which can be summarized in a short tutorial like this one.

First of all, there are 5 parts to an SQL statement which you should recognize:

---

| Command | Clauses  | Predicates | Operators | Aggregate Functions |
|---------|----------|------------|-----------|---------------------|
| Create  | From     | Distinct   | AND       | Avg                 |
| Drop    | Where    | Top        | OR        | Count               |
| Alter   | Group By |            | NOT       | Sum                 |
| Select  | Having   |            | Between   | Max                 |
| Insert  | Order By |            | Like      | Min                 |
| Update  |          |            |           | In                  |
| Delete  |          |            |           |                     |

With a little inspection you can pretty much guess what each of these pieces of an SQL statement can do. However, here are a couple which you'll not want to miss and which I use pretty regularly. Don't miss the last one in my list, which is a very easy way to sort a recordset!

- **Select**  
This is the most basic command which tells VB which fields to show in the recordset.
- **Delete**  
Simple, but very powerful means of deleting many records at one time.
- **From**  
Defines the Table from which the fields will be extracted.
- **Where**  
Precedes the conditions by which records are selected from the database
- **Order By**  
Sorts the records by any combination of fields you chose.

This should get you started. Try out a few of the examples on one of the databases which come with VB and you'll see that it's really very easy to use the power of SQL!

### Code Librarian

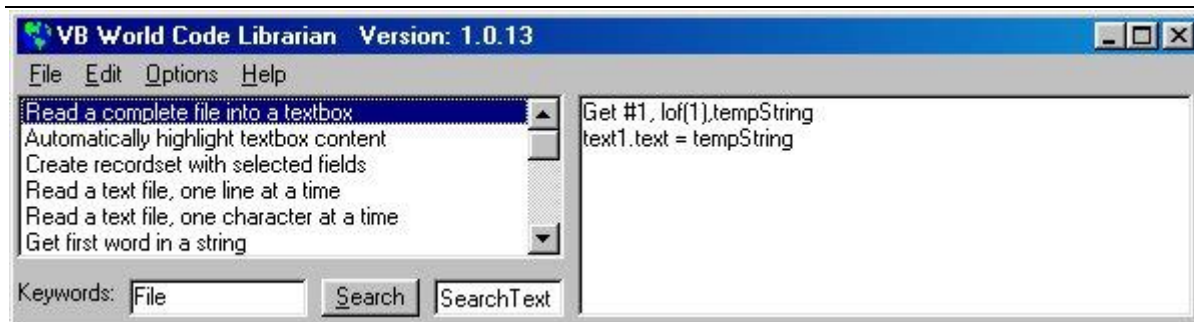
In response to requests for sample VB code, I've come up with the **VB Information Center Code Librarian (CodeLib), Version 1.6**. You can download the [source code](#) and then read the following discussion of the code to help you understand how the application works. CodeLib has dozens of code snippets which you can use in your own applications. Watch the [revision history](#) for new versions that become available. Plus, here are a few other sample projects that you might find interesting:

• [Countdown Timer](#) • [CaseMaker](#)

---

### Introduction

I hate dry, useless example projects. It makes the project so hard to walk through because you have to work to get to the good parts. CodeLib is written with the "instant gratification" in mind. The code is kept short, but useful. The list below gives the features that CodeLib demonstrates. You can jump to each section to see the source code and to look at my comments on how it works. To begin the discussions, here's an image of CodeLib's main screen.



The left side of the screen is a list of code snippet titles. On the right side is the code itself. In the lower left is a list of keywords that have been associated with each snippet of code. Finally, there is a text box where you can type in a search string. CodeLib can search through the titles, keywords, and code lines for the search string.

Before I get into the code, though, let's talk about the CodeLib program. When you're in the VB IDE it's nice to be able to reach into a library somewhere and pull out a snippet of code that adds a particular features to your own application. That's exactly what CodeLib does. While you're in the VB IDE, start CodeLib also. It's window is kept reasonably small so that it can be placed at the bottom of your screen, pretty much out of the way (or you can simply minimize it).

When you need a code snippet, go to CodeLib and scroll through the available routines by moving through the list on the left side of the screen. Once you find what you want, simply click on the "Copy" menu selection and the entire multi-line code snippet will be placed on the clipboard.

Then go back into the VB IDE, put the cursor where you want the code to go and use the Windows paste command (Ctrl-V or Edit/Paste) to put the code snippet into IDE code window. You can also search through the code for any keyword/text string that you want, making it easier to find the code you need.

I'll upgrade CodeLib over the next few months and provide additional features - but always with an eye towards using CodeLib to demonstrate how to implement VB feature options.

Of course, CodeLib also allows you to enter new snippet or to change the ones that are already there. I use a second window for the edit screen to help demonstrate how to talk between windows within an application.

Here are some of the features which CodeLib demonstrates:

[Splash Screen](#)

[Clipboard](#)

[Registry](#)

[Inter-form variable referencing](#)

[Menus](#)

[Auto-sizing forms](#)

[About form](#)

[Calling HELP files](#)

[Confirming an action](#)

[Confirming exit from a program](#)

[PopUp menus](#)

[Splash Screen](#)

[Read/Writing Binary Files](#)

[Automatic selection of text when control gets focus](#)

[Go to next control when ENTER is pressed](#)

[Save/Restore window positions for next session](#)

[Search an array](#)

[Sort an array](#)

[Deleting an element from an array](#)

[Adding an element to an array](#)

[Printing a simple report](#)

[To Database or Not?](#)

More often than not it takes several seconds for a VB application to load up all of the files (EXE, DLLs, OCXs, and others) needed by the program. To entertain the user while all that loading is going on, many VB programs first display a simple (fast-loading) screen which draws the users attention. After a moment, once all the application's files have been loaded, the VB application unloads the Splash screen and displays the main screen of the application. CodeLib does this as well.

All it takes is a simple form with a graphics to entertain the user and a timer control placed on the form. In CodeLib I set the timer interval for 1.5 seconds, at which time the splash form

---

unloads itself and then loads CodeLib's main form. The 1.5 seconds is adequate for the user to actually look at the splash form, and long enough for the rest of CodeLib to load, but not too long to bore the user.

You may have to experiment with the timer's interval setting on your own application to determine how long the splash screen should be kept visible. Here's the code for the Timer control event:

```
Private Sub Timer1_Timer()
Unload Me 'unloads the splash form
frmMain.Show 'loads the main form
End Sub
```

### **Menus, Sub-Menus, and Control Arrays**

With VB it is exceptionally easy to create a menu. In the IDE, just highlight the form for which you want to make a menu and then call up the Menu Editor. The tools within the editor are pretty simple to understand and they allow you to create exactly the same kind of menu as most other Windows application.

In several places in my menu I create sub-menus. However, I've show a little trick which can often help you to greatly simplify your code. In the CodeLib toplevel menu "Options", there are two normal sub-menus: "Minimize on Copy" and "Search". They work just like you would expect. However, at the "Search" menu there is another level of sub-menus and there is where is use the concept of a control array to save me from writing more code than is necessary.

The concept of a control array is pretty simple. Create two controls with example the same name and VB will treat them as an array (VB forces you to give each one a unique number, called an Index). When you click on any one control that is within a control array VB will execute an event which is common to all of the controls within the control array. VB also passes the Index value to the event procedure so that you know which control was clicked.

The magic of this is that you don't have to repeat code for each array! Make sure you caught this.

**You don't have to repeat the code!** In CodeLib, under the "Search" menu, are the sub-menus "All", "Key", "Title", and "Code". Each is part of the same control array and when you click on any one of them you see the exact same event procedure. Then, within that event procedure, you only have to write code once which uses the Index value to help determine what action you want to take.

In the case of CodeLib, I use code to check the selected item and un-select all others. I also save the Index value in the Windows registry so that the next time I start CodeLib I can restore the last setting.

### **Auto-sizing the contents of a form**

Windows allows users to manually change the size of its windows but it is up to the VB programmer to create the code to change the size of the controls inside the window appropriately. In frmMain and frmEdit I use code to keep the controls sized proportionate to the window size. Here is the code for sizing the contents of frmMain - note that the code is placed within the resize event procedure of the form.

```
Private Sub Form_Resize()
'set width/height of controls (check for illegal values)
If frmEdit.ScaleWidth - txtCode.Left - 50 > 200 Then
 txtCode.Width = frmEdit.ScaleWidth - txtCode.Left - 50
 txtKey.Width = txtCode.Width
 txtTitle.Width = txtCode.Width
End If
If frmEdit.ScaleHeight - txtCode.Top - 50 > 200 Then
 txtCode.Height = frmEdit.ScaleHeight - txtCode.Top - 50
End If
End Sub
```

### **Clipboard**

The clipboard is a really great feature! It's the poor man's way of communicating between programs and of making data available to other programs. And the really cool thing about it is

---

that it can handle graphics or text equally well! Believe me, that's a big benefit - eliminating code you would otherwise have to write.

One of the most often used features of the clipboard is to simply copy text into the clipboard's memory so that it can then be pasted into another program. With CodeLib, that's exactly what happens. In many programs it is normal for you to have to select some text and then use the Edit/Copy menu commands (or Ctrl-C) to copy the selected text. With CodeLib I've opted to just have a single copy command in the menu which will copy the entire code snippet, without the need for you to first select the code.

Microsoft has done a good job of keeping access to the clipboard simple. The code from frmMain which puts the code snippets onto the clipboard is:

```
Private Sub mnuEditCopy_Click()
'clear clipboard then set to code
Clipboard.Clear
Clipboard.SetText lblCode.Caption
```

Once you have it in the clipboard, you can paste it into any other Windows program by using Ctrl-V or by using the menu selections Edit/Paste. It's no more complicated than that.

### **Registry**

The ability to store data in the registry is analogous to storing project information into a file. Previously, applications used a .INI file, which was a simple text file that stored information the program would need the next time it ran.

For reasons that I have trouble appreciating, Microsoft has decided to get away from the .INI file strategy and now goes with a "Master INI" file that they call the Registry. In lieu of using individual .INI files, Microsoft encourages all application designers to put their program-specific data into the Registry. VB provides functions which can read/write information into either the Registry or to a .INI file, so you still have freedom of choice. Personally, I've been using the Registry simply because that means my users have fewer files to worry about.

For your purposes, it's really not that critical whether a .INI file or the Registry is used to store data. They both are transparent to the user of your application and both are supported by VB. In general, I'd suggest going to the Registry approach because Microsoft may yet decide to discontinue the .INI file features.

In case you don't appreciate what you **could** do with the Registry, take a look at this list:

- Store program options that your user selected in a prior session
- Store window positions from a prior session
- Store password data for verification of valid users
- Store information that can be easily retrieved by other programs

Of course, for every "store" feature there is a corresponding "retrieve" option for reading and using the data in an application.

In CodeLib I use the two simple functions that are all most folks need to save and retrieve Registry information - "GetSetting" and "SaveSetting":

```
Status = GetSetting("CodeLib", "Options", "Minimize", "TRUE")
SaveSetting "CodeLib", "Options", "Minimize", "TRUE"
```

The first line reaches into the Registry into a section called CodeLib/Options and reads a value of the property "Minimize", using a default value of "TRUE". The second line write a value into the Registry, again in a section called CodeLib/Options, setting the "Minimize" property to a value of "TRUE". If it looks simple that's because it is. As a beginner I remember having the impression that using the Registry was going to be difficult but it really is as simple as I just described.

### **Inter-form variable referencing**

What do you do if you're executing code in one window and you want to use values from another form? The following code from CodeLib shows what to do:

---

```
txtTitle = frmMain.lstTitles.List(frmMain.lstTitles.ListIndex)
```

In this example (found on the frmEdit) I set the value of a textbox called txtTitle. I fill it with the content of a listbox on frmMain. All I have to do to access the listbox is to precede the control name by the name of the form! So, the "frmMain." at the front distinguishes between a control on the current form or on another form entirely!

### The About form

Almost all applications use an "About" form, which is typically used to identify the author of the program and any information he chooses to provide to help you locate him. The next line of code calls the About form. Note the use of the vbmodal constant. This forces the user to close the About form before continuing with the application.

```
frmAbout.show vbmodal
```

The next code listing is used in the load event of the About form to display. I keep the credits in an external text file that I can easily change, and simply read the entire text file into a textbox for viewing.

```
Private Sub Form_Load()
Dim temp As String, message As String
'load credits file
ChDir App.Path
Open "credits.txt" For Input As #1
While Not EOF(1)
 Line Input #1, temp
 message = message & vbCrLf & temp
Wend
Close #1
txtCredits.Text = message
End Sub
```

### Calling HELP files

In the VB IDE, use the Project/Project Properties menu to get to the dialog box where you can type in the name of a HELP file that will be called up when you press F1 while your program is running.

You can also set the HELP filename in code using:

```
app.helpfile = filename
```

Once you set the HELP filename for an application pressing F1 will call up the HELP file. You can also call up a HELP file by using the Windows API, by using the CommonDialog Box, or by simply running the WinHelp EXE program that displays the HELP files. In frmMain I use the following code to display the CodeLib HELP file:

```
Private Sub mnuHelpContent_Click()
Shell "winhelp " & App.HelpFile, vbNormalFocus
End Sub
```

### Confirming an action

In the CodeLib menu is a Delete option. To prevent accidental deletions of a code snippet I include the following statement in the click event of the menu control:

```
If MsgBox("Are you sure?", vbOKCancel, "Delete Code") = vbCancel Then Exit Sub
```

Since MsgBox is a function I can simply call it and check the return value to determine which button in the MsgBox was pressed by the user. By using the "vbcancel" VB constant I don't have to remember the actual numerical value.

### PopUp Menus

Creating a popup menu that appears when you right-mouse click an object is made easy by the VB PopUpMenu function. In CodeLib, the main menu provides the following code which provide popup menus with most of the controls on the form. The following code is placed in the MouseDown event procedure:

```
If Button = 2 Then PopupMenu mnuListPopup
```

One thing to notice is that the popup menus **do not** replicate the code of the other menus. I just use the popup menu to call out the name of an event procedure for one of the other menus. Also,

---

notice that the menu which is called by the right mouse is made invisible but that only it's visible sub-menus will be show in the popup menu.

### **Confirming exit from a program**

In almost all programs you will want to ask the user to confirm that he wishes to exit from the program - primarily to give the user a chance to save any changes that were made. In CodeLib I created a Global variable called FileChanged and whenever a change to the data was made I set FileChanged to TRUE. Then, in the Unload event of the form I can check to see if changes were made and ask the user whether to save the changes. Note that if I Save changes to the data before exiting from the program I set FileChanged to FALSE.

To ask the user if changes should be saved, I again use the MsgBox function, using the returned value as in a Select Case decision loop. The code is put in the Form\_Unload event and the Cancel argument is changed to match the users selection:

```
Private Sub Form_Unload(Cancel As Integer)
If FileChanged = True Then
 Select Case MsgBox("Save changes?", vbYesNoCancel, "Exit")
 Case vbYes
 'save the data, leave cancel as is
 mnuFileSave_Click
 Case vbNo
 'take no action, leave cancel as is
 Case vbCancel
 'stop the unload action
 Cancel = 1
 End Select
End If
```

### **Read/Writing Binary Files**

In another part of my tutorial, I noted that VB supports the creation and editing of both ASCII text files as well as the direct manipulation of binary files. In CodeLib, I made use of the fact that in binary file writing you do not have to specify the format of the stored material. You'll see in my GET and PUT statements that I simply list the variable names which I want to write to the file and VB handles the formatting for me.

Even more useful is that VB allows you to write variables of a User-Defined Type by referencing only the variable name. You do not have to specifically list all of the sub-elements to the user-defined type variable! This is very powerful so you'll want to review my Save code.

```
Private Sub mnuFileSave_Click()
'save the data
ChDir App.Path
Kill "codelib.dat"
Open "codelib.dat" For Binary As #1
LibDate = "-04/25/99-"
Put #1, , LibDate 'single variable
Put #1, , MaxCode 'single variable
Put #1, , Code 'the array
Close #1
FileChanged = False
End Sub
```

### **Automatic selection of text when control gets focus**

When a user clicks on a textbox it is useful to have the entire content of the textbox become highlighted. This code does that for you:

```
Private Sub txtTitle_GotFocus()
txtTitle.SelStart = 0
txtTitle.SelLength = Len(txtTitle.Text)
End Sub
```

### **Go to next control when ENTER is pressed**

Sometimes, when the user is entering data you may want to take an action only when the ENTER key is pressed. In frmEdit you will find the following code in the KeyPress event

---

procedure. The KeyAscii value for ENTER is 13, so in this example pressing the ENTER key causes the focus to move to the control txtKey.

```
Private Sub txtTitle_KeyPress(KeyAscii As Integer)
If KeyAscii = 13 Then txtKey.SetFocus
End Sub
```

### **Save/Restore window positions for next session**

After a user takes the time to move the windows to the location that meets his work habits, you might as well go to the effort to store the positions of those windows. The following two procedures are used in the frmMain to save the positions on exit and to restore the positions on return to the program.

```
Private Sub Form_Unload(Cancel As Integer)
'save position of window
SaveSetting "CodeLib", "Options", "EditPositionTop", frmEdit.Top
SaveSetting "CodeLib", "Options", "EditPositionLeft", frmEdit.Left
SaveSetting "CodeLib", "Options", "EditPositionHeight", frmEdit.Height
SaveSetting "CodeLib", "Options", "EditPositionWidth", frmEdit.Width
End Sub
```

This next code was taken from the Form\_Load event of frmMain, and sets the starting window position and size:

```
frmMain.Top = Val(GetSetting("CodeLib", "Options", "PositionTop",
frmMain.Top))
frmMain.Left = Val(GetSetting("CodeLib", "Options", "PositionLeft",
frmMain.Left))
frmMain.Height = Val(GetSetting("CodeLib", "Options", "PositionHeight",
frmMain.Height))
frmMain.Width = Val(GetSetting("CodeLib", "Options", "PositionWidth",
frmMain.Width))
```

### **Search an array**

There are many sophisticated ways to search through data. However, in CodeLib, the library size is not very large and so I chose to simply walk through each and every entry to see if I could find a match for the search string. There's nothing all that special about my search routine. I do allow the user to chose whether to search the title, keywords, or codelines for a search string. I make use of the INSTR function to do my string searching.

```
Private Sub cmdSearch_Click()
Dim i As Integer, Location As Integer, StartPoint As Integer
Dim SearchString As String, Status As Boolean

'set the starting point
If mnuOptionsStartTop.Checked = True Then
 StartPoint = 0
Else
 If lstTitles.ListIndex < lstTitles.ListCount - 1 Then
 StartPoint = lstTitles.ListIndex + 1
 End If
End If

'search the database
For i = StartPoint To lstTitles.ListCount - 1
 'set the search string
 If mnuOptionsSearchChoice(0).Checked = True Then 'all
 SearchString = lblKey & lblCode & lstTitles.List(i)
 ElseIf mnuOptionsSearchChoice(1).Checked = True Then 'key
 SearchString = lblKey
 ElseIf mnuOptionsSearchChoice(2).Checked = True Then 'title
 SearchString = lstTitles.List(i)
 ElseIf mnuOptionsSearchChoice(3).Checked = True Then 'code
 SearchString = lblCode
 End If
 'search the string
```

---

```

 Location = InStr(LCase(SearchString), Trim$(LCase(txtSearch.Text)))
 If Location > 0 Then
 Status = True
 Location = i
 Exit For
 End If
 Next i

 'go to the match if found
 If Status = True Then
 lstTitles.ListIndex = Location
 Else
 MsgBox "Search string not found!", vbOKOnly, "Search"
 End If

End Sub

```

### Sort an array

There are many ways to sort a list or an array of data. I've used what is known as a bubble sort. It's not very efficient but it is easy to code and for small lists it works quickly enough not to be a problem. Here's the sort routine from frmMain:

```

Private Sub mnuEditSort_Click()
 Dim i As Integer, j As Integer
 Dim temptitle As String, tempkey As String, tempcode As String
 'sort the Code() array
 For i = 0 To MaxCode - 1
 For j = i + 1 To MaxCode
 If Code(i).title > Code(j).title Then
 temptitle = Code(i).title
 tempkey = Code(i).keywords
 tempcode = Code(i).CodeLines
 Code(i).title = Code(j).title
 Code(i).keywords = Code(j).keywords
 Code(i).CodeLines = Code(j).CodeLines
 Code(j).title = temptitle
 Code(j).keywords = tempkey
 Code(j).CodeLines = tempcode
 End If
 Next j
 Next i

```

### Deleting an element from an array

To remove an element from an array you basically move all of the data in the array down by one position and then eliminate the top-most position by resizing the array to an upper bound of 1 less than it was before the delete action. I use the technique in frmMain:

```

Private Sub mnuEditDelete_Click()
 'verify actions (only if confirm on delete is checked)
 If mnuOptionsConfirm.Checked = True Then
 End If

 'delete the code snippet
 Dim i As Integer, Start As Integer
 FileChanged = True
 Start = lstTitles.ListIndex

 'move all data down 1 position, unless the line to delete is the last line
 If lstTitles.ListIndex <> lstTitles.ListCount - 1 Then
 For i = lstTitles.ListIndex To lstTitles.ListCount - 1
 Code(i).title = Code(i + 1).title
 Code(i).keywords = Code(i + 1).keywords
 Code(i).CodeLines = Code(i + 1).CodeLines
 Next i
 End If
 End If

```

---

```
'lower the array size by one
MaxCode = MaxCode - 1
ReDim Preserve Code(MaxCode) 'note the PRESERVE keyword!
```

### **Adding an element to an array**

To add an element to an array you simply resize the array by 1, using the keyword PRESERVE so as not to lose the data. Then write the new data into the new position at the end of the array. In frmEdit, the following code is executed by the Save command when a new snippet is being added.

```
'if new, then increment MaxCode, array size, and lstTitles
If EditStatus = "new" Then
 'resize the array
 MaxCode = MaxCode + 1
 ReDim Preserve Code(MaxCode)
 'add new entry to lstTitles
 frmMain.lstTitles.AddItem txtTitle
 frmMain.lstTitles.ListIndex = MaxCode - 1
End If
```

### **Printing a simple report**

The printing routine I use in CodeLib is pretty plain, but it still can be used to show you how to print out data. In printing you may often want to precisely determine where a print action is to take place, and that means you would be setting the .CurrentX and .CurrentY properties. If you simply follow one print statement after another then VB will simply set the CurrentX and CurrentY values to correspond to normal lines of print one below the other starting at the left margine of the page. That's what I do in CodeLib.

```
Printer.Print "VB Information Center Code Librarian"
Printer.Print
Printer.Print
For i = StartSnippet To EndSnippet
 Printer.Print "Title: "; Tab(15); Code(i).title
 Printer.Print "Keywords: "; Tab(15); Code(i).keywords
 Printer.Print
 Printer.Print Code(i).CodeLines
 Printer.Print
 Printer.Print "===== "
 Printer.Print
Next i
Printer.EndDoc
```

### **Reading sequential text files**

I often need to read a sequential text file, either to grab the whole thing or to take action on a line by line basis. The CodeLib library is a binary file, but the credits listed in the About form are kept in a sequential file called "credits.txt". The following code is from the frmAbout load event and shows two different ways to read in the sequential text file.

```
Private Sub Form_Load()
Dim tempString As String
ChDir App.Path
Open "credits.txt" For Binary As #1
tempString = Input(LOF(1), #1) 'reads the whole file with a single line of
code
Close #1
txtCredits.Text = tempString
```

```
'The following code is an alternate way of reading a text file
'reading one line at a time
'Dim temp As String, message As String
'load credits file
'ChDir App.Path
'Open "credits.txt" For Input As #1
'While Not EOF(1)
' Line Input #1, temp
' message = message & vbCrLf & temp
```

---

```
'Wend
'Close #1
'txtCredits.Text = message
```

### To Database or Not?

I plan to release a second version of CodeLib in a few weeks. The second version will not be a replacement of this one. It will use VB database capabilities (versus the binary file handling used in this version of CodeLib), and will be useful in showing how the application of database features can both be helpful as well as cause problems in writing and releasing an application.

### CodeLib Revision History

- Version 1.8 - 10 June 1999
  - Added a simple game under the HELP menu
  - Added API code to show how to execture a WAV file (in the game)
  - Added code to show how to implement drag & drop (in the game)
- Version 1.7 - 08 May 1999
  - Added API code to keep the CodeLib window on top
  - Added API code to start the default browser at Gary Beene's VB Information Center
- Version 1.6 - 30 April 1999
  - Updated database to include the API references from my tutorial
  - Corrected problem with data corrupting during edit
- Version 1.5 - 29 April 1999
  - Added option to stop display of splash screen by double-clicking label
- Version 1.4 - 29 April 1999
  - Added right-mouse popup menus
- Version 1.3 - 28 April 1999
  - Corrected problem with Save/Delete giving error
  - Corrected path problem with reading the CodeLib database

### API

As a programmer you might never have to use a Windows API.... Nahhh! The fact is that serious programmers use API all the time. When they need to do **something that VB cannot handle**, they turn to the Windows API! The API are procedures that exist in files on your PC which you can call from within your VB program - and there are **thousands** of them!. Written by Microsoft, debugged by tens of thousands of users, and available for free with Windows - the API are one of the very best tools you have available to add power to your VB application.

---

### API - Application Programmer's Interface

When Microsoft wrote Windows they put a huge amount of code into procedure libraries which programmers can access. No matter which language you're using (VB, C++, ...) you can use the Windows API to greatly expand the power of your application.

---

There are a lot of Windows programs whose code is spread across multiple files. The .EXE file does not always contain all of the code that a programmer might use. For example, by creating his own library of procedures (usually in the form of a file with a .DLL extension) a programmer can allow more than one of his applications to access the same code.

Microsoft does a similar thing with Windows. There are many files which have code that you can access, but the three most often named files are:

- **user32.dll** - controls the visible objects that you see on the screen
- **gdi32** - home of most graphics oriented API
- **kernel32.dll** - provides access to low level operating system features

Later, I'll bring up some of the other files whose procedures you might want access. However, there are some key issues which you should note before making a decision to use an API call.

- **Version Compatibility**

Microsoft has long been known to update it's files without much fanfare - read that as without telling anyone about it until it's already happened! And often, the updated code may not perform exactly as did the older version. Users often find this out by seeing unexected errors or by having their system crash and/or lock up on them! In VB5 there were a huge number of programmer's who got bit by this problem.

If you stick with the basic 3 OS files listed above, you won't see to much of this. But the further away you go from the main 3 files, the more likely you are to get into code which hasn't seen the testing and improvement cycle that the main Windows OS files have gone through.

- **File Size**

One of the **very major** downsides to the concept of API is that all of this great code lives in some very big files! Worse yet, sometimes the API you want are spread over multiple files and you may be using only one or two procedures from enormous files which have hundreds of procedures in them. Where this becomes a problem is in a)load time - where it can takes several seconds to load the procedure libraries, and b) - where you want to distribute your application and in order to make sure that all of the procedure libraries are on your user's machine, you have to put all of them into the distribute files. This can add many megabytes of files to your distribution applications. It is a major problem for distribution of software over the net, where 5 minutes per megabyte can deter a usage from trying out an application just because he doesn't want to wait for the download!

- **Documentation**

Finding the documentation of what procedures are in a library and how to use them can be very difficult. On my PC I have 3,380 files with a .DLL extension with a total size of 539MB. That's a lot of code! Unfortunately I can count on one hand the pages of documentation that I have to tell me what that code is or does! You'll learn **how** to use DLLs in this tutorial, but without the documentation from the creator of the DLLs you cannot use them successfully.

Despite these problems, the powerful magic of the API is that they are code which you don't have to write. If you've read my Beginner's section you know that I am a big fan of using 3rd party software to lighten my own programming load. As with 3rd party controls, the API provide procedures which someone else wrote, debugged, and made availble for you to benefit from. In the Windows DLLs files, there are literally thousands of procedures. The key to API programming is learning which of these procedures are useful and which ones you are unlikely to ever need! This tutorial tries to address just that problem.

---

**Getting Started** It's actually simpler than you might imagine. By now, you've already written procedures for your own VB programs. Using procedures from other files is almost exactly the same as using procedures from within your own program.

The one big difference is that you must tell your application which file the procedure is contained in. To do so, you must put 1 line of code into your VB program. And, you have to do this for every external procedure that you plan to use. I suppose it would be nice for VB to have the ability to find the procedures for you - but you can see that searching through the 3,380 procedures on my PC might slow my applications down a lot!

Ok, let's get to an example. Telling VB about the procedure you want to use is known as "declaring" the procedure, and (no surprise) it uses a statement which starts with the word **declare**. Here's what a declaration looks like:

```
Declare Function ExitWindowsEx Lib "user32" (ByVal uFlags as Long, ByVal dwReserved as Long) as Long
```

Let's take about the parts of the declaration statement:

- **"Declare"**  
This is reserved word that VB uses to begin a declaration. There is no alternative - you have to use it.
- **"Function"**  
Also a reserved word, but in this case it distinguishes between a SUB procedured and a FUNCTION procedure. The API use Function procedures so that they can return a value to indicate the results of the action. Although you can discard the returned value, it's also possible to check the return value to determine that the action was successfully completed. completed. alternative - you have to use it.
- **"ExitWindowsEx"**  
Inside each DLL is a list of the procedures that it contains. Normally, in a VB declaration statement you simply type in the name of the procedure just as it is named in the DLL. Sometimes, the DLL true name of the procedure may be a name that is illegal in VB. For those cases, VB allows you to put in the text string "Alias NewProcedurename" right behind the filename. In this example, VB would make a call to the procedure by using the name "NewProcedureName".
- **"Lib 'user32'"**  
Here's where you tell VB which file the procedure is in. Normally you would put "user32.dll", showing the extension of the procedure library. For the special case of the three Windows system DLLs listed above, the API will recognize the files when simply named "user32", "kernel32", and "gdi32" - without the DLL extensions shown. In most other cases you must give the complete file name. Unless the file is in the system PATH, you must also give the complete path to the file.
- **"(ByVal uFlags as Long ...)"**  
Exactly like your own procedures, Windows API functions can have a list of arguments. However, while your VB procedures often use arguments passed by reference (i.e., their values can be changed), most Windows API require that the arguments be passed by value (i.e, a copy of the argument is passed to the DLL and the original variable cannot be changed).

Also, you'll note that a constant or variable is normally used as the argument for an API call. It's technically acceptable to simply use a number for an argument but it is common practice among experienced programmers to create constants (or variables) whose name is easy to remember and then to use those in the argument list. When you're reading or debugging your code later, the use of these easy to read constant/variable names makes it much easier to figure out what went wrong!

---

- **"as Long"**

This is exactly like the code you use to create your own functions. Windows API are functions which return values and you must define what type of variable is returned.

While I make it sound simple (and it is), there are still issues which ought to concern you when using the Windows API. Because the API code executes outside the VB program itself, your own program is susceptible to error in the external procedure. If the external procedure crashes, then your own program will crash as well. It is very common for an API problem to freeze your system and force a reboot.

The biggest issue that VB programmers would see in this case is that any unsaved code **will be lost!**. So remember the rule when using API - save often!

Because many of the DLLs you will use have been debugged extensively you probably won't see many cases where the DLL crashes because of programming bug. Far more frequently VB programmers will see a crash because they passed arguments to the procedure which the procedure could not handle! For example, passing a string when an integer was needed will likely crash the system. The DLLs don't include extensive protection in order to keep their own code size small and fast.

It is simple to say that if you pass the correct type of argument, that you won't see API crashes. However, the documentation is not always clear exactly what argument type is needed, plus when writing code it is all too common to simply make a mistake!

Finally, it is the case that most of the DLLs you'll want to use were written in C++. The significance of this is that the data types in C++ do not map cleanly into the data types that are used in Visual Basic. Here are some of the issues which you need to be aware of:

- **Issue1**

- **Issue2**

Okay, stay with me just a bit longer and we'll get into the actual use of some API. But first, here is a list of other DLLs which have procedures that could be of use to you. These DLLs will show up later in this tutorial when we get to the API which I recommend that you consider for use in your own applications.

- Advapi32.dll - Advanced API services including many security and Registry calls
- Comdlg32.dll - Common dialog API library
- Lz32.dll - 32-bit compression routines
- Mpr.dll - Multiple Provider Router library
- Netapi32.dll - 32-bit Network API library
- Shell32.dll - 32-bit Shell API library
- Version.dll - Version library
- Winmm.dll - Windows multimedia library
- Winspool.drv - Print spooler interface

Often, the documentation that you might find for an API will be written for a C++ programmer. Here's a short table which helps you translate the C++ variable type declaration to its equivalent in Visual Basic:

|      |                           |
|------|---------------------------|
| ATOM | ByVal variable as Integer |
| BOOL | ByVal variable as Long    |
| BYTE | ByVal variable as Byte    |
| CHAR | ByVal variable as Byte    |

---

|          |                                                 |
|----------|-------------------------------------------------|
| COLORREF | ByVal variable as Long                          |
| DWORD    | ByVal variable as Long                          |
| HWND     | ByVal variable as Long                          |
| HDC      | ByVal variable as Long                          |
| HMENU    | ByVal variable as Long                          |
| INT      | ByVal variable as Long                          |
| UINT     | ByVal variable as Long                          |
| LONG     | ByVal variable as Long                          |
| LPARAM   | ByVal variable as Long                          |
| LPDWORD  | variable as Long                                |
| LPINT    | variable as Long                                |
| LPUINT   | variable as Long                                |
| LPRECT   | variable as Type any variable of that User Type |
| LPSTR    | ByVal variable as String                        |
| LPCSTR   | ByVal variable as String                        |
| LPVOID   | variable As Any use ByVal when passing a string |
| LPWORD   | variable as Integer                             |
| LPRESULT | ByVal variable as Long                          |
| NULL     | ByVal Nothing or ByVal 0& or vbNullString       |
| SHORT    | ByVal variable as Integer                       |
| VOID     | Sub Procedure not applicable                    |
| WORD     | ByVal variable as Integer                       |
| WPARAM   | ByVal variable as Long                          |

We're not quite ready to get into using the API. Here is a scattering of issues/comments about using API which you will want to be aware of:

- **Declare**

- DECLARE in standard module are PUBLIC by default and be used anywhere in your app
- DECLARE in any other module are PRIVATE to that module and MUST BE marked PRIVATE
- Procedure names are CASE-SENSITIVE
- You cannot Declare a 16-bit API function in VB6

- **ALIAS**

- Is the "real" name of the procedure as found in the DLL
- If the API uses string, you MUST use ALIAS with "A" to specify the correct character set (A=ANSI W=UNICODE)
- WinNT supports W, but Win95/Win98 do not
- Some DLLs have illegal VB name, so you must use ALIAS to rename the procedure
- Can also be the ordinal number of the procedure

- **Variable Type**

- Very few DLLs recognize VARIANT

- 
- ByRef is VB default
  - Most DLLs expect ByVal
  - In C documentation, C passes all arguments except arrays by value
  - AS ANY can be used but it turns off all type checking
  - **Strings**
    - API generally require fixed length strings
    - Pass string ByVal means passing pointer to first data byte in the string
    - Pass string ByRef means passing memory address to another memory address which refers to first data byte in the string
    - Most DLLs expect LPSTR (ASCIIZ) strings (end in a null character), which point to the first data byte
    - VB Strings should be passed ByVal (in general)
    - VB uses BSTR strings (header + data bytes) - BSTR is passed as a pointer to the header
    - DLL can modify data in a string variable that it receives as an argument - WARNING: if returned value is longer than passed value, system error occurs!
    - Generally, API do not expect string buffers longer than 255 characters
    - C & VB both treat a string array as an array of pointers to string data
    - Most API require you to pass the length of the string and to fill the string with spaces
  - **Arrays**
    - Pass entire array by passing the first element of the array ByRef
    - Pass individual elements of array just like any other variable
    - If pass binary data to DLL, use array of Byte characters
  - **Callback Function**
    - Use AddressOf to pass a user-defined function that the DLL procedure can use
    - Must have specific set of arguments, AS DEFINED by the API procedure
    - Procedure MUST be in a .BAS module
    - Passed procedure must be As Any or As Long
  - **Passing a null value**
    - To pass a null value - zero-length string ("") will not work
    - To pass a null value - use vbNullString

- 
- To pass a null value - change Type to Long and then use 0&
  - **Window Handle**
    - A handle is simply a number assigned by Windows to each window
    - In VB, the handle is the same as the property hWnd
    - Handles are always Long variable types
  - **Callbacks**
    - Some API can run one of your own VB functions. Your VB function is called a "Callback"
    - VB supports callbacks with a function "AddressOf", which give the API the location of the function to execute
    - Callback functions must be in a module. They cannot be in a form.
  - **Subclassing**
    - All windows work by processing messages from the Windows operating system
    - You can change how a window responds to a message by intercepting the message
    - To intercept a message, use the API SetWindowsLong
  - **Miscellaneous**
    - Control properties MUST be passed by value (use intermediate value to pass ByRef)
    - Handles - always declare as ByVal Long
    - Variant - to pass Variant to argument that is not a Variant type, pass the Variant data ByVal
    - UDT - cannot be passed except as ByRef

### **Which API Should I Use?**

Finally we get to the good part. First the bad news, then the good news. In this section I do not provide code that you can simply copy into your own applications. The good news is that I provide a list of features that you might want to incorporate into your own application and then tell you which of the API to use. For the purposes of this relatively short tutorial, the best I can do is to point you off in the right direction!

In case you don't know, VB6 comes with a tool to help you use API in your own applications.

The **API Viewer** is installed automatically with VB, and to use it go to the

Start/Programs/VB/Tools menu and select "API Viewer". The viewer actions much like my own **VB Information Center Code Librarian** in that you can browse through the various API, select one for copying to the clipboard, and then paste the declaration into your own application's code window. You'll definitely want to try this out. The data file that comes with the viewer is very extensive, listing 1550 API Declarations.

In my case I use API regularly, but I've never come close to using 1550 API. At best, I barely have broken the 100 mark. It seems that for the most part I can get VB to do whatever task I want without resorting to the API. However, in some cases you just can do any better than a few

---

lines of API code to get the job done! So, here's my own list of useful tasks and the API needed to perform them:

|                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Play sound</b>                 | Declare Function sndPlaySound Lib "winmm.dll" Alias "sndPlaySoundA" (ByVal lpzSoundName as string, ByVal uFlags as Long) as Long<br>Result = sndPlaySound (SoundFile, 1)                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>SubClassing</b>                | 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<br>Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA" (ByVal hwnd As Long, ByVal nIndex As Long, ByVal dwNewLong As Long) As Long                                                                                                                                                                                                          |
| <b>Run associated EXE</b>         | Declare Function ShellExecute Lib "shell32.dll" Alias "ShellExecuteA" (ByVal hwnd As Long, ByVal lpOperation As String, ByVal lpFile As String, ByVal lpParameters As String, ByVal lpDirectory As String, ByVal nShowCmd As Long) As Long                                                                                                                                                                                                                                                                                                                |
| <b>List window handles</b>        | Declare Function EnumWindows Lib "user32" (ByVal lpEnumFunc As Long, ByVal lParam As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Find prior instance of EXE</b> | Declare Function FindWindow Lib "user32" Alias "FindWindowA" (ByVal lpClassName As String, ByVal lpWindowName As String) As Long                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Draw dotted rectangle</b>      | Declare Function DrawFocusRect Lib "user32" Alias "DrawFocusRect" (ByVal hdc As Long, lpRect As RECT) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Invert colors of rectangle</b> | Declare Function InvertRect Lib "user32" Alias "InvertRect" (ByVal hdc As Long, lpRect As RECT) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Get cursor position</b>        | Declare Function GetCursorPos Lib "user32" Alias "GetCursorPos" (lpPoint As POINTAPI) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Always on top</b>              | Declare Function SetWindowPos Lib "user32" Alias "SetWindowPos" (ByVal hwnd As Long, ByVal hWndInsertAfter As Long, ByVal x As Long, ByVal y As Long, ByVal cx As Long, ByVal cy As Long, ByVal wFlags As Long) As Long                                                                                                                                                                                                                                                                                                                                   |
| <b>Send messages to a window</b>  | Declare Function SendMessage Lib "user32" Alias "SendMessageA" (ByVal hwnd As Long, ByVal wParam As Long, ByVal lParam As Long, ByVal wMsg As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Find directories</b>           | Declare Function GetWindowsDirectory Lib "kernel32" Alias "GetWindowsDirectoryA" (ByVal lpBuffer As String, ByVal nSize As Long) As Long<br>Declare Function GetSystemDirectory Lib "kernel32" Alias "GetSystemDirectoryA" (ByVal lpBuffer As String, ByVal nSize As Long) As Long<br>Declare Function GetTempPath Lib "kernel32" Alias "GetTempPathA" (ByVal nBufferLength As Long, ByVal lpBuffer As String) As Long<br>Declare Function GetCurrentDirectory Lib "kernel32" Alias "GetCurrentDirectory" (ByVal nBufferLength As Long, ByVal lpBuffer As |

---

|                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                               | String) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Text alignment</b>         | Declare Function GetTextAlign Lib "gdi32" Alias "GetTextAlign" (ByVal hdc As Long) As Long<br>Declare Function SetTextAlign Lib "gdi32" Alias "SetTextAlign" (ByVal hdc As Long, ByVal wFlags As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Flash a title bar</b>      | Declare Function FlashWindow Lib "user32" Alias "FlashWindow" (ByVal hwnd As Long, ByVal bInvert As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Manipulate bitmaps</b>     | Declare Function BitBlt Lib "gdi32" Alias "BitBlt" (ByVal hDestDC As Long, ByVal x As Long, ByVal y As Long, ByVal nWidth As Long, ByVal nHeight As Long, ByVal hSrcDC As Long, ByVal xSrc As Long, ByVal ySrc As Long, ByVal dwRop As Long) As Long<br>Declare Function PatBlt Lib "gdi32" Alias "PatBlt" (ByVal hdc As Long, ByVal x As Long, ByVal y As Long, ByVal nWidth As Long, ByVal nHeight As Long, ByVal dwRop As Long) As Long<br>Declare Function StretchBlt Lib "gdi32" Alias "StretchBlt" (ByVal hdc As Long, ByVal x As Long, ByVal y As Long, ByVal nWidth As Long, ByVal nHeight As Long, ByVal hSrcDC As Long, ByVal xSrc As Long, ByVal ySrc As Long, ByVal nSrcWidth As Long, ByVal nSrcHeight As Long, ByVal dwRop As Long) As Long<br>Declare Function CreateCompatibleBitmap Lib "gdi32" Alias "CreateCompatibleBitmap" (ByVal hdc As Long, ByVal nWidth As Long, ByVal nHeight As Long) As Long<br>Declare Function CreateCompatibleDC Lib "gdi32" Alias "CreateCompatibleDC" (ByVal hdc As Long) As Long |
| <b>Rotate text</b>            | Declare Function CreateFontIndirect Lib "gdi32" Alias "CreateFontIndirectA" (lpLogFont As LOGFONT) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Timing</b>                 | Declare Function GetTickCount Lib "kernel32" Alias "GetTickCount" () As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>File information</b>       | Declare Function GetFileAttributes Lib "kernel32" Alias "GetFileAttributesA" (ByVal lpFileName As String) As Long<br>Declare Function GetFileSize Lib "kernel32" Alias "GetFileSize" (ByVal hFile As Long, lpFileSizeHigh As Long) As Long<br>Declare Function GetFullPathName Lib "kernel32" Alias "GetFullPathNameA" (ByVal lpFileName As String, ByVal nBufferLength As Long, ByVal lpBuffer As String, ByVal lpFilePart As String) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Get window information</b> | Declare Function GetClassName Lib "user32" Alias "GetClassNameA" (ByVal hwnd As Long, ByVal lpClassName As String, ByVal nMaxCount As Long) As Long<br>Declare Function GetWindowText Lib "user32" Alias "GetWindowTextA" (ByVal hwnd As Long, ByVal lpString As String, ByVal cch As Long) As Long<br>Declare Function GetParent Lib "user32" Alias "GetParent" (ByVal hwnd As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Identify window</b>        | Declare Function WindowFromPoint Lib "user32" Alias "WindowFromPoint"                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |

---

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>at cursor</b>         | (ByVal xPoint As Long, ByVal yPoint As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Registry editing</b>  | Declare Function RegCreateKey Lib "advapi32.dll" Alias "RegCreateKeyA" (ByVal hKey As Long, ByVal lpSubKey As String, phkResult As Long) As Long<br>Declare Function RegDeleteKey Lib "advapi32.dll" Alias "RegDeleteKeyA" (ByVal hKey As Long, ByVal lpSubKey As String) As Long<br>Declare Function RegDeleteValue Lib "advapi32.dll" Alias "RegDeleteValueA" (ByVal hKey As Long, ByVal lpValueName As String) As Long<br>Declare Function RegQueryValueEx Lib "advapi32.dll" Alias "RegQueryValueExA" (ByVal hKey As Long, ByVal lpValueName As String, ByVal lpReserved As Long, lpType As Long, lpData As Any, lpcbData As Long) As Long<br>Declare Function RegSetValueEx Lib "advapi32.dll" Alias "RegSetValueExA" (ByVal hKey As Long, ByVal lpValueName As String, ByVal Reserved As Long, ByVal dwType As Long, lpData As Any, ByVal cbData As Long) As Long |
| <b>Drawing functions</b> | Declare Function MoveToEx Lib "gdi32" Alias "MoveToEx" (ByVal hdc As Long, ByVal x As Long, ByVal y As Long, lpPoint As POINTAPI) As Long<br>Declare Function LineTo Lib "gdi32" Alias "LineTo" (ByVal hdc As Long, ByVal x As Long, ByVal y As Long) As Long<br>Declare Function Ellipse Lib "gdi32" Alias "Ellipse" (ByVal hdc As Long, ByVal X1 As Long, ByVal Y1 As Long, ByVal X2 As Long, ByVal Y2 As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Get icon Declare</b>  | Function ExtractIcon Lib "shell32.dll" Alias "ExtractIconA" (ByVal hInst As Long, ByVal lpszExeFileName As String, ByVal nIconIndex As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Screen capture</b>    | Declare Function SetCapture Lib "user32" Alias "SetCapture" (ByVal hwnd As Long) As Long<br>Declare Function CreateDC Lib "gdi32" Alias "CreateDCA" (ByVal lpDriverName As String, ByVal lpDeviceName As String, ByVal lpOutput As String, lpInitData As DEVMODE) As Long<br>Declare Function DeleteDC Lib "gdi32" Alias "DeleteDC" (ByVal hdc As Long) As Long<br>Declare Function BitBlt Lib "gdi32" Alias "BitBlt" (ByVal hDestDC As Long, ByVal x As Long, ByVal y As Long, ByVal nWidth As Long, ByVal nHeight As Long, ByVal hSrcDC As Long, ByVal xSrc As Long, ByVal ySrc As Long, ByVal dwRop As Long) As Long<br>Declare Function ReleaseCapture Lib "user32" Alias "ReleaseCapture" () As Long<br>Declare Function ClientToScreen Lib "user32" Alias "ClientToScreen" (ByVal hwnd As Long, lpPoint As POINTAPI) As Long                                      |
| <b>Get user name</b>     | Declare Function GetUserName Lib "advapi32.dll" Alias "GetUserNameA" (ByVal lpBuffer As String, nSize As Long) As Long<br>Declare Function GetUserName Lib "advapi32.dll" Alias "GetUserNameA" (ByVal lpBuffer As String, nSize As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

|                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Get computer name</b>       | Declare Function GetComputerName Lib "kernel32" Alias "GetComputerNameA" (ByVal lpBuffer As String, nSize As Long) As Long<br>Declare Function GetComputerName Lib "kernel32" Alias "GetComputerNameA" (ByVal lpBuffer As String, nSize As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Get volume name/serial#</b> | Declare Function GetVolumeInformation Lib "kernel32" Alias "GetVolumeInformationA" (ByVal lpRootPathName As String, ByVal lpVolumeNameBuffer As String, ByVal nVolumeNameSize As Long, lpVolumeSerialNumber As Long, lpMaximumComponentLength As Long, lpFileSystemFlags As Long, ByVal lpFileSystemNameBuffer As String, ByVal nFileSystemNameSize As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Identify drive type</b>     | Declare Function GetDriveType Lib "kernel32" Alias "GetDriveTypeA" (ByVal nDrive As String) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Get free space</b>          | Declare Function GetDiskFreeSpace Lib "kernel32" Alias "GetDiskFreeSpaceA" (ByVal lpRootPathName As String, lpSectorsPerCluster As Long, lpBytesPerSector As Long, lpNumberOfFreeClusters As Long, lpTotalNumberOfClusters As Long) As Long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>INI editing</b>             | Declare Function WritePrivateProfileSection Lib "kernel32" Alias "WritePrivateProfileSectionA" (ByVal lpAppName As String, ByVal lpString As String, ByVal lpFileName As String) As Long<br>Declare Function WritePrivateProfileString Lib "kernel32" Alias "WritePrivateProfileStringA" (ByVal lpApplicationName As String, ByVal lpKeyName As Any, ByVal lpString As Any, ByVal lpFileName As String) As Long<br>Declare Function GetPrivateProfileInt Lib "kernel32" Alias "GetPrivateProfileIntA" (ByVal lpApplicationName As String, ByVal lpKeyName As String, ByVal nDefault As Long, ByVal lpFileName As String) As Long<br>Declare Function GetPrivateProfileSection Lib "kernel32" Alias "GetPrivateProfileSectionA" (ByVal lpAppName As String, ByVal lpReturnedString As String, ByVal nSize As Long, ByVal lpFileName As String) As Long<br>Declare Function GetPrivateProfileString Lib "kernel32" Alias "GetPrivateProfileStringA" (ByVal lpApplicationName As String, ByVal lpKeyName As Any, ByVal lpDefault As String, ByVal lpReturnedString As String, ByVal nSize As Long, ByVal lpFileName As String) As Long |
| <b>Put icon in system tray</b> | 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<br>Declare Function GetWindowLong Lib "user32" Alias "GetWindowLongA" (ByVal hwnd As Long, ByVal nIndex As Long) As Long<br>Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA" (ByVal hwnd As Long, ByVal nIndex As Long, ByVal dwNewLong As Long) As Long<br>Declare Function Shell_NotifyIcon Lib "shell32.dll" Alias "Shell_NotifyIconA" (ByVal dwMessage As Long, lpData As NOTIFYICONDATA) As Long<br>Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory"                                                                                                                                                                                                                                                                                                                                                                                                                                                |

---

(Destination As Any, Source As Any, ByVal Length As Long)  
Declare Function DrawEdge Lib "user32" Alias "DrawEdge" (ByVal hdc As Long, qrc As RECT, ByVal edge As Long, ByVal grfFlags As Long) As Long

**Wait for program to stop** Declare Function CreateProcess Lib "kernel32" Alias "CreateProcessA" (ByVal lpApplicationName As String, ByVal lpCommandLine As String, lpProcessAttributes As SECURITY\_ATTRIBUTES, lpThreadAttributes As SECURITY\_ATTRIBUTES, ByVal bInheritHandles As Long, ByVal dwCreationFlags As Long, lpEnvironment As Any, ByVal lpCurrentDirectory As String, lpStartupInfo As STARTUPINFO, lpProcessInformation As PROCESS\_INFORMATION) As Long  
Declare Function WaitForSingleObject Lib "kernel32" Alias "WaitForSingleObject" (ByVal hHandle As Long, ByVal dwMilliseconds As Long) As Long

**Stop ctrl-alt-del** Declare Function SystemParametersInfo Lib "user32" Alias "SystemParametersInfoA" (ByVal uAction As Long, ByVal uParam As Long, ByRef lpvParam As Any, ByVal fuWinIni As Long) As Long

Hopefully, this section of the tutorial has sparked some excitement! You should now see that a door of tremendous proportions has been opened to you. You've begun to leave the limitations of VB behind and joined the rest of the programming community who have already been using the API for years. I hope to add quite a bit to this tutorial section so check back often over the next few weeks.

## Objects

Since VB4, Microsoft has been promoting the "object-oriented" aspects of Visual Basic. It seems like everything you read is object-this and object-that! So what is the real story about objects? Who needs them, and why? How do you use them and when are they overkill? Can you use them without having to be an expert in all aspects of objects? If you're like me, a solo programmer whose contact with the VB world is limited to email and magazines, do the answers apply to you? I'm interested in writing the tutorial as much to find out the answers myself, as you are to get the answers!

---

## Introduction to Objects

Let's get down the bottom line quickly. **Objects are a combination of code and data which you can treat as a single item.** You have already worked with objects - the controls and forms in a VB project. So the purpose of this tutorial is not so much to explain what they are, but rather to go into the details of how they work and how you can use them in your own programs. There is a lot more to objects than just drawing a control on your form and that's what we want to cover here. We'll especially get into how you create and handle objects in code, which is **much different** than simply drawing controls onto a form!

In case you have any question about the value of objects, let me state without reservation, that **objects (in the form of controls) are the single biggest reason why VB has been so successful.** In more recent versions of VB, Microsoft has expanded the object features of VB, making it even more powerful than ever.

The availability of powerful tools does not automatically guarantee success. From my viewpoint, the jury is still out on how useful these newer features are going to be for the average programmer! I have no doubts about the power of the newer tools, but just like 70% of Excel/Word capabilities are never applied by users, so too are many of VB's object-oriented features going unused by most programmers.

VB is known for its ability to allow a programmer to quickly, and easily, create a Windows program. In part, this is possible because **objects provide code which you don't have to write!**

---

I cannot say this with enough force! Objects are written by programmers to provide certain capabilities which you can access by executing the code with the objects. In a way, this makes objects just another way of creating a code library - similar in function to the use of API that you learned in the last tutorial!

However, don't mistake my simple comparison to imply that objects are as easy to use as are the API. They're not! You'll spend a lot more time studying how to work with objects. Mastering how to use API is a one-day session, with additional time spent on figuring out which API to use. Learning to master the use of objects, however, is going to be one of the major study tasks that you undertake in VB.

Some parts of objects usage are actually pretty simple. Some, however, are very complex. In this tutorial we'll work our way up from the simple to the difficult.

You'll have to make a personal assessment as to the extent to which you implement the range of object-oriented capabilities that VB offers. You may find that your applications work fine with just the use of the basic controls that come with VB or by using the Windows API when VB falls short. You may also find that you cannot get the functionality from you want from your application or that you cannot reach the level of efficiency at creating applications without depending heavily on objects.

In the end, I suggest that the criteria for using objects is whether they help you get the job done with less effort than it would take to do the job without them! My personal experience is that limited application of object programming is adequate to do just about anything I need to do. I'll admit that I'm having trouble relating my own experience with the object-intensive media blitz that you see in the VB magazines today.

There's an old saying that to someone with a hammer, everything looks like a nail! The moral to the story is that you should develop skills with new tools in order to evaluate which one applies to certain jobs. Let's begin that development right now!

### **What are Objects?**

I gave a definition earlier - objects are a combination of code and data. But let's expand on the idea. Equally key is the idea that the code/data combination can be handled as a single item. If you copy the object, both the code and data go with it. If you save the object, both the code and data get saved.

Examples are an easy way for you to understand what VB treats as object:

- Command Button - In fact, any of the VB controls are objects.
- Form - Just like controls, forms are objects.
- Database - Even though it doesn't have a windows, like a control, a database is still an object.

These are all examples of objects to which you should have already been exposed as part of your VB training.

### **Properties, Methods, and Events**

Later, we'll get more into creating your own objects. But for now, let's look at how to use them. All objects share a common way of accessing the data and code that we've said make up an object.

The data inside an object is known as the **properties** of the object. In familiar terms, it corresponds to the values of variables which are stored inside the object. For example, all objects have names. The name is one of the properties of the object and the way that you access a property in code is as follows:

```
tempString = ObjectName.Name
```

In this simple example, a variable called "tempString" is filled with the Name of an object. All properties of objects work exactly the same way, using what is called the "dot" notation. Here are some more examples:

```
h = Form1.height
```

---

```
w = listbox1.fontname
r = textbox.backcolor
g = timer1.interval
```

In each of these examples, a variable on the left is assigned the value of the property on the right. The format for identifying the property name is also done the same in every case - "objectname.propertyname".

Typically, whoever creates the object "exposes" properties which you can access. In the VB IDE environment you can click on a control and the properties box (press F4) will display all of the property names along with possible values for those properties. Object properties are just like variable in that they can have a type. They can be integer, string, dates, or any other VB data type. So remember when assigning property values to variables that the VB rules for type conversion must be taken into consideration.

When in the IDE code windows, VB also provides a big help in remembering the properties of an object. Once you've type in the object name and the ".", VB will display a dropdown listbox which give all of the available properties.

Two other nuances about properties will be of interest. First of all, not all properties of an object are "exposed" for use in your VB program. When we get around to creating our own objects in code, you'll find that you can choose whether to expose a property for use in code, or whether to keep it hidden within the object (so as not to allow its value to get changed).

Also, ... (author mind block - as soon as I remember the other nuance, I'll put it here ).

### COM & Exposed Objects

One of the key features about objects is that they can be made available **outside** of an application for use by other programmers. Making an object available for use is known as "exposing" the object. Many of the major applications such as Word or Excel expose objects for your use. Microsoft has established a standard known as COM (Common Object Model) which guides programmers through the techniques of exposing objects from within their applications.

Exposed objects **cannot**, however, be distributed with your application. If you plan to create an application which uses exposed objects from a second application, then your user must have that second application installed on his PC. This is **not a trivial** problem. Unless you have some control over the user's environment you'll find it best to distribute stand-alone applications - ones which do not assume that any particular software is resident on your user's PCs.

All controls, whether in the form of ActiveX Controls (.OCX files) or in the form of exposed objects, must be registered with Windows (as part of the Registry) before your VB program can use the object. In the VB IDE you can call up the Components Dialog Box which will list all of the objects (ActiveX files or exposed objects) that are available for your use. If you select one of the objects for use, the object will appear on your VB IDE toolbox and can then be used just like any of the intrinsic controls that you have already used.

There is a utility called **regsvr32.exe** that comes with Windows which you will find very useful. Regsvr32 allows you to manually register an OCX in the registry. The need to do this manually can arise for a variety of reasons, but mostly because an installation program failed to automatically make the registration. The code to register an OCX file is:

```
regsvr32 filename.ocx
```

You can do this from a DOS command line or through the Start/Run menu option.

### Classes

We're just about to get into the How of creating objects, but one more topic needs discussion first. Before an object can be created, an object template must first be created. Once that template is created a copy can be created. It's the copy with which you will work in your VB programs.

In VB terminology, the template is called a **class** and the copies of the template are called **instances**. In the registration process that we just discussed, it is the template, or class, of an object that gets registered with Windows.

### Object Browser

VB provides a tool with which you can browse all registered objects, including the properties and methods which they expose. The Object Browser window can be displayed by pressing F2

---

within the VB IDE. One advantage of the Object Browser over the Properties window is that the browser can show both properties and methods of the registered classes. Additionally, the browser shows all registered objects, **including** the objects exposed by applications that you have installed on your PC. It's the poor man's documentation of all usable classes on your PC!

### Creating Objects - References, Class Modules, & Collections

You can create objects at both design time and run time. Controls, forms, and class modules are examples of objects which you can create at design time (within the VB IDE). You can also create controls and form objects at run time, as well as creating variable objects and collections at run time. We'll cover these topics next in the tutorial.

- **References**

It is possible to use the DIM function to create a variable whose type is that of any registered class. To do so, use this code:

- `Dim X as ListBox`

Variables which have been defined as an object type are useful because they can be made to refer to an existing object. For example, this code, when used with the line of code just given, associates X with the ListBox1 control.

```
Set X = ListBox1
```

Notice the use of the Set statement? To establish a variable reference to an object, **you cannot** simply use the equal sign. You must use the SET statement.

There are several reasons why you should be interested in creating a second reference to a control.

First of all, once the variable is set to reference the control, all of the control's properties and methods can be accessed as follows:

```
X.name
is the same as
ListBox1.name
```

If nothing else, using variables rather than the full name of the control can save you some typing as you write your code.

However there are other, more pressing reasons why the use of object variables in code is valuable. Later, we'll get into the idea of Collections, which represent a way to group objects so that you can address them one by one in code. It's similar to the concept of an array of objects, but the code for accessing a collection is different. I bring it up now because the following code for walking through a collection of objects demonstrates why the use of object variables is so valuable in VB:

```
Dim X as Control
For Each X in Form1.Controls
 If X.Top > 500 then
 MsgBox "It is below the line"
 End If
Next X
```

What this code does is to walk through all controls on Form1 (the Controls collection) and see if the top of the control is below the position 500. The magic to this code is that the object variable X was successively assigned a reference to every control on the form, and we could access the Top property of each control simply by using the variable X!

Otherwise we would have had to put code which specifically named every single control and evaluated its Top property. The use of an object variable made it possible to significantly reduce the amount of coding that I had to do!

There is a variation on the use of the Set statement which you should learn, and that is the Keyword "New". Elsewhere in these tutorials we

- **Collections**

VB also supports an object known as a collection. A collection object is exactly what the name implies - a group of objects whose properties can be addressed through code. And

---

as the example above showed, it is not necessary to code the individual names of each member of the collection.

VB provides the fairly simple means of working with collections. A collection has only one property, which is "Count", the number of objects in the collection. When an object is added to a collection it is given an index, with the first object index starting at zero. To address an object in a collection you can use either its name, or its index as follows:

- Specify the name of the object:
  - Controls("List1")
  - Controls!List1
- Specify the index number of the object
  - Controls(3)

A collection also has only three methods:

- **Add**  
Allows you to add members to a collection. Optional. One thing to note is that when you add a member to a collection you can give it a unique string (a key) that can be used, instead of a positional index, to access that member in the future.
- **Remove**  
Allows you to remove members from a collection, either by specifying the index number of the object or its key.
- **Item**  
Returns a specific member of a collection, either by number or by key.

Finally the loop construct I gave above is worth repeating because you can now see how it applies to collections.

```
Dim X as Control
For Each X in Form1.Controls
 If X.Top > 500 then
 MsgBox "It is below the line"
 End If
Next X
```

The "For Each" structure provides the ability to walk through every member of a collection ("Controls" in this example) and then operate on each member as your code/application requires.

As you'll see collections have become an integral and important part of the VB coding environment. The example I just gave shows that using a collection (VB built-in collections managed by VB or collections which you can create yourself) can save you a lot of time and effort.

The old Texas saying "If you don't own an oil well, then get one!" very much applies to collections. Not only can you save coding time and effort as I showed in the example, but you'll find that even the controls which come with VB contain collections of their own. Whether you want to or not, collections are one area where you'll be forced into learning about them simply because they are becoming an integral part of the daily controls that you use.

- **Class Modules**

What we've discussed so far was perhaps the easier part of the tutorial. The real power that Microsoft has added with VB5 and VB6 has to do with the ability to create your own objects, including objects which you can expose to other applications and which you can compile separately and re-use in other applications.