

MOMBASA TECHNICAL TRAINING INSTITUTE



**DEPARTMENT OF INFORMATION AND COMMUNICATION
TECHNOLOGY**

VISUAL PROGRAMMING

P.O. BOX 81220-80100, TEL 041-2226458, MOMBASA

Whatsapp 0726 733 444

TABLE OF CONTENTS

TABLE OF CONTENTS	2
INTRODUCTION TO VISUAL PROGRAMMING	5
Event-Driven Programming	5
Advantages of Visual Programming	5
Disadvantages of Visual Programming	5
VISUAL BASIC 6.0	6
System Requirements for Visual Basic 6.0	6
Properties, Methods and Events	6
VISUAL ENVIRONMENT - The Integrated Development Environment.....	7
Toolbox	8
Project Explorer	10
Properties Window	10
Object Browser	10
Anchoring and Docking	12
FORM LAYOUT WINDOW	12
CONTROLS IN VISUAL BASIC	14
Label and Frame Controls in Visual Basic 6 (VB6)	14
Frame Controls	14
IMAGE CONTROLS IN VB	15
CommandButton and OptionButton Controls	17
CommandButton Controls in VB6	17
OptionButton Controls in VB6	18
The Timer, Line, Shape and OLE Controls	20
The Timer Control	20
The Line Control	21
The Shape Control	22
The OLE Control	22
ListBox and ComboBox Controls	22
VB ScrollBar	28
DriveListBox, DirListBox, and FileListBox Controls	30
Using a CheckBox control in Visual Basic 6	32
TabIndex property of Controls	34
Working with Forms in Visual Basic 6	35
Difference between Unload and Hide method	38
Control Arrays in Visual Basic 6	39
Working with Menus in Visual Basic 6 (VB6)	41
The Multiple Document Interface (MDI) in Visual Basic 6	47
InputBox Function in Visual Basic 6 (VB6)	50
MessageBox Function in Visual Basic 6 (VB6)	52
VB Constants used together with MsgBox Function	54
PROGRAM STRUCTURE	57
Object Naming Conventions of Controls (Prefixes)	57
Data Types, Modules and Operators	57
Modules	57
Operators in Visual Basic	59
Variables in Visual Basic 6	61

Explicit Declaration	61
Scope of variables.....	62
CONTROL STRUCTURES IN VISUAL BASIC 6.0	65
Types of Control Structures	65
Selection Structures/Decision Structures	65
If...Then selection structure.....	65
If...Then...Else selection structure	66
If...ElseIf...Else selection structure.....	67
Select...Case selection structure.....	68
Loops (Repetition Structures) in Visual Basic 6	69
Do While... Loop Statement.....	70
While... Wend Statement.....	70
Do...Loop While Statement.....	70
Do Until...Loop Statement	71
The For...Next Loop	71
The For Each...Next Loop	72
Exit For and Exit Do Statement in Visual basic 6	72
With...End With statement.....	73
DATA STRUCTURES	74
1. Arrays	74
Fixed-sized (Static) Arrays	75
Variable-Length (Dynamic) Arrays	76
Multidimensional Arrays	77
Searching Arrays	78
Sorting Arrays	81
Control Arrays	84
2. User-Defined Data Types in Visual Basic 6.....	84
3. ENUMERATIONS (Enums)	85
Constants, Data Type Conversion, Visual Basic Built-in Functions.....	86
Data Type Conversion.....	87
Date and Time Functions in Visual Basic 6	88
SUB-PROGRAMS	90
Types of Sub-Procedures.....	91
Function Procedures.....	92
Property Procedures.....	93
Arguments/Parameters	93
ERROR HANDLING AND DEBUGGING	97
Types of Programming Errors	97
Run-Time Error Trapping and Handling.....	99
Eliminating Logic Errors	101
Using the Debugging Toolbar	101
SEQUENTIAL FILES.....	109
Writing to Sequential Files	109
Reading From Sequential Files	110
Sequential File-Related Statements and Functions	110
Comma-Delimited Format.....	111
VB Statements and Functions for Sequential File Processing	111
Using App.Path.....	112
Using FreeFile.....	113
LINKING TO DATABASES	117

ADO (ActiveX Data Object) data control	117
Data Bound Tools	119
ADO CURSORS AND LOCKS	120
Cursor Types.....	122
Lock Types	123
Data Manager in Visual Basic 6 (VB6)	124
Example - Accessing the Books Database	126
Creating a Data Report in Visual Basic 6 (VB6).....	128
Creating a Data Report	130
STRING FUNCTIONS	133
Converting Strings.....	134
The ASCII Functions	135
The Substring Functions Practical.....	135
Converting to Uppercase and Lowercase	137
Counting Number of Words In A String	138

Whatsapp 0726 733 444

INTRODUCTION TO VISUAL PROGRAMMING

Visual programming is the creation of programs by manipulating program elements graphically, partially or completely, rather than by specifying them textually.

Examples of visual programming languages (VPL) are **PowerBuilder, Developer 2000, Labview and Microsoft Visual Basic 6.0.**

Programming in VB is a combination of visually arranging components or controls on a form, specifying attributes and actions for those components, and writing additional lines of code for more functionality.

A visual programming environment provides all the features that are required to develop a graphical user interface as ready-to-use components. The programmer does not have to write code to create and display the commonly used components.

Event-Driven Programming

Visual Basic programs are built around events. Events are various things that can happen in a program. In procedural languages, an application is written and executed logically through the program statements, one after another. In an **event driven application, the program statements are executed only when a particular event calls a specific part of the code that is assigned to the event.**

The execution of statements in an event driven program does not follow a predetermined path. Events can be triggered by the user's actions, messages from the system or other applications. Programs thereof are highly suitable for Graphical User Interfaces. Such languages are simple and flexible to program in.

Advantages of Visual Programming.

Visual programming enables:

1. Visual development of graphical user interface which are easy to use and to learn.
2. It is less time consuming by providing ready to use components. The programmer only needs to drag the required components from a toolbox and drop them on a form.
3. The components can be moved, resized and even deleted with ease, if so required.
4. There is no restriction on the number of controls that can be placed on a form.
5. The programs developed are visually appealing.

Disadvantages of Visual Programming

1. The development environment in itself is highly graphical in nature and therefore requires higher system specifications i.e.
 - Larger capacity hard disk.
 - More RAM.
 - Faster Processors
 - GPU- Graphics Processing Unit
2. Visual programming can be done only with GUI operating systems such as Windows.
3. The programs are also slow to execute due to the graphics involved.

VISUAL BASIC 6.0

Visual Basic is initiated by using the Programs option > Microsoft Visual Basic 6.0 > Visual Basic 6.0. Then it opens in to a new screen with the interface elements Such as MenuBar, ToolBar, New Project dialog box. These elements permit the user to buid different types of Visual Basic applications.

System Requirements for Visual Basic 6.0

The following hardware and software is required for Visual Basic applications:

- Pentium® 90MHz or higher microprocessor.
- VGA 640x480 or higher-resolution screen supported by Microsoft Windows.
- Microsoft Windows NT 4.0 or later or Microsoft Windows 95 or later.
- 24 MB RAM for Windows 95/98, 32 MB for Windows NT or later.
- Disk space requirements:
 - Standard Edition: typical installation 48 MB, full installation 80 MB.
 - Professional Edition: typical installation 48 MB, full installation 80 MB.
 - Enterprise Edition: typical installation 128 MB, full installation 147 MB.
 - Additional components (if required): MSDN (for documentation): 67 MB, Internet Explorer 4.x: approximately 66 MB.
- CD-ROM.

Properties, Methods and Events

All the controls in the ToolBox except the Pointer are objects in Visual Basic. These objects have associated properties, methods and events.

1. Real world objects are loaded with properties. For example, a flower is loaded with certain color, shape and fragrance. Programming objects are loaded with properties. **A property is a named attribute of a programming object. Properties define the characteristics of an object such as Size, Color etc. or sometimes the way in which it behaves.** For example, a TextBox accepts properties such as Enabled, Font, MultiLine, Text, Visible, Width, etc.

Some properties are **design-time** that can be set at design time by selecting the Properties Window. But certain properties cannot be set at design time, only at **run-time**. For example, the name, CurrentX and CurrentY properties of a Form cannot be set at the design time.

2. **A method is an action that an object can perform.** It can also be defined as **a procedure that acts on an object.** For example, **AddItem** is a method of the **ComboBox** object that adds a new entry to a combo box. Others include **Clear, Drag, Move, and Refresh.** For example, a cat is an object. Its properties might include long white hair, blue eyes, 3 pounds weight etc. A complete definition of cat must not only encompass its looks, but should also include a complete itemization of its activities. Therefore, a cat's methods might be move, jump, play, breath etc.

The TextBox control has associated methods such as Refresh, SetFocus, etc.

- The Refresh method enforces a complete repaint of the control or a Form. For example, Text1.Refresh refreshes the TextBox.
- The Setfocus method moves the focus on the control. For Example Text1.SetFocus sets the focus to TextBox control Text1.

Some more methods

Method	Action
AddItem	To add new item in a list or a combo box
Drag	To respond to the drag-and-drop action by the user
Hide	To hide a form

Move	To change an object's position in response to certain events
Print	To display text on an object, such as a form.
SetFocus	To shift focus to an object
Show	To display a form

The **Unload** statement removes the form from memory. In most simple VB 6 projects, Form1 is the startup object so the program stops running too.

The **Hide** method only removes the form from the screen. Nothing else changes. For example, another process can still communicate with objects on the form after the Hide method is called.

3. An **event** is an activity that occurs during a program's execution, such as a mouse click or a keystroke. **Event-driven programming applies to programming that responds to Windows events.** Events in visual programming are handled (respond to) using procedures.

Event	Description
Click	The user clicks the mouse button on an object.
DblClick	The user double-clicks the mouse button on an object.
DragDrop	The user drags an object to another location.
GotFocus	An object receives focus.
KeyDown	The user presses a keyboard key.
KeyPress	The user presses a keyboard key .
KeyUp	The user releases a keyboard key while an object has focus.
LostFocus	An object loses focus.
MouseDown	The user presses any mouse button.
MouseUp	The user releases any mouse button.

VISUAL ENVIRONMENT - The Integrated Development Environment

IDE is a term commonly used in the programming world to describe the **interface** and **environment** that we use to create our applications i.e. **tools to code, build/compile, test and debug applications**. It is called **integrated** because we can access virtually all of the development tools that we need from one screen called an **interface**. The IDE is also commonly referred to as the **design environment**, or the **program**.

The Visual Basic IDE is made up of a number of components

- Menu Bar
- Tool Bar
- Project Explorer
- Properties window
- Form Layout Window
- Toolbox
- Form Designer
- Object Browser

In previous versions of Visual Basic, the IDE was designed as a Single Document Interface (SDI). In a Single Document Interface, each window is a free-floating window that is contained within a main window and can move anywhere on the screen as long as Visual Basic is the current application. But, in Visual Basic 6.0, the IDE is in a Multiple Document Interface (MDI) format. In this format, the windows associated with the project will stay within a single container known as the parent. Code and form-based windows will stay within the main container form.

Figure 1 Visual Basic startup dialog box



Menu Bar

This Menu Bar **displays the commands** that are required to build an application. The main menu items have sub menu items that can be chosen when needed.

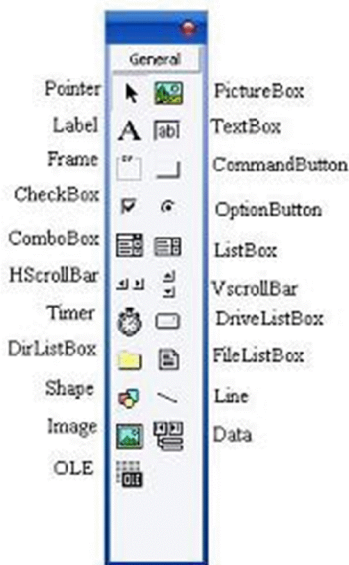
Toolbar

The buttons in the toolbar provide **quick access to the commonly used commands** and a button in the toolbar is clicked once to carry out the action represented by it. To find out what kind of control is displayed, position your mouse on a control and observe the tool tip.

Toolbox

The Toolbox contains a **set of controls** that can be placed on a Form at design time thereby creating the user interface area. Additional controls can be included in the toolbox by using the Components menu item on the Project menu. A Toolbox is represented in figure 2 shown below.

Figure 2 Toolbox window with its controls available commonly.



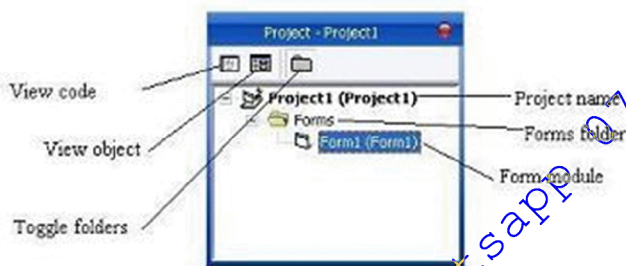
<u>Control</u>	<u>Description</u>
Pointer	Provides a way to move and resize the controls on the form
PictureBox	Displays icons/bitmaps and metafiles. It displays text or acts as a visual container for other controls.
TextBox	Used to display message and enter text.
Frame	Serves as a visual and functional container for controls
CommandButton	Used to carry out the specified action when the user chooses it.
CheckBox	Displays a True/False or Yes/No option.
OptionButton	OptionButton control which is a part of an option group allows the user to select only one option even it displays multiple choices.
ListBox	Displays a list of items from which a user can select one.
ComboBox	Contains a TextBox and a ListBox. This allows the user to select an item from the dropdown ListBox, or to type in a selection in the TextBox.
HScrollBar and VScrollBar	These controls allow the user to select a value within the specified range of values
Timer	Executes the timer events at specified intervals of time
DriveListBox	Displays the valid disk drives and allows the user to select one of them.
DirListBox	Allows the user to select the directories and paths, which are displayed.
FileListBox	Displays a set of files from which a user can select the desired one.
Shape	Used to add shape (rectangle, square or circle) to a Form
Line	Used to draw straight line to the Form

Image	Used to display images such as icons, bitmaps and metafiles. But less capability than the PictureBox
Data	Enables the user to connect to an existing database and display information from it.
OLE	Used to link or embed an object, display and manipulate data from other windows based applications.
Label	Displays a text that the user cannot modify or interact with.

Project Explorer

Docked on the right side of the screen, just under the toolbar, is the Project Explorer window. The window can also be in **floating mode**. **The Project Explorer Window displays a list of the resource files and references in a project i.e. forms, classes and standard modules in the project.** All of the objects that make up the application are packed in a project. A simple project will typically contain one form, which is a window that is designed as part of a program's interface. It is possible to develop any number of forms for use in a program, although a program may consist of a single form. In addition to forms, the Project Explorer window also lists code modules and classes.

Figure 3 Project Explorer



Properties Window

The Properties Window is docked under the Project Explorer window. **The Properties Window exposes the various characteristics (properties) of selected objects.** Each and every form in an application is considered an object. Now, each object in Visual Basic has characteristics such as color and size. Other characteristics affect not just the appearance of the object but the way it behaves too. All these characteristics of an object are called its **properties**. Thus, a form has properties and any controls placed on it will have properties too. All of these properties are displayed in the Properties Window.

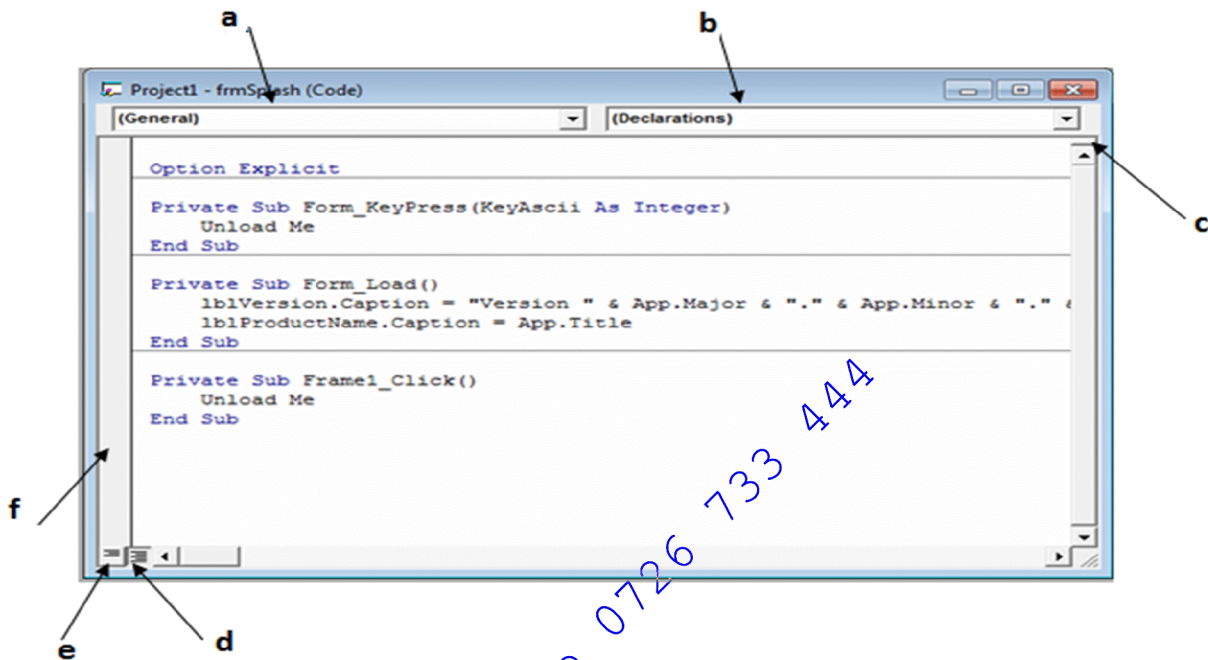
Object Browser

The Object Browser allows programmers to browse through the various properties, events and methods that are made available in a project. It is accessed by selecting Object Browser from the View menu or pressing the key F2. The left column of the Object Browser lists the objects and classes that are available in the projects that are opened and the controls that have been referenced in them. It is possible for us to scroll through the list and select the object or class that we wish to inspect. After an object is picked up from the Classes list, we can see its members (properties, methods and events) in the right column.

A property is represented by a small icon that has a hand holding a piece of paper. Methods are denoted by little green blocks, while events are denoted by yellow lightning bolt icon.

CODE EDITOR WINDOW

The Visual Basic Code Editor is a window where you write most of your code. It is like a highly specialized word processor with a number of features that make writing Visual Basic code a lot easier.



a. Object Box

This displays the name of the selected object. Click the arrow to the right of the list box to display a list of all objects associated with the form.

b. Procedures/Events Box

Lists all the events recognized by Visual Basic for a form or control displayed in the Object box. When you select an event, the event procedure associated with that event name is displayed in the Code window.

c. Split Bar

Dragging this bar down, splits the Code window into two horizontal panes, each of which scrolls separately. You can then view different parts of your code at the same time. The information that appears in the Object box and Procedures/Events box applies to the code in the pane that has the focus. Dragging the bar to the top or the bottom of the window or double-clicking the bar closes a pane.

d. Full Module View icon

Displays the entire code in the module.

e. Procedure View icon

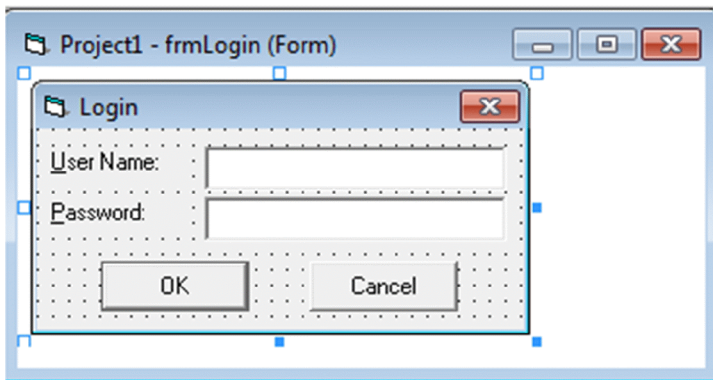
Displays the selected procedure. Only one procedure at a time is displayed in the Code window.

f. Margin Indicator Bar

A gray area on the left side of the Code window where margin indicators are displayed.

FORM DESIGNER WINDOW

Allows you to add controls to a form, arrange them, and write code for their events by double-clicking the form or another control in the designer and write code for the control's default event.

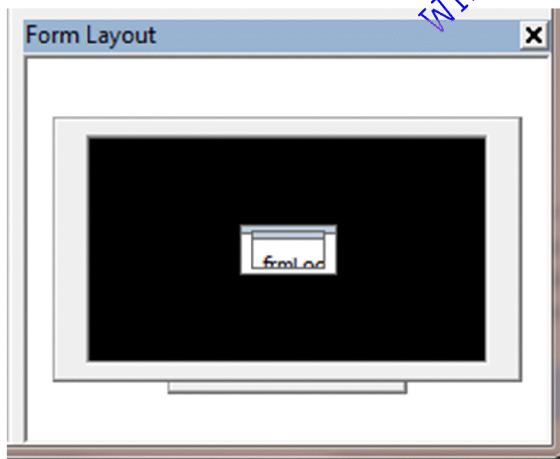


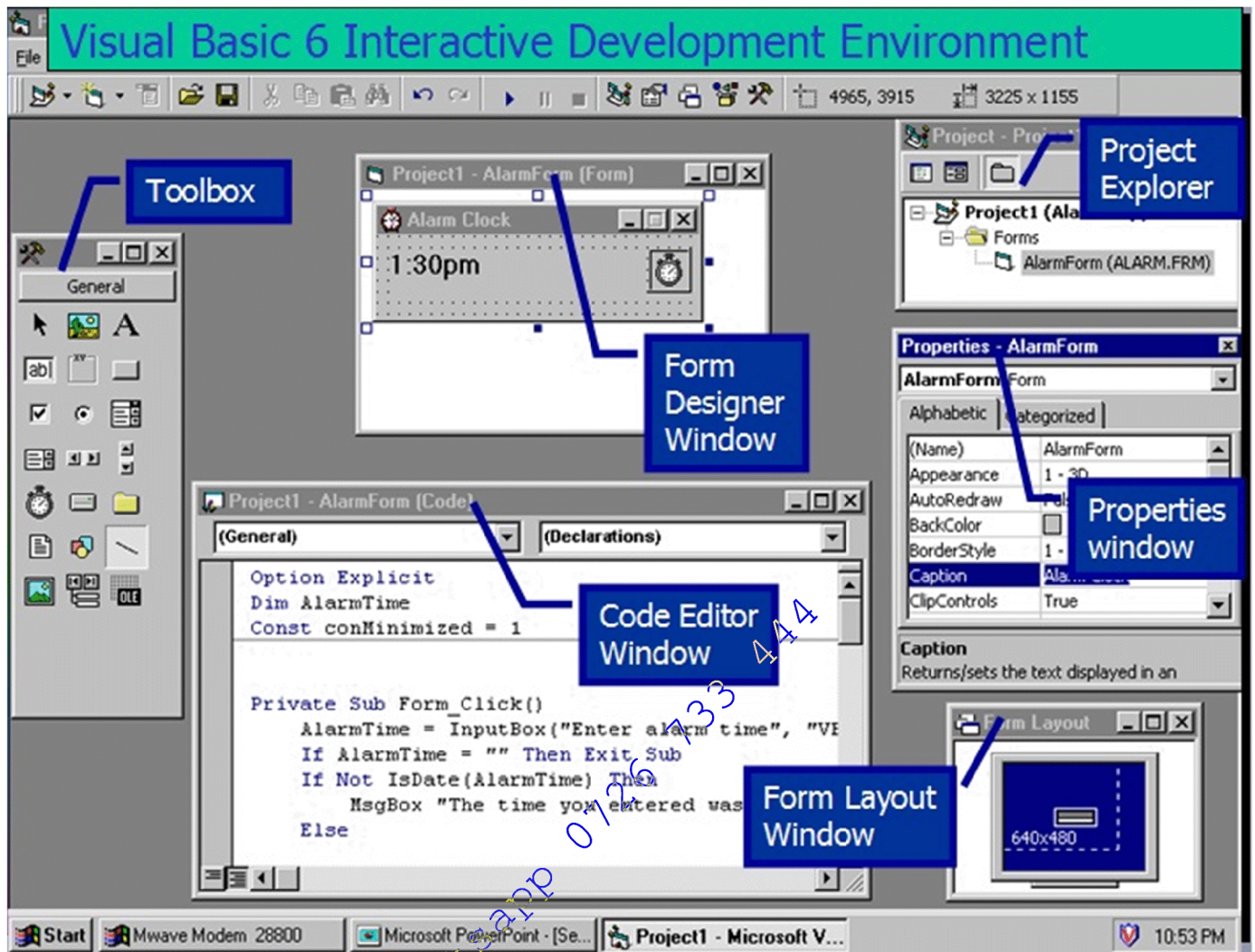
Anchoring and Docking

The Anchor and Dock properties of a form are two separate properties. Anchor refers to the position a control has relative to the edges of the form. A textbox, for example, that is anchored to the left edge of a form will stay in the same position as the form is resized. Docking refers to how much space you want the control to take up on the form. If you dock a control to the left of the form, it will stretch itself to the height of the form, but its width will stay the same.

FORM LAYOUT WINDOW

The Form Layout window allows you to position your form or forms with regard to the rest of the screen. It uses a graphical interface to guide you.





Formatting Text

Many business applications let you enter data in one format and then display it in another. For example, numeric values can be formatted with thousand separators and a fixed number of decimal digits. Currency values might have a \$ symbol (or whatever your national currency symbol is) automatically inserted. Phone numbers can be formatted with dashes to split into groups of digits. Credit-card numbers can be made more readable with embedded spaces. Dates can be shown in long-date format ("July 22, 2007"). And so on.

The LostFocus event is an ideal occasion to format the contents of a TextBox control as soon as the input focus leaves it. In most cases, you can perform all your formatting chores using the Format function. For example, you can add thousand separators to a numeric value in the txtNumber control using this code:

```
Private Sub txtNumber_LostFocus()
On Error Resume Next
txtNumber.Text = Format(CDbl(txtNumber.Text), _
"#,###,###,##0.#####")
End Sub
```

When the field regains the focus, you'll want to get rid of those thousand separators. You can do it easily using the CDbl function:

```
Private Sub txtNumber_GotFocus()
' On Error is necessary to account for empty fields.
On Error Resume Next
txtNumber.Text = CDbl(txtNumber.Text)
End Sub
```

In some cases, however, formatting and unformatting a value isn't that simple. For example, you can format a Currency value to add parentheses around negative numbers, but there's no built-in Visual Basic function able to return a string formatted in that way to its original condition. Fear not, because nothing prevents you from creating your own formatting and unformatting routines. I have built two general-purpose routines for you to consider.

You can also determine whether the system uses dates in "mm/dd/yy" (U.S.) format or "dd/mm/yy" (European) format, using this code:

```
If Left$(Format$("12/31/1999", "short date"), 2) = 12 Then
' mm/dd/yy format

Else
' dd/mm/yyyy format

End If
```

There's no direct way to determine the currency symbol, but you can derive it by analyzing the result of this function:

```
Format$(0, "currency") ' Returns "$0.00" in US
```

CONTROLS IN VISUAL BASIC

Label and Frame Controls in Visual Basic 6 (VB6)

Label and Frame controls have a few features in common, so it makes sense to explain them together. First they're mostly "decorative" controls that contribute to the user interface but are seldom used as programmable objects. In other words, you often place them on the form and arrange their properties as your user interface needs dictate, but you rarely write code to serve their events, generally, or manipulate their properties at run time.

Frame Controls

Frame controls are similar to Label controls in that they can serve as captions for those controls that don't have their own. Moreover, Frame controls can also (and often do) behave as containers and host other controls. In most cases, you only need to drop a Frame control on a form and set its Caption property. If you want to create a borderless frame, you can set its BorderStyle property to 0-None.

Controls that are contained in the Frame control are said to be child controls. Moving a control at design time over a Frame control—or over any other container, for that matter—doesn't automatically make that control a child of the Frame control. After you create a Frame control, you can create a child control by selecting the child control's icon in the Toolbox and drawing a new instance inside the Frame's border. Alternatively, to make an existing control a child of a Frame control, you must select the control, press Ctrl+X to cut it to the Clipboard, select the Frame control, and press Ctrl+V to paste the control inside the Frame. If you don't follow this procedure and you simply move the control over the Frame, the two controls remain completely independent of each other, even if the other control appears in front of the Frame control.

Frame controls, like all container controls, have two interesting features. If you move a Frame control, all the child controls go with it. If you make a container control disabled or invisible, all its child controls also become disabled or invisible. You can exploit these features to quickly change the state of a group of related controls.

IMAGE CONTROLS IN VB

1. Picture Box

a) **PROPERTIES** (Just a few)

- ✓ Name
- ✓ picture
- ✓ Autosize: the picture box will resize to match the dimensions of the picture. If set to false, the picture will be cropped

b) **USES**

- ✓ To display pictures
- ✓ As a container for other controls including graphics
- ✓ As a toolbar
- ✓ As a status bar where u can place labels to display status messages
 1. Align property to bottom
- ✓ As a button

c) **METHODS**

- ✓ Print
- ✓ Cls – to clear any text printed on it
- ✓ Animation

To load an Image at run time you use the statement below
`picPhoto.Picture = LoadPicture ("Path")`

To remove an Image at run time you use the same statement but now without the path
`picPhoto.Picture = LoadPicture`

2. Image Control

a) **USES**

- ✓ Displaying pictures

b) PROPERTIES (Just a few)

Sizing behaviour is different from that of picture box

- ✓ Stretch property: If False, the image control resizes to the dimensions of the picture. If set to True, the picture resizes to the size of the image control, distorting it.

NB/ The image control is **limited in its uses** but a **lightweight** graphical control. It, therefore, **requires less system resources and loads faster than the picture box**.

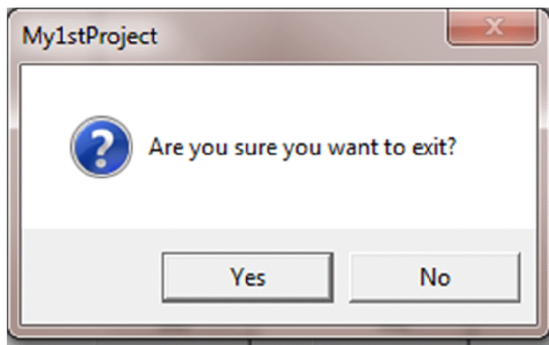
Exercise

1. Below is a screen shot of a program that displays five pictures in a picture box, from picture one to five or downwards and displays the number of the picture displayed. When the user clicks on the restart button, the picture box is emptied and the user can start the process afresh.



Required:

- i. Create a subfolder in your project folder and call it myimages.
 - ii. Browse for and save five **jpg** images of about 8.0 kb in size and give them names ranging from photo1 to photo5 in myimages folder.
 - iii. Write the program such that the photos are displayed incrementally or decrementally as the user clicks the buttons.
2. The message box shown below is generated when a user clicks the Exit button in the program above. If the user clicks yes, the program is closed and nothing happens when the user clicks the No button. Write the procedure to implement the functionality.



CommandButton and OptionButton Controls

When compared to TextBox controls, these controls are really simple. Not only do they expose relatively few properties, they also support a limited number of events, and you don't usually write much code to manage them.

CommandButton Controls in VB6

Using CommandButton controls is trivial. In most cases, you just draw the control on the form's surface, set its Caption property to a suitable string (adding an & character to associate a hot key with the control if you so choose), and you're finished, at least with user-interface issues. To make the button functional, you write code in its Click event procedure, as in this fragment:

```
Private Sub Command1_Click()  
' Save data, then unload the current form.  
Call SaveDataToDisk  
Unload Me  
End Sub
```

You can use two other properties at design time to modify the behavior of a CommandButton control. You can set the Default property to True if it's the default push button for the form (the button that receives a click when the user presses the Enter key—usually the OK or Save button). Similarly, you can set the Cancel property to True if you want to associate the button with the Escape key.

The only relevant CommandButton's run-time property is Value, which sets or returns the state of the control (True if pressed, False otherwise). Value is also the default property for this type of control. In most cases, you don't need to query this property because if you're inside a button's Click event you can be sure that the button is being activated. The Value property is useful only for programmatically clicking a button:

This fires the button's Click event.
`Command1.Value = True`

The CommandButton control supports the usual set of keyboard and mouse events (KeyDown, KeyPress, KeyUp, MouseDown, MouseMove, MouseUp, but not the DblClick event) and also the GotFocus and LostFocus events, but you'll rarely have to write code in the corresponding event procedures.

Properties of a CommandButton control

- To display text on a CommandButton control, set its *Caption* property.
- An event can be activated by clicking on the CommandButton.
- To set the background colour of the CommandButton, select a colour in the BackColor property.
- To set the text colour set the Forecolor property.
- Font for the CommandButton control can be selected using the Font property.
- To enable or disable the buttons set the Enabled property to True or False
- To make visible or invisible the buttons at run time, set the Visible property to True or False.
- Tooltips can be added to a button by setting a text to the Tooltip property of the CommandButton.
- A button click event is handled whenever a command button is clicked. To add a click event handler, double click the button at design time, which adds a subroutine like the one given below.

```
Private Sub Command1_Click()  
.....  
End Sub
```

OptionButton Controls in VB6

OptionButton controls are also known as radio buttons because of their shape. You always use OptionButton controls in a group of two or more because their purpose is to offer a number of mutually exclusive choices. Anytime you click on a button in the group, it switches to a selected state and all the other controls in the group become unselected.

Preliminary operations for an OptionButton control are similar to those already described for CheckBox controls. You set an OptionButton control's Caption property to a meaningful string, and if you want you can change its Alignment property to make the control right aligned. If the control is the one in its group that's in the selected state, you also set its Value property to True. (The OptionButton's Value property is a Boolean value because only two states are possible.) Value is the default property for this control.

At run time, you typically query the control's Value property to learn which button in its group has been selected. Let's say you have three OptionButton controls, named optWeekly, optMonthly, and optYearly. You can test which one has been selected by the user as follows:

```
If optWeekly.Value Then  
' User prefers weekly frequency.  
ElseIf optMonthly.Value Then  
' User prefers monthly frequency.  
ElseIf optYearly.Value Then  
' User prefers yearly frequency.  
End If
```

Strictly speaking, you can avoid the test for the last OptionButton control in its group because all choices are supposed to be mutually exclusive. But the approach I just showed you increases the code's readability.

A group of `OptionButton` controls is often hosted in a `Frame` control. This is necessary when there are other groups of `OptionButton` controls on the form. As far as Visual Basic is concerned, all the `OptionButton` controls on a form's surface belong to the same group of mutually exclusive selections, even if the controls are placed at the opposite corners of the window. The only way to tell Visual Basic which controls belong to which group is by gathering them inside a `Frame` control. Actually, you can group your controls within any control that can work as a container—`PictureBox`, for example—but `Frame` controls are often the most reasonable choice.

Example

Open a new Standard EXE project and then save the Form as `Option.frm` and save the project as `Option.vbp`.

Design the Form as per the following specifications table.

Object	Property	Settings
Label	Caption	Enter a Number
	Name	Label1
TextBox	Text	(empty)
	Name	Text1
CommandButton	Caption	&Close
	Name	Command1
OptionButton	Caption	&Octal
	Name	optOct
OptionButton	Caption	&Hexadecimal
	Name	optHex
OptionButton	Caption	&Decimal
	Name	optDec

The application responds to the following events

- The change event of the `TextBox` reads the value and stores it in a form-level numeric variable.
- The click event of `optOct` button returns currentval in octal.
- The click event of the `optHex` button returns currentval in hexadecimal.
- The click event of the `optDec` button returns the decimal equivalent of the value held in currentval.

The following code is entered in the general declarations section of the Form.

Dim currentval as variant

The variable is initialized to 0 by default. The change event procedure checks to ascertain the number system (Octal, Hexadecimal) that is in effect and then reads in the number.

```
Private Sub Text1_Change()  
If optOct.Value = True Then  
currentval = Val("&O" & LTrim(Text1.Text) & "&")  
Elseif optDec.value = True Then  
currentval = Val(LTrim(Text1.Text) & "&")  
Else  
currentval = Val("&H" & LTrim(Text1.Text) & "&")  
End if  
End Sub
```

The Val function is used to translate string to a number and can recognize Octal and Hexadecimal strings. The LTrim function trims the leading blanks in the text. The following code is entered in the click events of the OptionButton controls.

```
Private Sub optOct_Click()  
Text1.Text = Oct(currentval)  
End Sub
```

```
Private Sub optHex_Click()  
Text1.Text = Hex(currentval)  
End Sub
```

```
Private Sub optDec_Click()  
Text1.Text = Format(currentval)  
End Sub
```

The following code is entered in the click event of the Close button.

```
Private Sub cmdClose_Click()  
Unload Me  
End Sub
```

The Application is run by pressing F5 or clicking on the Run icon in the tool bar. By pressing the Exit button the program is terminated.

The Timer, Line, Shape and OLE Controls

The Timer Control

A Timer control is invisible at run time, and **its purpose is to send a periodic pulse to the current application**. You can trap this pulse by writing code in the Timer's Timer event procedure and take advantage of it to **execute a task in the background or to monitor a user's actions**. This control exposes only two meaningful properties: Interval and Enabled. Interval stands for the number of milliseconds between subsequent pulses (Timer events), while Enabled lets you activate or deactivate events. When you place the Timer control on a form, its Interval is 0, which means no events. Therefore, remember to set this property to a suitable value in the Properties window or in the Form_Load event procedure:

```
Private Sub Form_Load()  
Timer1.Interval = 500 ' Fire two Timer events per second.  
End Sub
```

Timer controls let you write interesting programs with just a few lines of code. The typical (and abused) example is a digital clock. Just to make things a bit more compelling, I added flashing colons:

```
Private Sub Timer1_Timer()  
Dim strTime As String  
strTime = Time$  
If Mid$(lblClock.Caption, 3, 1) = ":" Then  
Mid$(strTime, 3, 1) = " "  
Mid$(strTime, 6, 1) = " "  
End If  
lblClock.Caption = strTime  
End Sub
```

You must be careful not to write a lot of code in the Timer event procedure because this code will be executed at every pulse and therefore can easily degrade your application's performance. Just as important, never execute a DoEvents statement inside a Timer event procedure because you might cause the procedure to be reentered, especially if the Interval property is set to a small value and there's a lot of code inside the procedure.

Timer controls are often useful for updating status information on a regular basis. For example, you might want to display on a status bar a short description of the control that currently has the input focus. You can achieve that by writing some code in the GotFocus event for all the controls on the form, but when you have dozens of controls this will require a lot of code (and time). Instead, at design time load a short description for each control in its Tag property, and then place a Timer control on the form with an Interval setting of 500. This isn't a time-critical task, so you can use an even larger value. Finally add two lines of code to the control's Timer event:

```
Private Sub Timer1_Timer()  
On Error Resume Next  
lblStatusBar.Caption = ActiveControl.Tag  
End Sub
```

The Line Control

The Line control is a decorative control whose only purpose is let you draw one or more straight lines at design time, instead of displaying them using a Line graphical method at run time. This control exposes a few properties whose meaning should sound familiar to you by now: **BorderColor** (the color of the line), **BorderStyle** (the same as a form's **DrawStyle** property), **BorderWidth** (the same as a form's **DrawWidth** property), and **DrawMode**. While the Line control is handy, remember that using a Line method at run time is usually better in terms of performance.

The Shape Control

In a sense, the Shape control is an extension of the Line control. It can display six basic shapes: Rectangle, Square, Oval, Circle, Rounded Rectangle, and Rounded Square. It supports all the Line control's properties and a few more: **BorderStyle** (0-Transparent, 1-Solid), **FillColor**, and **FillStyle** (the same as a form's properties with the same names). The same performance considerations I pointed out for the Line control apply to the Shape control.

Shape control is used to draw different shapes on the form. Available shapes are:

1. 0-Rectangle
2. 1-Square
3. 2-Oval
4. 3-Circle
5. 4-Rounded Rectangle
6. 5-Rounded Square

The OLE Control

OLE refers to **Object Linking and Embedding**.

OLE Container control was to embed objects/other applications e.g. Word documents or Excel spreadsheets in a form. This is the control to use when you want to link or embed an object into your Visual Basic

Properties include:

Autoactivate – determines how the object linked autoactivates

SourceDoc – Sets the name of the file to link

SizeMode – determines how the image is displayed e.g. if the control resizes to accomodate the object

The followings are the other options available in Insert Object Dialog box.

1. Create New Allows you to create a new file and then embed an object from newly created file.
2. Cretate From File Allows an object to be embedded or linked from a file.

ListBox and ComboBox Controls

ListBox and ComboBox controls present a set of choices that are displayed vertically in a column. If the number of items exceeds the value that is displayed, scroll bars will automatically appear on the control. These scroll bars can be scrolled up and down or left to right through the list.

The following Fig lists some of the common **ComboBox** properties and methods.

Property/Method	Description
Properties	
Enabled	By setting this property to True or False user can decide whether user can interact with this control or not
Index	Specifies the Control array index
List	String array. Contains the strings displayed in the drop-down list. Starting array index is 0.
ListCount	Integer. Contains the number of drop-down list items
ListIndex	Integer. Contains the index of the selected ComboBox item. If an item is not selected, ListIndex is -1
Locked	Boolean. Specifies whether user can type or not in the ComboBox
MousePointer	Integer. Specifies the shape of the mouse pointer when over the area of the ComboBox
NewIndex	Integer. Index of the last item added to the ComboBox. If the ComboBox does not contain any items , NewIndex is -1
Sorted	Boolean. Specifies whether the ComboBox's items are sorted or not.
Style	Integer. Specifies the style of the ComboBox's appearance
TabStop	Boolean. Specifies whether ComboBox receives the focus or not.
Text	String. Specifies the selected item in the ComboBox
ToolTipIndex	String. Specifies what text is displayed as the ComboBox's tool tip
Visible	Boolean. Specifies whether ComboBox is visible or not at run time
Methods	
AddItem	Add an item to the ComboBox
Clear	Removes all items from the ComboBox
RemoveItem	Removes the specified item from the ComboBox
SetFocus	Transfers focus to the ComboBox
Event Procedures	
Change	Called when text in ComboBox is changed
DropDown	Called when the ComboBox drop-down list is displayed
GotFocus	Called when ComboBox receives the focus
LostFocus	Called when ComboBox loses it focus

Adding items to a List

It is possible to populate the list at design time or run time

Design Time: To add items to a list at design time, click on List property in the property box and then add the items. Press CTRL+ENTER after adding each item as shown below.



Run Time: The AddItem method is used to add items to a list at run time. The AddItem method uses the following syntax.

Object.AddItem item, Index

The **item** argument is a string that represents the text to add to the list

The **index** argument is an integer that indicates where in the list to add the new item. Not giving the index is not a problem, because by default the index is assigned.

Following is an example to add item to a combo box. The code is typed in the Form_Load event

```
Private Sub Form_Load()
```

```
    Combo1.AddItem 1
    Combo1.AddItem 2
    Combo1.AddItem 3
    Combo1.AddItem 4
    Combo1.AddItem 5
    Combo1.AddItem 6
```

```
End Sub
```

Removing Items from a List

The RemoveItem method is used to remove an item from a list. The syntax for this is given below.

Object.RemoveItem index

The following code verifies that an item is selected in the list and then removes the selected item from the list.

```
Private Sub cmdRemove_Click()
    If List1.ListIndex > -1 Then
        List1.RemoveItem List1.ListIndex
    End If
End Sub
```

Sorting the List

The Sorted property is set to True to enable a list to appear in alphanumeric order and False to display the list items in the order which they are added to the list.

Using the ComboBox

A ComboBox combines the features of a TextBox and a ListBox. This enables the user to select either by typing text into the ComboBox or by selecting an item from the list. There are three types of ComboBox styles that are represented as shown below.



- Dropdown Combo (style 0)
- Simple Combo (style 1)
- Dropdown List (style 2)

The Simple Combo box displays an edit area with an attached list box always visible immediately below the edit area. A simple combo box displays the contents of its list all the time. The user can select an item from the list or type an item in the edit box portion of the combo box. A scroll bar is displayed beside the list if there are too many items to be displayed in the list box area.

The Dropdown Combo box first appears as only an edit area with a down arrow button at the right. The list portion stays hidden until the user clicks the down arrow button to drop down the list portion. The user can either select a value from the list or type a value in the edit area.

The Dropdown list combo box turns the combo box into a Dropdown list box. At run time, the control looks like the Dropdown combo box. The user could click the down arrow to view the list. The difference between Dropdown combo & Dropdown list combo is that the edit area in the Dropdown list combo is disabled. The user can only select an item and cannot type anything in the edit area. Anyway this area displays the selected item.

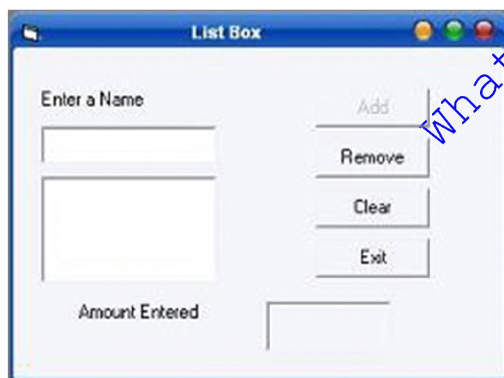
Example

This example is to Add, Remove, Clear the list of items and finally close the application.

- Open a new Standard EXE project is opened and named the Form as Listbox.frm and save the project as Listbox.vbp
- Design the application as shown below.

Object	Property	Settings
Form	Caption	Listbox
	Name	frmListBox
TextBox	Text	(empty)
	Name	txtName

Label	Caption	Enter a name
	Name	lblName
ListBox	Name	lstName
Label	Caption	Amount Entered
	Name	lblAmount
Label	Caption	(empty)
	Name	lblDisplay
	Border Style	1 Fixed Single
CommandButton	Caption	Add
	Name	cmdAdd
CommandButton	Caption	Remove
	Name	cmdRemove
CommandButton	Caption	Clear
	Name	cmdClear
CommandButton	Caption	Exit
	Name	cmdExit



The following event procedures are entered for the TextBox and CommandButton controls.

```
Private Sub txtName_Change()
If (Len(txtName.Text) > 0) Then 'Enabling the Add button
'if atleast one character
'is entered
cmdAdd.Enabled = True
End If
End Sub
```

```

Private Sub cmdAdd_Click()
lstName.AddItem txtName.Text 'Add the entered the characters to the list box

txtName.Text = "" 'Clearing the text box

txtName.SetFocus 'Get the focus back to the
'text box

lblDisplay.Caption = lstName.ListCount 'Display the number of items in the list box

cmdAdd.Enabled = False ' Disabling the Add button

End Sub

```

The click event of the Add button adds the text to the list box that was typed in the Text box. Then the text box is cleared and the focus is got to the text box. The number of entered values will be increased according to the number of items added to the listbox.

```

Private Sub cmdClear_Click()
lstName.Clear
lblDisplay.Caption = lstName.ListCount
End Sub

Private Sub cmdExit_Click()
Unload Me
End Sub

Private Sub cmdRemove_Click()
Dim remove As Integer

remove = lstName.ListIndex 'Getting the index

If remove >= 0 Then 'make sure an item is selected
'in the list box

lstName.RemoveItem remove 'Remove item from the list box

lblDisplay.Caption = lstName.ListCount 'Display the number of items
'in the listbox

End If

End Sub

```

Remove button removes the selected item from the list as soon as you pressed the Remove button. The number of items is decreased in the listbox and the value is displayed in the label.

The code for the clear button clears the listbox when you press it. And the number of items shown in the label becomes 0.

VB ScrollBar

The ScrollBar is a commonly used control, which enables the user to select a value by positioning it at the desired location. It represents a set of values. The Min and Max property represents the minimum and maximum value. The value property of the ScrollBar represents its current value, that may be any integer between minimum and maximum values assigned.

The HScrollBar and the VScrollBar controls are perfectly identical, apart from their different orientation. After you place an instance of such a control on a form, you have to worry about only a few properties: Min and Max represent the valid range of values, SmallChange is the variation in value you get when clicking on the scroll bar's arrows, and LargeChange is the variation you get when you click on either side of the scroll bar indicator. The default initial value for those two properties is 1, but you'll probably have to change LargeChange to a higher value. For example, if you have a scroll bar that lets you browse a portion of text, SmallChange should be 1 (you scroll one line at a time) and LargeChange should be set to match the number of visible text lines in the window.

The most important run-time property is Value, which always returns the relative position of the indicator on the scroll bar. By default, the Min value corresponds to the leftmost or upper end of the control:

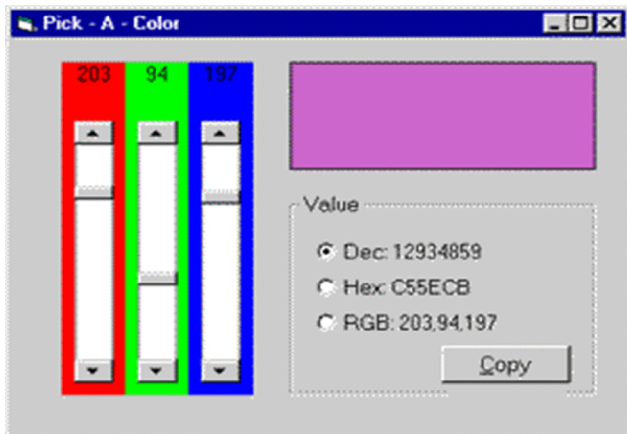
```
' Move the indicator near the top (or left) arrow.  
VScroll1.Value = VScroll1.Min  
' Move the indicator near the bottom (or right) arrow.  
VScroll1.Value = VScroll1.Max
```

While this setting is almost always OK for horizontal scroll bars, you might sometimes need to reverse the behavior of vertical scroll bars so that the zero is near the bottom of your form. This arrangement is often desirable if you want to use a vertical scroll bar as a sort of slider. You obtain this behavior by simply inverting the values in the Min and Max properties. (In other words, it's perfectly legal for Min to be greater than Max.)

There are two key events for scrollbar controls: the Change event fires when you click on the scroll bar arrows or when you drag the indicator; the Scroll event fires while you drag the indicator. The reason for these two distinct possibilities is mostly historical. First versions of Visual Basic supported only the Change event, and when developers realized that it wasn't possible to have continuous feedback when users dragged the indicator, Microsoft engineers added a new event instead of extending the Change event. In this way, old applications could be recompiled without unexpected changes in their behavior. At any rate, this means that you must often trap two distinct events:

```
' Show the current scroll bar's value.  
Private VScroll1_Change()  
Label1.Caption = VScroll1.Value  
End Sub  
Private VScroll1_Scroll()  
Label1.Caption = VScroll1.Value  
End Sub
```

The example shown in the following figure uses three VScrollBar controls as sliders to control the individual RGB (red, green, blue) components of a color. The three scroll bars have their Min property set to 255 and their Max property set to 0, while their SmallChange is 1 and LargeChange is 16. This example is also a moderately useful program in itself because you can select a color and then copy its numeric value to the clipboard and paste it in your application's code as a decimal value, a hexadecimal value, or an RGB function.



Use scrollbar controls to visually create colors.

Scrollbar controls can receive the input focus, and in fact they support both the TabIndex and TabStop properties. If you don't want the user to accidentally move the input focus on a scrollbar control when he or she presses the Tab key, you must explicitly set its TabStop property to False. When a scrollbar control has the focus, you can move the indicator using the Left, Right, Up, Down, PgUp, PgDn, Home, and End keys. For example, you can take advantage of this behavior to create a read-only TextBox control with a numeric value that can be edited only through a tiny companion scroll bar. This scroll bar appears to the user as a sort of spin button, as you can see in the figure below. To make the trick work, you need to write just a few lines of code:

```
Private Sub Text1_GotFocus()
' Pass the focus to the scroll bar.
VScroll1.SetFocus
End Sub
Private Sub VScroll1_Change()
' Scroll bar controls the text box value.
Text1.Text = VScroll1.Value
End Sub
```



You don't need external ActiveX controls to create functional spin buttons

Scrollbar controls are even more useful for building scrolling forms, like the one displayed in Figure 3-15. To be certain, scrolling forms aren't the most ergonomic type of user interface you can offer to your customers: If you have that many fields in a form, you should consider using a Tab control, child forms, or some other custom interface. Sometimes, however, you badly need scrollable forms, and in this situation you are on your own because Visual Basic forms don't support scrolling.

DriveListBox, DirListBox, and FileListBox Controls

Three of the controls on the ToolBox let you access the computer's file system. They are DriveListBox, DirListBox and FileListBox controls (see below figure) , which are the basic blocks for building dialog boxes that display the host computer's file system. Using these controls, user can traverse the host computer's file system, locate any folder or files on any hard disk, even on network drives. The files are controls are independent of one another, and each can exist on it's own, but they are rarely used separately. The files controls are described next.

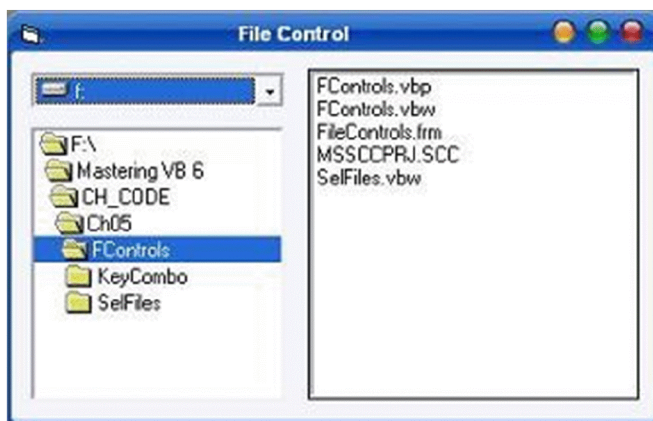
In a nutshell, the DriveListBox control is a combobox-like control that's automatically filled with your drive's letters and volume labels. The DirListBox is a special list box that displays a directory tree. The FileListBox control is a special-purpose ListBox control that displays all the files in a given directory, optionally filtering them based on their names, extensions, and attributes.

These controls often work together on the same form; when the user selects a drive in a DriveListBox, the DirListBox control is updated to show the directory tree on that drive. When the user selects a path in the DirListBox control, the FileListBox control is filled with the list of files in that directory. These actions don't happen automatically, however—you must write code to get the job done.

After you place a DriveListBox and a DirListBox control on a form's surface, you usually don't have to set any of their properties; in fact, these controls don't expose any special property, not in the Properties window at least. The FileListBox control, on the other hand, exposes one property that you can set at design time—the Pattern property. This property indicates which files are to be shown in the list area: Its default value is *.* (all files), but you can enter whatever specification you need, and you can also enter multiple specifications using the semicolon as a separator. You can also set this property at run time, as in the following line of code:

```
File1.Pattern = "*.txt;*.doc;*.rtf"
```

Following figure shows three files controls are used in the design of Forms that let users explore the entire structure of their hard disks.



- **DriveListBox** : Displays the names of the drives within and connected to the PC. The basic property of this control is the drive property, which set the drive to be initially selected in the control or returns the user's selection.
- **DirListBox** : Displays the folders of current Drive. The basic property of this control is the Path property, which is the name of the folder whose sub folders are displayed in the control.
- **FileListBox** : Displays the files of the current folder. The basic property of this control is also called Path, and it's the path name of the folder whose files are displayed.

The three File controls are not tied to one another. If you place all three of them on a Form, you will see the names of all the folders under the current folder, and so on. Each time you select a folder in the DirListBox by double clicking its name, its sub folders are displayed. Similarly, the FileListBox control will display the names of all files in the current folder. Selecting a drive in the DriveListBox control, however this doesn't affect the contents of the DirListBox.

To connect to the File controls, you must assign the appropriate values to the properties. To compel the DirListBox to display the folders of the selected drive in the DriveListBox, you must make sure that each time the user selects another drive, the Path property of the DirListBox control matches the Drive property of the DriveListBox.

After these preliminary steps, you're ready to set in motion the chain of events. When the user selects a new drive in the DriveListBox control, it fires a Change event and returns the drive letter (and volume label) in its Drive property. You trap this event and set the DirListBox control's Path property to point to the root directory of the selected drive:

```
Private Sub Drive1_Change()  
' The Drive property also returns the volume label, so trim it.  
Dir1.Path = Left$(Drive1.Drive, 1) & ":\"  
End Sub
```

When the user double-clicks on a directory name, the DirListBox control raises a Change event; you trap this event to set the FileListBox's Path property accordingly:

```
Private Sub Dir1_Change()  
File1.Path = Dir1.Path  
End Sub
```

Finally, when the user clicks on a file in the FileListBox control, a Click event is fired (as if it were a regular ListBox control), and you can query its Filename property to learn which file has been selected. Note how you build the complete path:

```
Filename = File1.Path  
If Right$(Filename, 1) <> "\" Then Filename = Filename & "\"  
Filename = Filename & File1.Filename
```

The DirListBox and FileListBox controls support most of the properties typical of the control they derive from—the ListBox control—including the ListCount and the ListIndex properties and the Scroll event. The FileListBox control supports multiple selection; hence you can set its MultiSelect property in the Properties window and query the SelCount and Selected properties at run time.

The FileListBox control also exposes a few custom Boolean properties, Normal, Archive, Hidden, ReadOnly, and System, which permit you to decide whether files with these attributes should be listed. (By default, the control doesn't display hidden and system files.) This control also supports a couple of custom events, PathChange and PatternChange, that fire when the corresponding property is changed through code. In most cases, you don't have to worry about them, and I won't provide examples of their usage.

The problem with the DriveListBox, DirListBox and FileListBox controls is that they're somewhat outdated and aren't used by most commercial applications any longer. Moreover, these controls are known to work incorrectly when listing files on network servers and sometimes even on local disk drives, especially when long file and directory names are used. For this reason, I discourage you from using them and suggest instead that you use the Common Dialog controls for your FileOpen and FileSave dialog boxes. But if you need to ask the user for the name of a directory rather than a file, you're out of luck because—while Windows does include such a system dialog box, named BrowseForFolders dialog—Visual Basic still doesn't offer a way to display it (unless you do some advanced API programming). Fortunately, Visual Basic 6 comes with a new control—the ImageCombo control—that lets you simulate the appearance of the DriveListBox control. It also offers you a powerful library—the FileSystemObject library—that completely frees you from using these three controls, if only as hidden controls that you use just for quickly retrieving information on the file system.

Using a CheckBox control in Visual Basic 6

The CheckBox control is similar to the option button, except that a list of choices can be made using check boxes where you cannot choose more than one selection using an OptionButton. By ticking the CheckBox the value is set to True. This control can also be grayed when the state of the CheckBox is unavailable, but you must manage that state through code.

When you place a CheckBox control on a form, all you have to do, usually, is set its Caption property to a descriptive string. You might sometimes want to move the little check box to the right of its caption, which you do by setting the Alignment property to 1-Right Justify, but in most cases the default setting is OK. If you want to display the control in a checked state, you set its Value property to 1-Checked right in the Properties window, and you set a grayed state with 2-Grayed.

The only important event for CheckBox controls is the Click event, which fires when either the user or the code changes the state of the control. In many cases, you don't need to write code to handle this event. Instead, you just query the control's Value property when your code needs to process user choices. You usually write code in a CheckBox control's Click event when it affects the state of other controls. For example, if the user clears a check box, you might need to disable one or more controls on the form and reenable them when the user clicks on the check box again. This is how you usually do it (here I grouped all the relevant controls in one frame named Frame1):

```
Private Sub Check1_Click()  
Frame1.Enabled = (Check1.Value = vbChecked)  
End Sub
```

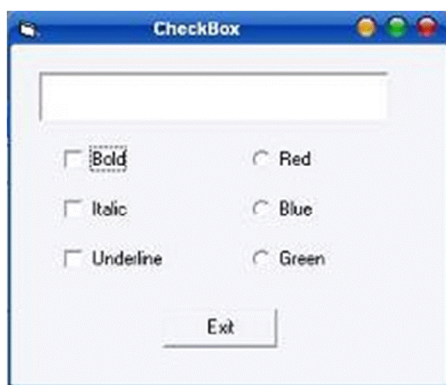
Note that Value is the default property for CheckBox controls, so you can omit it in code. I suggest that you not do that, however, because it would reduce the readability of your code.

The following example illustrates the use of CheckBox control

* Open a new Project and save the Form as CheckBox.frm and save the Project as CheckBox.vbp

* Design the Form as shown below

Object	Property	Setting
Form	Caption	CheckBox
	Name	frmCheckBox
CheckBox	Caption	Bold
	Name	chkBold
CheckBox	Caption	Italic
	Name	chkItalic
CheckBox	Caption	Underline
	Name	chkUnderline
OptionButton	Caption	Red
	Name	optRed
OptionButton	Caption	Blue
	Name	optBlue
OptionButton	Caption	Green
	Name	optGreen
TextBox	Name	txtDisplay
	Text	(empty)
CommandButton	Caption	Exit
	Name	cmdExit



Following code is typed in the Click() events of the CheckBoxes

```
Private Sub chkBold_Click()
If chkBold.Value = 1 Then
    txtDisplay.FontBold = True
Else
    txtDisplay.FontBold = False
End If
End Sub
```

```
Private Sub chkItalic_Click()
If chkItalic.Value = 1 Then
    txtDisplay.FontItalic = True
Else
    txtDisplay.FontItalic = False
End If
End Sub
```

```
Private Sub chkUnderline_Click()
If chkUnderline.Value = 1 Then
    txtDisplay.FontUnderline = True
Else
    txtDisplay.FontUnderline = False
End If
End Sub
```

Following code is typed in the Click() events of the OptionButtons

```
Private Sub optBlue_Click()
    txtDisplay.ForeColor = vbBlue
End Sub
```

```
Private Sub optRed_Click()
    txtDisplay.ForeColor = vbRed
End Sub
```

```
Private Sub optGreen_Click()
    txtDisplay.ForeColor = vbGreen
End Sub
```

To terminate the program following code is typed in the Click() event of the Exit button

```
Private Sub cmdExit_Click()
    End
End Sub
```

Run the program by pressing F5. Check the program by clicking on OptionButtons and CheckBoxes.

TabIndex property of Controls

Visual Basic uses the TabIndex property to determine the control that would receive the focus next when a tab key is pressed. Every time a tab key is pressed, Visual Basic looks at the value of the TabIndex for the control that has focus and then it scans through the controls searching for the next highest TabIndex number. When there are no more controls with higher TabIndex value, Visual Basic starts all over again with 0 and looks for the first control with TabIndex of 0 or higher that can accept keyboard input.

By default, Visual Basic assigns a tab order to control as we draw the controls on the Form, except for Menu, Timer, Data, Image, Line and Shape controls, which are not included in tab order. At run time, invisible or disabled controls also cannot receive the focus although a TabIndex value is given. Setting the TabIndex property of controls is compulsory in development environment.

Working with Forms in Visual Basic 6

The Appearance of Forms

The main characteristic of a Form is the title bar on which the Form's caption is displayed. On the left end of the title bar is the Control Menu icon. Clicking this icon opens the Control Menu. Maximize, Minimize and Close buttons can be found on the right side of the Form. Clicking on these buttons performs the associated function.

The following figure illustrates the appearance of a Form

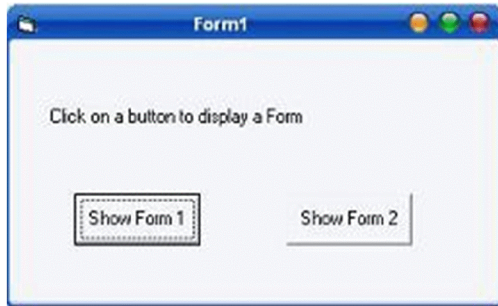


The control menu contains the following commands :

- **Restore** : Restores a maximized Form to the size it was before it was maximized; available only if the Form has been maximized.
- **Move** : Lets the user moves the Form around with the mouse
- **Size** : Lets the user resizes the control with the mouse
- **Minimize** : Minimizes the Form
- **Maximize** : Maximizes the Form
- **Close** : Closes the Form

Setting the Start-Up Form

A typical application has more than a single Form. When an application runs the main Form is loaded. By setting the Project properties you can control which Form is to be displayed in the Start-Up of the application. Following figure illustrates the Project property window.



By default, Visual Basic suggests the name of the first Form created when the project started.

At run time, the following code statements are used

$\text{Left} = (\text{Screen.Width} - \text{Width}) / 2$

$\text{Top} = (\text{Screen.Height} - \text{Height}) / 2$

Loading and Unloading Forms

In order to load and unload the forms, Load and Unload statements are used. The Load statement has the following syntax :

Load FormName

And the Unload statement has the following syntax :

Unload FormName

The FormName variable is the name of the Form to be loaded or unloaded. Unlike the Show method which cares of both loading and displaying the Form, the load statement doesn't show the Form. You have to call the Form's Show method to display it on the desktop.

Showing and Hiding Forms

Show method is used to Show a Form. If the Form is loaded but invisible, the Show method is used to bring the Form on Top every other window. If the Form is not loaded, the Show method loads it and then displays it.

Syntax of the Show method of the Form

FormName.Show mode

The FormName variable is the Form's name, and the optional argument mode determines whether the Form will be Modal or not. It can have one of the following syntax :

* 0-Modeless (default)

* 1-Modal

Modeless Forms are the normal Forms. Modeless Forms interact with the user and the user allowed to switch to any other Form of the application. If you do not specify the optional mode argument, by default the mode is set to modeless.

The Modal Forms takes the total control of the application where user cannot switch to any other Forms in the application unless the Form is closed. A modal Form, thus, must have a Close button or some means to close the Form in order to return to the Form where the Modal Form was loaded.

Hiding Forms

The Hide method is used to hide a Form. The following is the syntax of the Hide Method.

FormName.Hide

To hide a Form from within its own code, the following code can be used.

Me.Hide

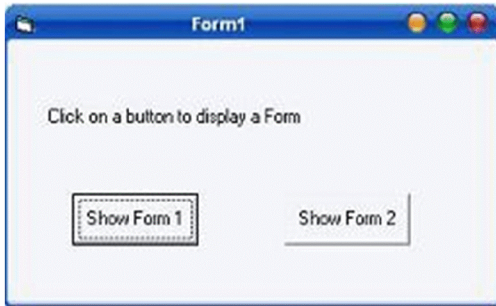
You must understand that the Forms that are hidden are not unloaded: they remain in the memory and can be displayed instantly with the Show Method. When a Form is hidden, you can still access its properties and code. For instance, you can change the settings of its Control Properties or call any Public functions in the Form.

The following is an example illustrates the Show method and Mode statement

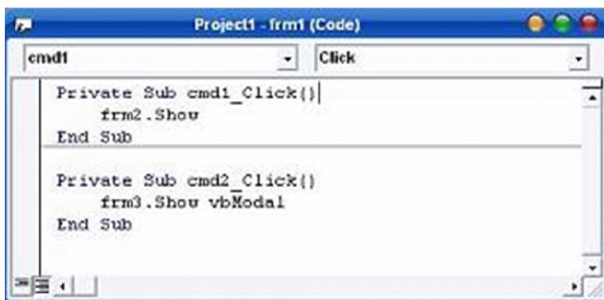
* Open a new Project and save the Project

Design the application as shown below

Object	Property	Setting
Form	Caption	Form1
	Name	frm1
Form	Caption	Form2
	Name	frm2
Form	Caption	Form3
	Name	frm3
Label	Caption	Click on a button to display a Form
	Name	Label1



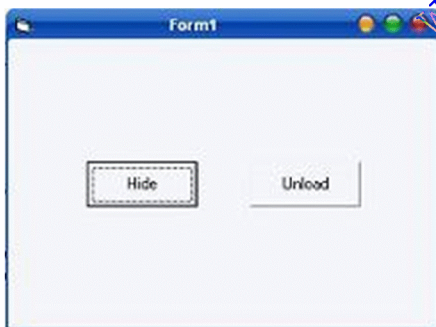
The following code is typed in the Click event of the command buttons



Run the application. Clicking on the buttons will display the Forms respectively. But you can see that in the cmd2_Click() event additionally VbModal argument has been added. You can see the difference after you display the forms by clicking on the command buttons. You can notice that you cannot switch to any other Forms in the application unless you close the Form3. [\(Download the source code\)](#)

Difference between Unload and Hide method

To know what the difference is between Unload and Hide methods we will do an example. Open a new project and save the project. Draw two buttons on the form and name those as shown above.



In the click event of the Hide button Following code is entered.

Me.Hide

In the click event of the Unload button following code is entered.

Unload Me

Save the project and run the application. Once you click on Hide button you can note that the Form is invisible but the application is still running. But when you click on Unload button you can see that the application is terminated.

Control Arrays in Visual Basic 6

A control array is a group of controls that share the same name type and the same event procedures. Adding controls with control arrays uses fewer resources than adding multiple control of same type at design time.

A control array can be created only at design time, and at the very minimum at least one control must belong to it. You create a control array following one of these three methods:

- You create a control and then assign a numeric, non-negative value to its Index property; you have thus created a control array with just one element.
- You create two controls of the same class and assign them an identical Name property. Visual Basic shows a dialog box warning you that there's already a control with that name and asks whether you want to create a control array. Click on the Yes button.
- You select a control on the form, press Ctrl+C to copy it to the clipboard, and then press Ctrl+V to paste a new instance of the control, which has the same Name property as the original one. Visual Basic shows the warning mentioned in the previous bullet.

Control arrays are one of the most interesting features of the Visual Basic environment, and they add a lot of flexibility to your programs:

- Controls that belong to the same control array share the same set of event procedures; this often dramatically reduces the amount of code you have to write to respond to a user's actions.
- You can dynamically add new elements to a control array at run time; in other words, you can effectively create new controls that didn't exist at design time.
- Elements of control arrays consume fewer resources than regular controls and tend to produce smaller executables. Besides, Visual Basic forms can host up to 256 different control names, but a control array counts as one against this number. In other words, control arrays let you effectively overcome this limit.

The importance of using control arrays as a means of dynamically creating new controls at run time is somewhat reduced in Visual Basic 6, which has introduced a new and more powerful capability.

Don't let the term array lead you to think control array is related to VBA arrays; they're completely different objects. Control arrays can only be one-dimensional. They don't need to be dimensioned: Each control you add automatically extends the array. The Index property identifies the position of each control in the control array it belongs to, but it's possible for a control array to have holes in the index sequence. The lowest possible value for the Index property is 0. You reference a control belonging to a control array as you would reference a standard array item:

```
Text1(0).Text = ""
```

Sharing Event Procedures

Event procedures related to items in a control array are easily recognizable because they have an extra Index parameter, which precedes all other parameters. This extra parameter receives the index of the element that's raising the event, as you can see in this example:

```
Private Sub Text1_KeyPress(Index As Integer, KeyAscii As Integer)
MsgBox "A key has been pressed on Text1(" & Index & ") control"
End Sub
```

The fact that multiple controls can share the same set of event procedures is often in itself a good reason to create a control array. For example, say that you want to change the background color of each of your TextBox controls to yellow when it receives the input focus and restore its background color to white when the user clicks on another field:

```
Private Sub Text1_GotFocus(Index As Integer)
Text1(Index).BackColor = vbYellow
End Sub
Private Sub Text1_LostFocus(Index As Integer)
Text1(Index).BackColor = vbWhite
End Sub
```

Control arrays are especially useful with groups of `OptionButton` controls because you can remember which element in the group has been activated by adding one line of code to their shared `Click` event. This saves code when the program needs to determine which button is the active one:

```
' A module-level variable
Dim optFrequencyIndex As Integer

Private Sub optFrequency_Click(Index As Integer)
' Remember the last button selected.
optFrequencyIndex = Index
End Sub
```

Iterating on the Items of a Control Array

Control arrays often let you save many lines of code because you can execute the same statement, or group of statements, for every control in the array without having to duplicate the code for each distinct control. For example, you can clear the contents of all the items in an array of `TextBox` controls as follows:

```
For i = txtFields.LBound To txtFields.UBound
txtFields(i).Text = ""
Next
```

Here you're using the `LBound` and `UBound` methods exposed by the control array object, which is an intermediate object used by Visual Basic to gather all the controls in the array. In general, you shouldn't use this approach to iterate over all the items in the array because if the array has holes in the Index sequence an error will be raised. A better way to loop over all the items of a control array is using the `For Each` statement:

```
Dim txt As TextBox
For Each txt In txtFields
txt.Text = ""
Next
```

A third method exposed by the control array object, `Count`, returns the number of elements it contains. It can be useful on several occasions (for example, when removing all the controls that were added dynamically at run time):

```
' This code assumes that txtField(0) is the only control that was
' created at design time (you can't unload it at run time).
Do While txtFields.Count > 1
Unload txtFields(txtFields.UBound)
Loop
```

Arrays of Menu Items

Control arrays are especially useful with menus because arrays offer a solution to the proliferation of menu Click events and, above all, permit you to create new menus at run time. An array of menu controls is conceptually similar to a regular control array, only you set the `Index` property to a numeric (non-negative) value in the Menu Editor instead of in the Properties window.

There are some limitations, though: All the items in an array of menu controls must be adjacent and must belong to the same menu level, and their `Index` properties must be in ascending order (even though holes in the sequence are allowed). This set of requirements severely hinders your ability to create new menu items at run time. In fact, you can create new menu items in well-defined positions of your menu hierarchy—namely, where you put a menu item with a nonzero `Index` value—but you can't create new submenus or new top-level menus.

Now that you have a thorough understanding of how Visual Basic's forms and controls work, you're ready to dive into the subtleties of the Visual Basic for Applications (VBA) language.

Working with Menus in Visual Basic 6 (VB6)

Windows applications provide groups of related commands in Menus. These commands depends on the application, but some—such as Open and Save—are frequently found in applications. Menus are intrinsic controls, and as such they deserve a place in this chapter. On the other hand, menus behave differently from other controls. For example, you don't drop menu items on a form from the Toolbox; rather, you design them in the Menu Editor window, as you can see in the figure below. You invoke this tool from the Menu Editor button on the standard toolbar or by pressing the `Ctrl+E` shortcut key. There's also a Menu Editor command in the Tools menu, but you probably won't use it often.

Visual Basic provides an easy way to create menus with the modal Menu Editor dialog. The below dialog is displayed when the Menu Editor is selected in the Tool Menu. The Menu Editor command is grayed unless the form is visible. And also you can display the Menu Editor window by right-clicking on the Form and selecting Menu Editor.

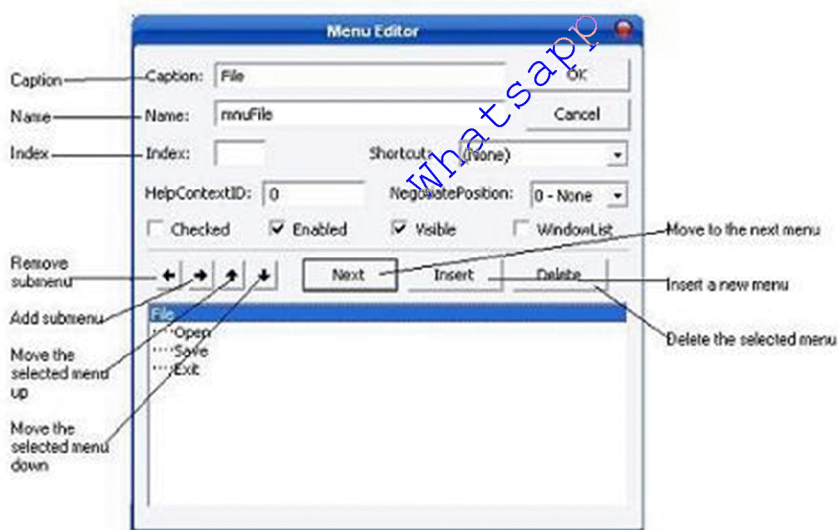
Basically, each menu item has a **Caption** property (possibly with an embedded & character to create an access key) and a **Name**. Each item also exposes three Boolean properties, Enabled, Visible, and Checked, which you can set both at design time and at run time. At design time, you can assign the menu item a shortcut key so that your end users don't have to go through the menu system each time they want to execute a frequent command. (Do you really like pulling down the Edit menu any time you need to clear some text or copy it to the Clipboard?) The assigned shortcut key can't be queried at run time, much less modified.

Building a menu is a simple, albeit more tedious, job: You enter the item's Caption and Name, set other properties (or accept the default values for those properties), and press Enter to move to the next item. When you want to create a submenu, you press the Right Arrow button (or the Alt+R hot key). When you want to return to work on top-level menus—those items that appear in the menu bar when the application runs—you click the Left Arrow button (or press Alt+L). You can move items up and down in the hierarchy by clicking the corresponding buttons or the hot keys Alt+U and Alt+B, respectively.

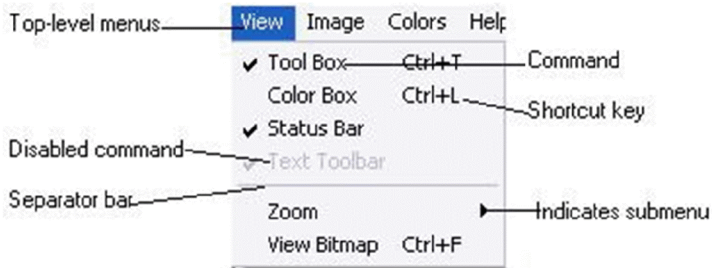
You can create up to five levels of submenus (six including the menu bar), which are too many even for the most patient user. If you find yourself working with more than three menu levels, think about trashing your specifications and redesigning your application from the ground up.

You can insert a separator bar using the hyphen (-) character for the Caption property. But even these separator items must be assigned a unique value for the Name property, which is a real nuisance. If you forget to enter a menu item's Name, the Menu Editor complains when you decide to close it. The convention used in this book is that all menu names begin with the three letters mnu.

An expanded Menu Editor window.

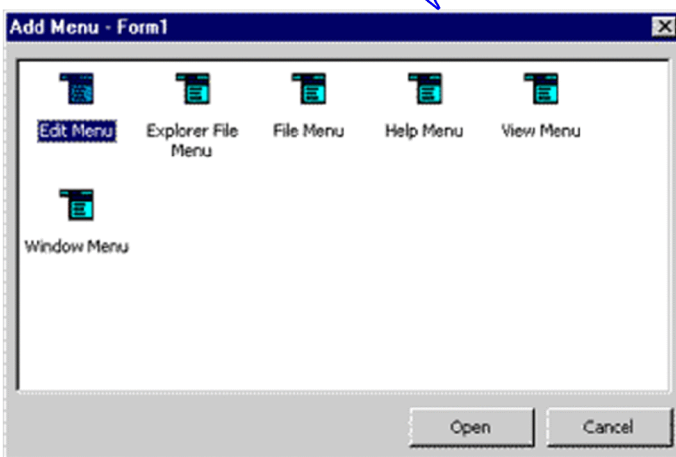


An expanded menu



One of the most annoying defects of the Menu Editor tool is that it doesn't permit you to reuse the menus you have already written in other applications. It would be great if you could open another instance of the Visual Basic IDE, copy one or more menu items to the clipboard, and then paste those menu items in the application under development. You can do that with controls and with pieces of code, but not with menus! The best thing you can do in Visual Basic is load the FRM file using an editor such as Notepad, find the portion in the file that corresponds to the menu you're interested in, load the FRM file you're developing (still in Notepad), and paste the code there. This isn't the easiest operation, and it's also moderately dangerous: If you paste the menu definition in the wrong place, you could make your FRM form completely unreadable. Therefore, always remember to make backup copies of your forms before trying this operation.

Better news is that you can add a finished menu to a form in your application with just a few mouse clicks. All you have to do is activate the Add-In Manager from the Add-Ins menu, choose the VB 6 Template Manager, and tick the Loaded/Unloaded check box. After you do that, you'll find three new commands in the Tools menu: Add Code Snippet, Add Menu, and Add Control Set. Visual Basic 6 comes with a few menu templates, as you can see in the following figure, that you might find useful as a starting point for building your own templates. To create your menu templates, you only have to create a form with the complete menu and all the related code and then store this form in the \Templates\Menus directory. (The complete path, typically c:\Program Files\Microsoft Visual Studio\VB98\Template, can be found in the Environment tab of the Options dialog box on the Tools menu. The Template Manager was already available with Visual Basic 5, but it had to be installed manually and relatively few programmers were aware of its existence.

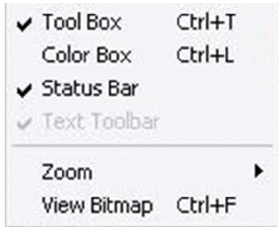


The Template Manager in action

The programmer can create menu control arrays. The Index TextBox specifies the menu's index in the control array.

The Menu Editor dialog also provides several CheckBoxes to control the appearance of the Menu.

Checked : This is unchecked by default and allows the programmer the option of creating a checked menu item(a menu item that act as a toggle and displays a check mark when selected. The following is a Check Menu items.



Enabled : specifies whether a menu is disabled or not. If you see a disabled command in a menu that means that feature is not available. The Visible checkbox specifies whether the menu is visible or not.

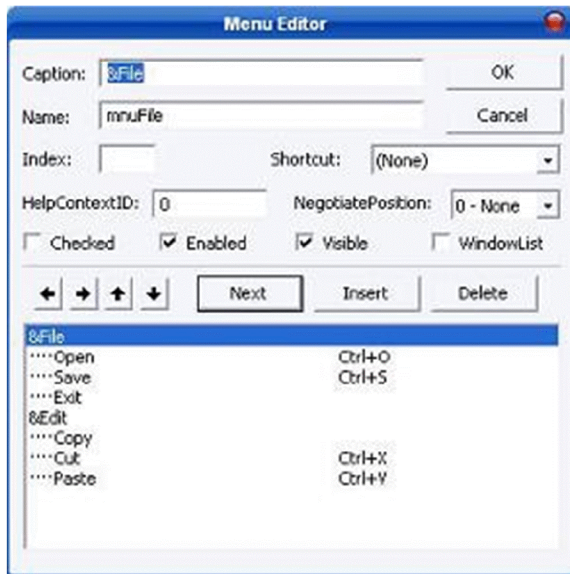
To add commands to the Form's menu bar, enter a caption and a name for each command. As soon as you start typing the command's caption, it also appears in a new line in the list at the bottom of the Menu Editor window. To add more commands click Enter and type the Caption and the Name.

Creating Menus

Open a new Project and save the form as menu.frm and save the project as menu.vbp.

Choose **Tools >>> Menu Editor** and type the menu items as shown below.

<i>Caption</i>	<i>Name</i>
File	mnuFile
Open	mnuOpen
Save	mnuSave
Exit	mnuExit
Edit	mnuEdit
Copy	mnuCopy
Cut	mnuCut
Paste	mnuPaste



Run the application by pressing F5. You can see that you can select a menu.

Accessing Menus at Run Time in Visual Basic 6

Menu controls expose only one event, Click. As you expect, this event fires when the user clicks on the menu:

```
Private Sub mnuFileExit_Click()
Unload Me
End Sub
```

You can manipulate menu items at run time through their Checked, Visible, and Enabled properties. For example, you can easily implement a menu item that acts as a switch and displays or hides a status bar:

```
Private Sub mnuViewStatus_Click()
' First, add or remove the check sign.
mnuViewStatus.Checked = Not mnuViewStatus.Checked
' Then make the status bar visible or not.
staStatusBar.Visible = mnuViewStatus.Checked
End Sub
```

While menu items can be responsible for their own Checked status, you usually set their Visible and Enabled properties in another region of the code. You make a menu item invisible or disabled when you want to make the corresponding command unavailable to the user. You can choose from two different strategies to achieve this goal: You can set the menu properties as soon as something happens that affects that menu command, or you can set them one instant before the menu is dropped down. Let me explain these strategies with two examples.

Let's say that the Save command from the File menu should look disabled if your application has loaded a read-only file. In this case, the most obvious place in code to set the menu Enabled property to False is in the procedure that loads the file, as shown in the code below.

```

Private Sub LoadDataFile(filename As String)
' Load the file in the program.
' ... (code omitted)...
' Enable or disable the menu enabled state according to the file's
' read-only attribute (no need for an If...Else block).
mnuFileSave.Enabled = (GetAttr(filename) And vbReadOnly)
End Sub

```

This solution makes sense because the menu state doesn't change often. By comparison, the state of most of the commands in a typical Edit menu (Copy, Cut, Clear, Undo, and so on) depends on whether any text is currently selected in the active control. In this case, changing the menu state any time a condition changes (because the user selects or deselects text in the active control, for example) is a waste of time, and it also requires a lot of code. Therefore, it's preferable to set the state of those menu commands in the parent menu's Click event just before displaying the menu:

```

Private Sub mnuEdit_Click()
' The user has clicked on the Edit menu,
' but the menu hasn't dropped down yet.
On Error Resume Next
' Error handling is necessary because we don't know if
' the Active control actually supports these properties.
mnuEditCopy.Enabled = (ActiveControl.SelText <> "")
mnuEditCut.Enabled = (ActiveControl.SelText <> "")
mnuEditClear.Enabled = (ActiveControl.SelText <> "")

```

Creating Pop-up Menus in Visual Basic 6

Visual Basic also supports pop-up menus, those context-sensitive menus that most commercial applications show when you right-click on a user interface object. In Visual Basic, you can display a pop-up menu by calling the form's `PopupMenu` method, typically from within the `MouseDown` event procedure of the object:

```

Private Sub List1_MouseDown(Button As Integer, Shift As Integer, _
X As Single, Y As Single)
If Button And vbRightButton Then
' User right-clicked the list box.
PopupMenu mnuListPopup
End If
End Sub

```

The argument you pass to the `PopupMenu` method is the name of a menu that you have defined using the Menu Editor. This might be either a submenu that you can reach using the regular menu structure or a submenu that's intended to work only as a pop-up menu. In the latter case, you should create it as a top-level menu in the Menu Editor and then set its `Visible` attribute to `False`. If your program includes many pop-up menus, you might find it convenient to add one invisible top-level entry and then add all the pop-up menus below it. (In this case, you don't need to make each individual item invisible.) The complete syntax of the `PopupMenu` method is quite complex:

```
PopupMenu Menu, [Flags], [X], [Y], [DefaultMenu]
```

By default, pop-up menus appear left aligned on the mouse cursor, and even if you use a right-click to invoke the menu you can select a command only with the left button. You can change these defaults using the Flags argument. The following constants control the alignment: 0-vbPopupMenuLeftAlign (default), 4-vbPopupMenuCenterAlign, and 8-vbPopupMenuRightAlign. The following constants determine which buttons are active during menu operations: 0-vbPopupMenuLeftButton (default) and 2-vbPopupMenuRightButton. For example, I always use the latter because I find it natural to select a command with the right button since it's already pressed when the menu appears:

PopupMenu mnuListPopup, vbPopupMenuRightButton

The x and y arguments, if specified, make the menu appear in a particular position on the form, rather than at mouse coordinates. The last optional argument is the name of the menu that's the default item for the pop-up menu. This item will be displayed in boldface. This argument has only a visual effect; If you want to offer a default menu item, you must write code in the MouseDown event procedure to trap double-clicks with the right button.

You can take advantage of the x and y arguments in a PopupMenu method to make your program more Windows compliant, and show your pop-up menus over the control that has the focus when the user presses the Application key (the key beside the Windows key on the right side of a typical extended keyboard, such as the Microsoft Natural Keyboard). But remember that Visual Basic doesn't define any key-code constant for this key. Here's how you must proceed:

```
Private Sub List1_KeyDown(KeyCode As Integer, Shift As Integer)
If KeyCode = 93 Then
' The system pop-up menu key has been pressed.
' Show a pop-up menu near the list box's center.
PopupMenu mnuListPopup, , List1.Left + _
List1.Width / 2, List1.Top + List1.Height / 2
End If
End Sub
```

Visual Basic's implementation of pop-up menus has a serious flaw. All Visual Basic TextBox controls react to right-clicks by showing the standard Edit pop-up menu (with the usual commands, such as Undo, Copy, Cut, and so on). The problem is that if you invoke a PopupMenu method from within the TextBox control's MouseDown event, your custom pop-up menu will be displayed only after the standard one, which is obviously undesirable. You can solve it only by resorting to the unorthodox and undocumented technique shown below.

```
Private Sub Text1_MouseDown(Button As Integer, _
Shift As Integer, X As Single, Y As Single)
If Button And vbRightButton Then
Text1.Enabled = False
PopupMenu mnuMyPopup
Text1.Enabled = True
End If
End Sub
```

The Multiple Document Interface (MDI) in Visual Basic 6

The Multiple Document Interface (MDI) was designed to simplify the exchange of information among documents, all under the same roof. **With the main application, you can maintain multiple open windows, but not multiple copies of the application.** Data exchange is easier when you can view and compare many documents simultaneously. Single Document Interface (SDI). In a Single Document Interface, each window is a free-floating window that is contained within a main window and can move anywhere on the screen

You almost certainly use Windows applications that can open multiple documents at the same time and allow the user to switch among them with a mouse-click. Multiple Word is a typical example, although most people use it in single document mode. Each document is displayed in its own window, and all document windows have the same behavior. The main Form, or MDI Form, isn't duplicated, but it acts as a container for all the windows, and it is called the parent window. The windows in which the individual documents are displayed are called Child windows.

An MDI application must have at least two Forms, the parent Form and one or more child Forms. Each of these Forms has certain properties. There can be many child forms contained within the parent Form, but there can be only one parent Form.

The parent Form may not contain any controls. While the parent Form is open in design mode, the icons on the ToolBox are not displayed, but you can't place any controls on the Form. The parent Form can, and usually has its own menu.

To create an MDI application, follow these steps:

1. Start a new project and then choose Project >>> Add MDI Form to add the parent Form.
2. Set the Form's caption to MDI Window
3. Choose Project >>> Add Form to add a SDI Form.
4. Make this Form as child of MDI Form by setting the MDI Child property of the SDI Form to True. Set the caption property to MDI Child window.

Visual Basic automatically associates this new Form with the parent Form. This child Form can't exist outside the parent Form; in the words, it can only be opened within the parent Form.



Parent and Child Menus

MDI Form cannot contain objects other than child Forms, but MDI Forms can have their own menus. However, because most of the operations of the application have meaning only if there is at least one child Form open, there's a peculiarity about the MDI Forms. The MDI Form usually has a menu with two commands to load a new child Form and to quit the application. The child Form can have any number of commands in its menu, according to the application. When the child Form is loaded, the child Form's menu replaces the original menu on the MDI Form

Following example illustrates the above explanation.

* Open a new Project and name the Form as Menu.frm and save the Project as Menu.vbp

* Design a menu that has the following structure.

◇ MDIMenu Menu caption

- MDIOpen opens a new child Form
- MDIExit terminates the application

* Then design the following menu for the child Form

◇ ChildMenu Menu caption

- Child Open opens a new child Form
- Child Save saves the document in the active child Form
- Child Close Closes the active child Form

At design time double click on MDI Open and add the following code in the click event of the open menu.

```
Form1.Show
```

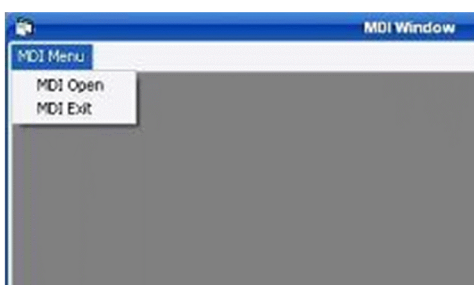
And so double click on MDI Exit and add the following code in the click event

End

Double click on Child Close and enter the following code in the click event

Unload Me

Before run the application in the project properties set MDI Form as the start-up Form. Save and run the application. Following output will be displayed.

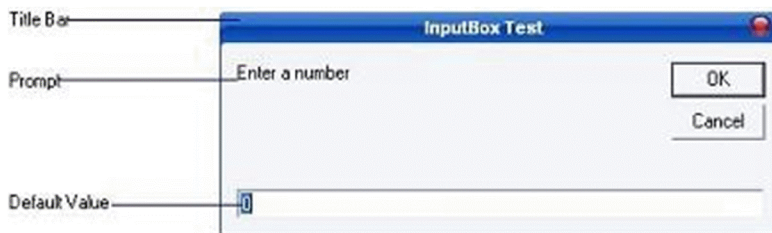


And as soon as you click MDI Open you can notice that the main menu of the MDI Form is replaced with the Menu of the Child Form. The reason for this behavior should be obvious. The operation available through the MDI Form are quite different from the operations of the child window. Moreover, each child Form shouldn't have it's own menu.

InputBox Function in Visual Basic 6 (VB6)

Displays a prompt in a dialog box, waits for the user to input text or click a button, and returns a String containing the contents of the text box.

Following is an expanded InputBox



Syntax :

`memory_variable = InputBox (prompt[,title][,default])`

`memory_variable` is a variant data type but typically it is declared as string, which accept the message input by the users. The arguments are explained as follows:

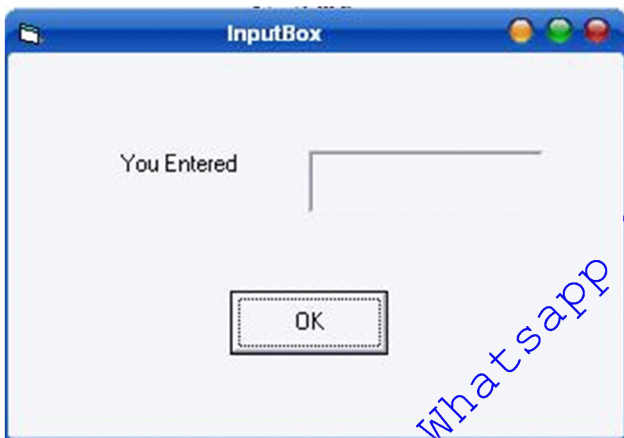
The arguments to the InputBox function are described below:

Argument	Description
<i>prompt</i>	Required. String expression displayed as the message in the dialog box. The maximum length of <i>prompt</i> is approximately 1024 characters, depending on the width of the characters used.
<i>title</i>	Optional. String expression displayed in the title bar of the dialog box. If you omit <i>title</i> , the application name is placed in the title bar.
<i>default</i>	Optional. String expression displayed in the text box as the default response if no other input is provided. If you omit <i>default</i> , the text box is displayed empty.
<i>xpos</i> and <i>ypos</i>	Both optional. Numeric expressions that specify custom positioning of the box on screen (by default, the box is displayed in the center of the screen, which is usually desired).
<i>helpfile</i> and <i>context</i>	Both optional. Can be used if a help file has been set up for the application. If either one of these arguments are used, they both must be used.

* Open a new project and save the Form as InputBox.frm and save the Project as InputBox.vbp

* Design the application as shown below.

Object	Property	Setting
Form	Caption	InputBox test
	Name	frmInputBox
Label	Caption	You entered
	Name	lbl1
Label	Caption	(empty)
	Name	lbl2
	BorderStyle	1-Fixed Single
CommandButton	Caption	OK
	Name	cmdOK



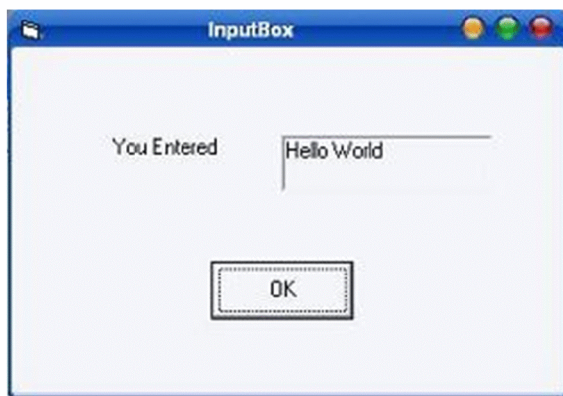
Following code is entered in **cmdOK_Click ()** event

```
Private Sub cmdok_Click()
Dim ans As String
ans = InputBox("Enter something to be displayed in the label", "Testing", 0)
If ans = "" Then
lbl2.Caption = "No message"
Else
lbl2.Caption = ans
End If
End Sub
```

Save and run the application. As soon as you click the OK button you will get the following InputBox



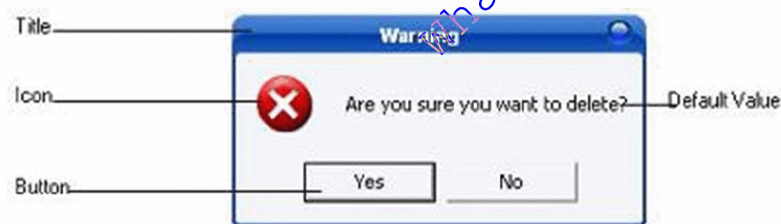
Here I have entered "Hello World" in text field. As soon as you click OK the output is shown as shown below



MessageBox Function in Visual Basic 6 (VB6)

Displays a message in a dialog box and wait for the user to click a button, and returns an integer indicating which button the user clicked.

Following is an expanded MessageBox



Syntax :

MsgBox (Prompt [,icons+buttons] [,title])

memory_variable = **MsgBox** (prompt [, icons+ buttons] [,title])

Prompt : String expressions displayed as the message in the dialog box. If prompt consist of more than one line, you can separate the lines using the **vbrCrLf** constant.

Icons + Buttons : Numeric expression that is the sum of values specifying the number and type of buttons and icon to display.

Title : String expression displayed in the title bar of the dialog box. If you omit title, the application name is placed in the title bar.

Icons

Constant	Value	Description
vbCritical	16	Display Critical message icon
vbQuestion	32	Display Warning Query icon
vbExclamation	48	Display Warning message icon
vbInformation	64	Display information icon

Buttons

Constant	Value	Description
vbOkOnly	0	Display OK button only
vbOkCancel	1	Display OK and Cancel buttons
vbAbortRetryIgnore	2	Display Abort, Retry and Ignore buttons
vbYesNoCancel	3	Display Yes, No and Cancel buttons
vbYesNo	4	Display Yes and No buttons
vbRetryCancel	5	Display Retry and Cancel buttons

Return Values

Constant	Value	Description
vbOk	1	Ok Button
vbCancel	2	Cancel Button
vbAbort	3	Abort Button
vbRetry	4	Retry Button
vbIgnore	5	Ignore Button
vbYes	6	Yes Button
vbNo	7	No Button

Following is an example illustrates the use of message boxes

* Open a new Project and save the Form as messageboxdemo.frm and save the Project as messageboxdemo.vbp

* Design the application as shown below.

Object	Property	Setting
--------	----------	---------

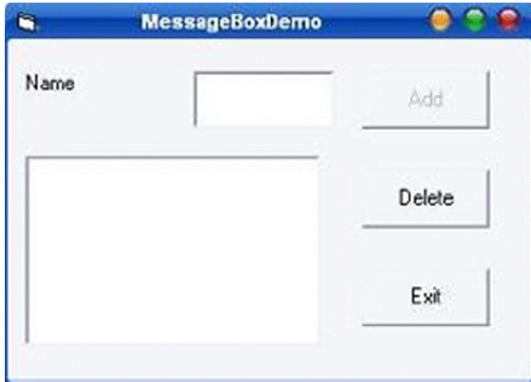
Form	Caption	MessageBoxDemo
	Name	frmMessageBoxDemo
Label	Caption	lblName
	Name	Name
TextBox	Name	txtName
	Text	(empty)
ListBox	Name	lstName
CommandButton	Caption	Add
	Name	cmdAdd
CommandButton	Caption	Delete
	Name	cmdDelete
CommandButton	Caption	Exit
	Name	cmdExit

VB Constants used together with MsgBox Function

Constant	Value	Description
vbCr	Chr(13)	Carriage return
vbCrLf	Chr(13)& Chr(10)	Carriage return-linefeed combination
vbLf	Chr(10)	Line feed

- **vbCr** : - *return to line beginning*
Represents a *carriage-return* character for print and display functions.
- **vbCrLf** : - *similar to pressing Enter*
Represents a *carriage-return* character combined with a *linefeed character* for print and display functions.
- **vbLf** : - *go to next line*
Represents a *linefeed character* for print and display functions.

NB: The three are constants have similar functions nowadays.



Following code is entered in the txtName_Change () event

```
Private Sub txtName_Change()
If Len(txtName.Text) > 0 Then
cmdAdd.Enabled = True
End If
End Sub
```

Following code has to be entered in the cmdAdd_Click () event

```
Private Sub cmdAdd_Click()
answer = MsgBox("Do you want to add this name to the list box?", vbExclamation + vbYesNo,
"Add Confirm")
If answer = vbYes Then
lstName.AddItem txtName.Text
txtName.Text = ""
txtName.SetFocus
cmdAdd.Enabled = False
End If
End Sub
```

Following code is entered in the cmdDelete_Click () event

```

Private Sub cmdDelete_Click()
Dim remove As Integer
remove = lstName.ListIndex
If remove < 0 Then
MsgBox "No names is selected", vbInformation, "Error"
Else
answer = MsgBox("Are you sure you want to delete " & vbCrLf & "the selected name?",_
vbCritical + vbYesNo, "Warning")
If answer = vbYes Then
If remove >= 0 Then
lstName.RemoveItem remove
txtName.SetFocus
MsgBox "Selected name was deleted", vbInformation, "Delete Confirm"

End If
End If
End If
End Sub

```

Following code is entered in the cmdExit_Click () event

```

Private Sub cmdExit_Click()
answer = MsgBox("Do you want to quit?", vbExclamation + vbYesNo, "Confirm")
If answer = vbYes Then
End
Else
MsgBox "Action canceled", vbInformation, "Confirm"
End If
End Sub

```

Save and run the application. You can notice the different type of message box types are used to perform an action

PROGRAM STRUCTURE

Object Naming Conventions of Controls (Prefixes)

OBJECT	PREFIX	EXAMPLE
Form	frm	frmLogin
Label	lbl	lblName
TextBox	txt	txtFirstName
CommandButton	cmd	cmdCalculateAge
CheckBox	chk	chkReadOnly
OptionButton	opt	optGender
ComboBox	cbo	cboFrench
ListBox	lst	lstStudentNames
Frame	fra	fraGender
PictureBox	pic	picMyIcon
Image	img	imgStudentPhoto
Shape	shp	shpCircle
Line	lin	linVerticle
HScrollBar	hsb	hsbVolume
VScrollBar	vsb	Rate
Data	dat	datBiblio
Drive listbox	drv	drvTargetDrive
Grid	grd	grdPrices
Menu	mnu	mnuFileOpen
Timer	tmr	tmrAlarm
OLE	ole	oleObject1

Data Types, Modules and Operators

Visual Basic uses building blocks such as Variables, Data Types, Procedures, Functions and Control Structures in its programming environment. This section concentrates on the programming fundamentals of Visual Basic with the blocks specified.

Modules

Code in Visual Basic is stored in the form of modules. **A module is a collection of declarations, statements, and procedures stored together as one named unit. There are three types of modules: form modules, standard modules and class modules.** A simple application may contain a single Form, and the code resides in that **Form module** itself with .frm extension. As the application grows, additional Forms are added and there may be a common code to be executed in several Forms. To avoid the duplication of code, **a separate module containing a procedure is created that implements the common code.** This is a **standard Module**. (.bas File Extension)

Class module (.CLS filename extension) are the foundation of the object based programming in Visual Basic. **New objects can be created by writing code in class modules.** Each module can contain:

Declarations : May include constant, type, variable and DLL procedure declarations.

Procedures : A sub function, or property procedure that contain pieces of code that can be executed as a unit.

These are the rules to follow when naming elements in VB - variables, constants, controls, procedures, and so on:

- A name must begin with a letter.
- May be as much as 255 characters long.
- Must not contain a space or an embedded period or type-declaration characters used to specify a data type; these are ! # % \$ & @
- Must not be a reserved word (that is part of the code, like Option, for example)
- The dash, although legal, should be avoided because it may be confused with the minus sign. Instead of First-name use First_name or FirstName.

DATA TYPES IN VISUAL BASIC 6.0

By default Visual Basic variables are of **variant** data types. The variant data type can store numeric, date/time or string data. When a variable is declared, a data type is supplied for it that determines the kind of data they can store. The fundamental data types in Visual Basic including variant are integer, long, single, double, string, currency, byte and boolean. Visual Basic supports a vast array of data types. Each data type has limits to the kind of information and the minimum and maximum values it can hold. In addition, some types can interchange with some other types. A list of Visual Basic's simple data types are given below.

1. NUMERIC

Byte	Store integer values in the range of 0 - 255
Integer	Store integer values in the range of (-32,768) - (+ 32,767)
Long	Store integer values in the range of (- 2,147,483,468) - (+ 2,147,483,467)
Single	Store floating point value in the range of (-3.4x1038) - (+ 3.4x1038): 7 significant digits
Double	Store large floating value which exceed the single data type value: 15 significant digits
Currency	Store monetary values. It supports 4 digits to the right of decimal point and 15 digits to the left

2. STRING

Used to store alphanumeric values. A variable length string can store approximately 4 billion characters

3. DATE

Used to store date and time values. A variable declared as date type can store both date and time values and it can store date values 01/01/0100 up to 12/31/9999

4. BOOLEAN

Boolean data types hold either a **true** or **false** value. These are not stored as numeric values and cannot be used as such. Values are internally stored as -1 (True) and 0 (False) and any non-zero value is considered as true.

5. VARIANT

Stores any type of data and is the default Visual Basic data type. In Visual Basic if we declare a variable without any data type or use a variable before declaring it by default the data type is assigned as variant.

Variable Data Types Prefixes

Use the following prefixes to indicate a variable's data type.

Data type	Prefix	Example
Boolean	bln	blnFound
Byte	byt	bytRasterData
Collection object	col	colWidgets
Currency	cur	curRevenue
Date (Time)	dtm	dtmStart
Double	dbl	dblTolerance
Error	err	errOrderNum
Integer	int	intQuantity
Long	lng	lngDistance
Object	obj	objCurrent
Single	sng	sngAverage
String	str	strFName
User-defined type	udt	udtEmployee
Variant	vnt	vntChecksum

Operators in Visual Basic

Arithmetical Operators

Operators	Description	Example	Result
+	Add	5+5	10
-	Subtract	10-5	5
/	Divide	25/5	5

\	Integer Division	20\3	6
*	Multiply	5*4	20
^	Exponent (power of)	3^3	27
Mod	Remainder of division	20 Mod 6	2
&	String concatenation	"George" & "Bush"	"George Bush"

Relational Operators

Operators	Description	Example	Result
>	Greater than	10>8	True
<	Less than	10<8	False
>=	Greater than or equal to	20>=10	True
<=	Less than or equal to	10<=20	True
<>	Not Equal to	5<>4	True
=	Equal to	5=7	False

Logical Operators

<i>Operator</i>	<i>Description</i>	<i>Example</i>	<i>Result</i>
And	Both sides must be true	(2 < 3) And (4 < 5)	True
Or	One side or other must be true	(2 < 3) Or (6 < 7)	True
Xor	One side or other must be true but <i>not both</i>	(2 < 3) Xor (7 > 4)	False
Not	Negates truth	Not (3 = 3)	False

xor - Exclusive OR

OPERATOR PRECEDENCE

Visual Basic follows an order of operators when computing results.

<i>Order</i>	<i>Operators</i>	<i>Example</i>	<i>Result</i>
1	Parentheses ()	$(2 + 3) * 7$	35
2	^	$2 ^ 3 + 1$	9
3	*, /, \, Mod	$2 + 3 * 7$	23
4	+, -	$10 - 4 * 2 + 1$	3

Unless parentheses override the default order, Visual Basic always calculates the intermediate results of any multiplication and division in an expression before the addition and subtraction. Exponentiation has an even higher priority.

If multiplication and division both appear in the same expression, Visual Basic performs them from left to right unless **parentheses override that order**. Therefore, the following expression produces a result of 15 because Visual Basic first divides 10 by 2 to get 5 and then multiplies the 5 by 3 to get 15. In the same way, addition and subtraction compute in their left-to-right order if they appear in an expression without other operators or parentheses changing their behavior.

$10 / 2 * 3$

If you embed one set of parentheses within another, Visual Basic computes the innermost set first. Therefore, Visual Basic computes the $(8 - 3)$ before anything else in the following expression:

$(10 + 2 - (8 - 3)) + 1$

Variables in Visual Basic 6

Variables are the memory locations which are used to store values temporarily. A defined naming strategy has to be followed while naming a variable. A variable name must begin with an alphabet letter and should not exceed 255 characters. It must be unique within the same scope. It should not contain any special character like %, &, !, #, @ or \$.

There are many ways of declaring variables in Visual Basic. Depending on where the variables are declared and how they are declared, we can determine how they can be used by our application. The different ways of declaring variables in Visual Basic are listed below and elucidated in this section.

- Explicit Declaration
- Using Option Explicit statement
- Scope of Variables

Explicit Declaration

Declaring a variable tells Visual Basic to reserve space in memory. It is not must that a variable should be declared before using it. Automatically whenever Visual Basic encounters a new variable, it assigns the default variable type and value. This is called implicit declaration. Though this type of declaration is easier for the user, to have more control over the variables, it is advisable to declare them explicitly. The variables are declared with a Dim statement to name the variable and its type. The As type clause in the Dim statement allows to define the data type or object type of the variable. This is called explicit declaration.

Syntax

Dim variable [As Type]

For example,

```
Dim strName As String  
Dim intCounter As Integer
```

Using Option Explicit statement

It may be convenient to declare variables implicitly, but it can lead to errors that may not be recognized at run time. Say, for example a variable by name *intcount* is used implicitly and is assigned to a value. In the next step, this field is incremented by 1 by the following statement

```
Intcount = Intcount + 1
```

This calculation will result in *intcount* yielding a value of 1 as *intcount* would have been initialized to zero. This is because the *intcount* variable has been mistyped as *incont* in the right hand side of the second variable. But Visual Basic does not see this as a mistake and considers it to be new variable and therefore gives a wrong result.

In Visual Basic, to prevent errors of this nature, we can force the user to declare all his variables by adding the following statement to the general declaration section of the Form.

Option Explicit

This forces the user to declare all the variables. The Option Explicit statement checks in the module for usage of any undeclared variables and reports an error to the user. The user can thus rectify the error on seeing this error message.

The Option Explicit statement can be explicitly placed in the general declaration section of each module using the following steps.

- Click Options item in the Tools menu
- Click the Editor tab in the Options dialog box
- Check Require Variable Declaration option and then click the OK button

Scope of variables

A variable is scoped to a procedure-level (local) or module-level variable depending on how it is declared. The scope of a variable, procedure or object determines which part of the code in our application are aware of the variable's existence. A variable is declared in general declaration section of a Form, and hence is available to all the procedures. Local variables are recognized only in the procedure in which they are declared. They can be declared with *Dim* and *Static* keywords. If we want a variable to be available to all of the procedures within the same module, or to all the procedures in an application, a variable is declared with broader scope.

Local Variables

A local variable is one that is declared inside a procedure. This variable is only available to the code inside the procedure and can be declared using the Dim statements as given below.

Dim sum As Integer

The local variables exist as long as the procedure in which they are declared, is executing. Once a procedure is executed, the values of its local variables are lost and the memory used by these variables is freed and can be reclaimed. Variables that are declared with keyword Dim exist only as long as the procedure is being executed.

Static Variables

Static variables are not reinitialized each time Visual invokes a procedure and therefore retains or preserves value even when a procedure ends. In case we need to keep track of the number of times a command button in an application is clicked, a static counter variable has to be declared. These static variables are also ideal for making controls alternately visible or invisible. A static variable is declared as given below.

Static intPermanent As Integer

Variables have a lifetime in addition to scope. The values in a module-level and public variables are preserved for the lifetime of an application whereas local variables declared with Dim exist only while the procedure in which they are declared is still being executed. The value of a local variable can be preserved using the Static keyword. The following procedure calculates the running total by adding new values to the previous values stored in the static variable value.

```
Function RunningTotal ()  
Static Accumulate  
Accumulate = Accumulate + num  
RunningTotal = Accumulate  
End Function
```

If the variable Accumulate was declared with Dim instead of static, the previously accumulated values would not be preserved across calls to the procedure, and the procedure would return the same value with which it was called. To make all variables in a procedure static, the Static keyword is placed at the beginning of the procedure heading as given in the below statement.

Static Function RunningTotal ()

Example

The following is an example of an event procedure for a CommandButton that counts and displays the number of clicks made.

```
Private Sub Command1_Click ()  
Static Counter As Integer  
Counter = Counter + 1  
Print Counter  
End Sub
```

The first time we click the CommandButton, the Counter starts with its default value of zero. Visual Basic then adds 1 to it and prints the result.

Module Level Variables

A module level variable is available to all the procedures in the module. They are declared using the *Public* or the *Private* keyword. If you declare a variable using a *Private* or a *Dim* statement in the declaration section of a module—a standard BAS module, a form module, a class module, and so on—you're creating a private module-level variable. Such variables are visible only from within the module they belong to and can't be accessed from the outside. In general, these variables are useful for sharing data among procedures in the same module:

```
' In the declarative section of any module
Private LoginTime As Date ' A private module-level variable
Dim LoginPassword As String ' Another private module-level variable
```

You can also use the *Public* attribute for module-level variables, for all module types except BAS modules. (Public variables in BAS modules are global variables.) In this case, you're creating a strange beast: a *Public* module-level variable that can be accessed by all procedures in the module to share data and that also can be accessed from outside the module. In this case, however, it's more appropriate to describe such a variable as a property:

```
' In the declarative section of Form1 module
Public CustomerName As String ' A Public property
```

You can access a module property as a regular variable from inside the module and as a custom property from the outside:

```
' From outside Form1 module...
Form1.CustomerName = "John Smith"
```

The lifetime of a module-level variable coincides with the lifetime of the module itself. Private variables in standard BAS modules live for the entire life of the application, even if they can be accessed only while Visual Basic is executing code in that module. Variables in form and class modules exist only when that module is loaded in memory. In other words, while a form is active (but not necessarily visible to the user) all its variables take some memory, and this memory is released only when the form is completely unloaded from memory. The next time the form is re-created, Visual Basic reallocates memory for all variables and resets them to their default values (0 for numeric values, "" for strings, Nothing for object variables).

Public vs Local Variables

A variable can have the same name and different scope. For example, we can have a public variable named *R* and within a procedure we can declare a local variable *R*. References to the name *R* within the procedure would access the local variable and references to *R* outside the procedure would access the public variable.

CONTROL STRUCTURES IN VISUAL BASIC 6.0

Definition

Control structures represent the forms by which statements in a program are executed. **Flow of control** refers to the order in which the individual statements, instructions or function calls of a program are executed or evaluated.

Importance of Control Structures

Generally, a program should execute the statements one by one until the defined end. This type of a program is called sequential structure. The functionality of this type of program is limited since it flows in single direction. However, all high-level programming languages enable the programmer to change the flow of program execution. This is done by the use of control structures whose main benefits are to enable **decision making** and **repetition** as well as giving the power to do far more **complex processing** and **provide flexibility with logic**. The sophisticated logic is necessary for a program to solve complex problems.

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- continuation at a different statement i.e. unconditional jump e.g. GoTo statements
- executing a set of statements only if some condition is met i.e. choice
- executing a set of statements zero or more times, until some condition is met i.e. loop
- executing a set of distant statements, after which the flow of control usually returns e.g. subroutines/functions

Types of Control Structures

There are three types in C:

1. Sequence structures

Program statements are executed in the sequence in which they appear in the program.

2. Selection structures/Decision Structures

Statement block is executed only if some condition is met. These structures include **if**, **if/else**, and **switch**. Selection structures are extensively used in programming because they allow the program to decide an action based upon user's input or other processes for instance in password checking.

3. Repetition/Iterative/looping structures

This is where a group of statements in a program may have to be executed repeatedly until some condition is satisfied. These include **while**, **do/while** and **for loops**.

Selection Structures/Decision Structures

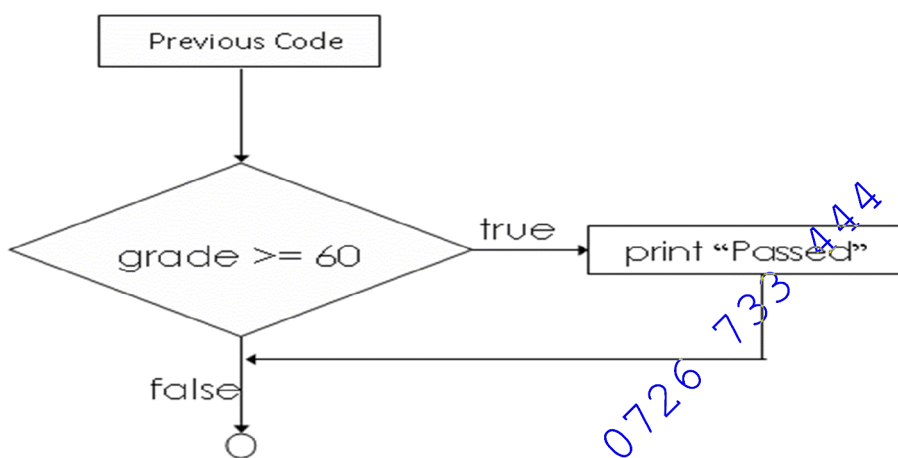
If...Then selection structure

The If...Then selection structure performs an indicated action only when the condition is True; otherwise the action is skipped.

Syntax of the If...Then selection

```
If <condition> Then  
statement  
End If
```

e.g. If average>75 Then
txtGrade.Text = "A"
End If

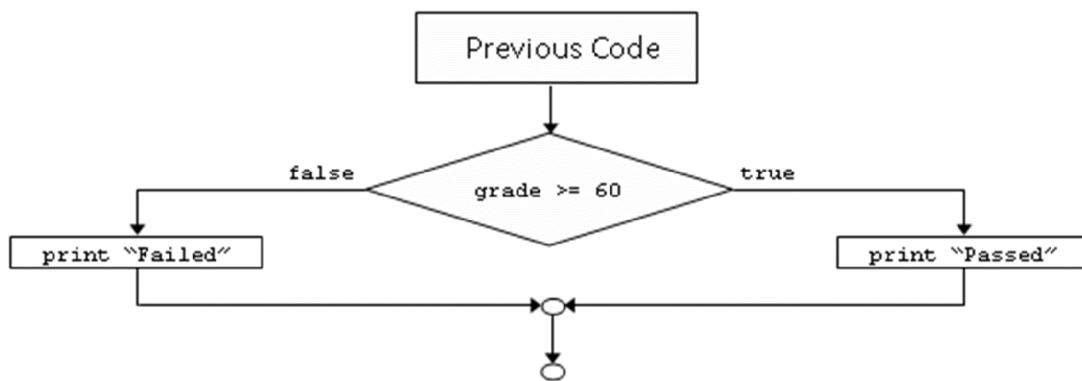


If...Then...Else selection structure

While **if** only performs an action if the condition is **true**, **if/else** specifies an action to be performed both when the condition is **true** and when it is **false**. Syntax of the If...Then...Else selection

```
If <condition > Then  
statements  
Else  
statements  
End If
```

e.g.: If average>50 Then
txtGrade.Text = "Pass"
Else
txtGrade.Text = "Fail"
End If



If...ElseIf...Else selection structure

- These test for multiple cases/conditions.
- Once condition is met, the rest of the statements are skipped
- Deep indentation usually not used in practice

Method 1

```

If < condition 1 > Then
  statements
ElseIf < condition 2 > Then
  statements
ElseIf < condition 3 > Then
  statements
Else
  Statements
End If
  
```

Method 2

```

If < condition 1 > Then
  statements
Else
  If < condition 2 > Then
    statements
  Else
    If < condition 3 > Then
      statements
    Else
      Statements
    End If
  End If
End If
End If
  
```

e.g.: Assume you have to find the grade using nested if and display in a text box

```

If average > 75 Then
txtGrade.Text = "A"
ElseIf average > 65 Then
txtGrade.Text = "B"
ElseIf average > 55 Then
txtGrade.text = "C"
ElseIf average > 45 Then
txtGrade.Text = "S"
Else
txtGrade.Text = "F"
End If

```

Nested IF Structure

Nested If...Then...Else selection structures test for multiple cases by placing If...Then...Else selection structures inside If...Then...Else structures. E.g.

```

Private Sub cmdNestedIf_Click()
Dim strYear As String, intYOB As Integer, intRetirementYear As Integer

strYear = txtYOB.Text

If strYear = "" Then 'to check for null values
MsgBox "Null values not accepted please", vbExclamation
ElseIf Len(strYear) < 4 Or Len(strYear) > 4 Then 'to determine the length of the string
MsgBox ("Please enter a valid figure"), vbExclamation
txtYOB.SetFocus
ElseIf IsNumeric(strYear) = False Then 'test whether value is numeric or not
MsgBox ("Please enter numeric values"), vbExclamation
txtYOB.Text = ""
txtYOB.SetFocus
Else
intYOB = CInt(strYear)
If intYOB >= 2000 Then
MsgBox "Too young to be employed", vbExclamation
Else
intRetirementYear = intYOB + 60
lstResult.AddItem ("You will retire in the year " & intRetirementYear)
End If
End If
End Sub

```

Select...Case selection structure

Select...Case structure is an alternative to If...Then...ElseIf for selectively executing a single block of statements from among multiple block of statements. Select...case is more convenient to use than the If...Else...End If. The following program block illustrate the working of Select...Case.

Syntax of the Select...Case selection structure

```
Select Case Index
Case 0
Statements
Case 1
Statements
Case Else
Statements
End Select
```

e.g.: Assume you have to find the grade using select...case and display in the text box

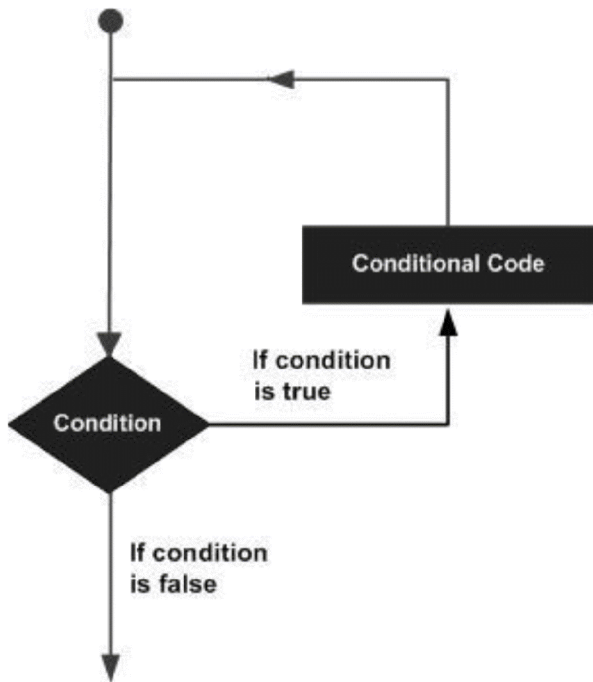
Dim average as Integer

```
average = val(txtAverage.Text)
Select Case average
Case 75 To 100
txtGrade.Text ="A"
Case 65 To 74
txtGrade.Text ="B"
Case 55 To 64
txtGrade.Text ="C"
Case 45 To 54
txtGrade.Text ="D"
Case 44 To 0
txtGrade.Text ="F"
Case Else
MsgBox "Invalid average marks"
End Select
```

Note: In this example I have used a message box function. In later lessons you will learn how to use message box functions.

Loops (Repetition Structures) in Visual Basic 6

This is where a group of statements in a program may have to be executed repeatedly until some condition is satisfied.



Do While... Loop Statement

The **Do While...Loop** is used to execute statements **until a certain condition tests false**. The following Do Loop counts from 1 to 100.

```

Dim number As Integer
number = 1
Do While number <= 3
number = number + 1
Loop
  
```

A variable number is initialized to 1 and then the Do While Loop starts. First, the condition is tested; if condition is True, then the statements are executed. When it gets to the Loop it goes back to the Do and tests condition again. If condition is False on the first pass, the statements are never executed.

While... Wend Statement

A **While...Wend** statement behaves like the **Do While...Loop** statement. The following **While...Wend** counts from 1 to 100

```

Dim number As Integer
number = 1
While number <=100
number = number + 1
Wend
  
```

Do...Loop While Statement

The **Do...Loop While** statement first executes the statements and then tests the condition after each execution. The following program block illustrates the structure:

```
Dim number As Long
number = 0
Do
number = number + 1
Loop While number < 20
```

The program executes the statements between Do and Loop While structure in any case. Then it determines whether the counter is less than 20. If so, the program again executes the statements between Do and Loop While else exits the Loop.

Do Until...Loop Statement

Unlike the **Do While...Loop** and **While...Wend** repetition structures, the **Do Until... Loop** structure tests a condition for **falsity**. Statements in the body of a **Do Until...Loop** are executed repeatedly as long as the loop-continuation test evaluates to False.

An example for **Do Until...Loop** statement. The coding is typed inside the click event of the command button

```
Dim number As Long
number=0
Do Until number > 10
number = number + 1
Print number
Loop
```

Numbers between 1 to 10 will be displayed on the form as soon as you click on the command button.

The For...Next Loop

A **For...Next** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. The following loop counts the numbers from 1 to 50:

```
Dim x As Integer
For x = 1 To 50
Print x
Next
```

In order to count the numbers from 1 to 50 in steps of 2, the following loop can be used

```
For x = 1 To 50 Step 2
Print x
Next
```

The above coding will display numbers vertically on the form. In order to display numbers horizontally the following method can be used.

```

For x = 1 To 50
Print x & Space$ (2);
Next

```

To increase the space between the numbers increase the value inside the brackets after the & Space\$.

Following example is a **For...Next** repetition structure which is with the If condition used.

```

Dim number As Integer
For number = 1 To 10
If number = 4 Then
Print "This is number 4"
Else
Print number
End If
Next

```

In the output instead of number 4 you will get the "This is number 4".

The For Each...Next Loop

This is similar to a For...Next loop, but it repeats a group of statements for each element in a collection of objects or in an array instead of repeating the statements a specified number of times. This is especially helpful if you don't know how many elements are in a collection.

```

For Each element In group
Statements
Next element

```

e.g.

```

Dim object As control

```

```

For Each object In frmLoops
If TypeOf object Is TextBox Then
object.Text = ""
End If
Next object

```

Exit For and Exit Do Statement in Visual basic 6

A **For...Next** loop condition can be terminated by an **Exit For** statement. Consider the following statement block.

```

Dim x As Integer
For x = 1 To 10
Print x
If x = 5 Then
Print "The program exited at x=5"
Exit For
End If
Next

```


The preceding code increments the value of x by 1 until it reaches the condition x = 5. The **Exit For** statement is executed and it terminates the **For...Next** loop. The Following statement block containing **Do...While** loop is terminated using **Exit Do** statement.

```
Dim x As Integer
Do While x < 10
Print x
x = x + 1
If x = 5 Then
Print "The program is exited at x=5"
Exit Do
End If
Loop
```

With...End With statement

When properties are set for objects or methods are called, a lot of coding is included that acts on the same object. It is easier to implement the code using **With...End With** statement. Multiple properties can be set and multiple methods can be called by using the **With...End With** statement. The code is executed more quickly and efficiently as the object is evaluated only once. The concept can be clearly understood with following example.

```
With Text1
.Font.Size = 14
.Font.Bold = True
.ForeColor = vbRed
.Height = 230
.Text = "Hello World"
End With
```

In the above coding, the object Text1, which is a text box is evaluated only once instead of every associated property or method. This makes the coding simpler and efficient.

NB/

A **conditional loop** checks a **condition**, and the **loop** ends if the **condition** is not met. An **unconditional loop**, also called an endless/infinite **loop**, does not check any **condition**, it just repeats forever.

DATA STRUCTURES

These refer to **groups of data elements that are organized in a single unit so that they can be used more efficiently** as compared to the simple variables. Examples of data structures in VB are

1. Arrays
2. User Defined Types(UDTs)
3. Enumerations.

Data structures are important for grouping sets of similar data together and passing them as one unit. For example, if you have a method that prints a set of data but you don't know when writing the procedure how large that set is going to be, you could use an array to pass the data into that method and loop through it.

1. Arrays

An array is a list of variables, all with the same data type and name. It can also be defined as a consecutive group of memory locations all of which have the same name and the same data type. To distinguish between the individual variables in the list, the element's subscript is used. A subscript is the numeric index value of the elements in an array.

Arrays have upper and lower bounds and the elements have to lie within those bounds. Each index number in an array is allocated individual memory space and therefore users must evade declaring arrays of larger size than required. We can declare an array of any of the basic data types including variant, user-defined types (UDT) and object variables.

The following variables are individual variables and are not part of an array:

intCount curPay sngLength strDeptCode

The amount of memory required by an array depends on the length of the array and the size of the data type of the elements in the array.

If you need to store individual values in variables, such as a bonus amount for a salesperson, individual variables work well. However, if you need to store a list of similar values, you'll want to use an array.

For instance, if you had to process 100 salespeople's bonus payment statistics and needed to find the average bonus, the maximum, the minimum, and the standard deviation between the values? One way to keep track of all 100 salespeople's bonus amounts is to declare 100 variables, all with different names, such as curBonus1, curBonus2, and so on. Think of how you would add together those bonuses! You would have to list, every time you added or compared the bonuses, 100 variable names inside a single statement. Such a situation lends itself perfectly to arrays. Instead of 100 variable names, you only need to declare one array. The array will contain the list of 100 bonuses. The array has only one name. To step through the array values, you don't need a list of 100 variable names, you only need a simple For loop, which steps through the subscripts from 1 to 100.

To make things even clearer, here's a fragment of the code used to add 100 separate variables together if they had different names:

`curTotal = curBonus1 + curBonus2 + curBonus3 + curBonus4 + ...`

Here's the code that adds together 100 elements from an array:

```
For intCtr = 1 To 100
curTotal = curTotal + curBonus(intCtr)
Next intCtr
```

NB: curBonus is the name of the array.

You can also loop backwards through the bonuses if you want. In addition, you never have to access the array elements in order. Suppose to want to add the first and last bonuses in the array. The following statement does just that:

```
curAvgBonus = (curBonus(1) + curBonus(100))
```

Just from this code, you can glean the following information about arrays:

- **Array subscripts go inside parentheses after the array name.**
- **All elements of an array have the same data type.**
- **You can access any array element in any order by specifying its subscript.**
- **The subscripts range from 1 to the total elements (nth value) in the array.**

All subscripts begin at 0 unless you insert the following statement in the module's Declarations section:

```
Option Base 1
```

There are two types of arrays in Visual Basic namely:

- **Fixed-size (static) array:** The size of array always remains the same-size - doesn't change during the program execution.
- **Variable-length (Dynamic) array:** The size of the array can be changed at run time - size changes during the program execution.

Fixed-size arrays are much more efficient and more utilized in Visual Basic programming.

Fixed-sized (Static) Arrays

When an upper bound is specified in the declaration, a Fixed-array is created. The upper limit should always be within the range of the long data type.

Declaring and Initializing a Fixed-size Array

As with individual variables, you must tell Visual Basic that you're going to use an array by declaring the array before you use it. You use the Public, Private or Dim statement to declare arrays just as you would use them for declaring individual variables.

Public/Dim arName(Sub) As dataType

Dim arBonus (5) As Currency

Dim arBonus(1 To 5) as Currency

Name arrays just as you would regular variables (with the exception of using the extra ar prefix characters if you wish). You can create an array of any data type, so dataType can be Integer, Single, or any of the data types with which you're familiar. The intSub portion of the commands describes the number of elements and how you refer to those array elements.

The following statement declares an integer array with five elements (assume an Option Base 1 statement appears in the Declarations section):

Dim intCounts(5) As Integer

The 5 tells Visual Basic the array's upper bound or highest-numbered element in the array. Without the Option Base 1 statement, the Dim statement that declares the intCounts array would have declared a six-element array with the subscripts 0 through 5.

If we want to specify the lower limit, then the parentheses should include both the lower and upper limit along with the To keyword. An example for this is given below.

Dim numbers (0 To 5) As Integer

Like any other variable, the program must initialize the array elements before they are used. As long as you use the subscript, you can use array elements anywhere you use other variables, as in the following statements:

intCounts(0) = 3

intCounts(1) = 2

intCounts(2) = 1

intCounts(3) = 5

intCounts(4) = 6

Array Declaration	Notes
Dim aintCount(9) As Integer	declares a 10-element array, indexed 0 to 9
Dim aintCount(0 To 9) As Integer	same as above, with explicit lower bound
Dim aintCount(1 To 10) As Integer	declares a 10-element array, indexed 1 To 10
Dim aintCount(3 To 12) As Integer	declares a 10-element array, indexed 3 To 12
Dim aintCount(-4 To 5) As Integer	declares a 10-element array, indexed -4 To 5
Dim aintCount() As Integer	declares a variable-length array whose bounds will be determined at run-time

Variable-Length (Dynamic) Arrays

When you declare a static array (specifying an array's upper bound), you are committed to the specified number of elements for the duration of the program. However, you can change the number of elements of a dynamic array at runtime. You declare a dynamic array by omitting the specification of the bounds in the parentheses of the declaration e.g.

Dim Names() As String

This array initially has no elements. You may resize your dynamic array later in the program with the use of the ReDim keyword:

ReDim Names(0 to 1) As String

This would create two new elements. You can assign names to these elements in the same way as before:

```
Names(0) = "Alec"
```

```
Names(1) = "Jack"
```

Multidimensional Arrays

An array can be one dimensional or multidimensional. A one dimensional array is like a list of items or a table that consists of one row of items or one column of items. A two dimensional array is a table of items that is made up of rows and columns.

Continuing our table analogy, declaring a multidimensional array requires that both the number of rows and columns be defined when the array is created.

Dim strBooks(4, 2) As String

The above code excerpt creates a two dimensional String array of 5 rows and 2 columns.

A common use of multidimensional arrays is to represent tables of values consisting of information arranged in rows and columns. To identify a particular table element, we must specify two indexes: The first (by convention) identifies the element's row and the second (by convention) identifies the element's column.

Tables or arrays that require two indexes to identify a particular element are called two dimensional arrays. Note that multidimensional arrays can have more than two dimensions. Visual Basic supports at least 60 array dimensions, but most people will need to use more than two or three dimensional-arrays.

It is also possible to define the lower limits for one or both the dimensions as for fixed size arrays. An example for this is given here.

Dim Marks (101 To 200, 1 To 100) as Integer

One dimensional Array

Student Name	Name(1)	Name(2)	Name(3)	Name(4)	Name(5)	Name(6)
--------------	---------	---------	---------	---------	---------	---------

Two Dimensional Array

Name(1,1)	Name(1,2)	Name(1,3)	Name(1,4)
Name(2,1)	Name(2,2)	Name(2,3)	Name(2,4)
Name(3,1)	Name(3,2)	Name(3,3)	Name(3,4)

As in

Jane	Mary	Damaris	Ann
Amolo	Oloo	Onyango	Atieno
Nyangweso	Nyaguthie	Omtata	Maina

To refer to an element of a two-dimensional array, a line of code might read:

arNames(2,3) which would refer to Onyango

The following code uses a two dimensional array to hold eight records, two rows and three columns of elements. i.e.

00 01 02 03

10 11 12 13

Private Sub cmd2Dim1_Click()

Dim intCtrl1 As Integer, intCtrl2 As Integer

Dim cdRack(0 To 1, 0 To 3) As String

' Initialize the array with 8 records

cdRack(0, 0) = "Rubber Soul"

cdRack(0, 1) = "Revolver"

cdRack(0, 2) = "The White Album"

cdRack(0, 3) = "Let It Be"

cdRack(1, 0) = "Sticky Fingers"

cdRack(1, 1) = "Beggars Banquet"

cdRack(1, 2) = "Let It Bleed"

cdRack(1, 3) = "Tattoo You"

For intCtrl1 = 0 To 1

For intCtrl2 = 0 To 3

Print cdRack(intCtrl1, intCtrl2)

Next intCtrl2

Next intCtrl1

End Sub

Whatsapp 0726 733 444

Searching Arrays

When working with arrays, it is often necessary to perform a search or "lookup" to determine whether an array contains a value that matches a certain key value. The process of locating a particular element value in an array is called searching. There are two types of search mechanisms: **serial/linear search** and **binary search**

a) Serial Search

Private Sub cmdSearch_Click()

' This sub uses the serial search algorithm to sort an array of integers.

Dim myInteger As Integer

Dim intSearchValue As Integer

Dim myArray(5) As Integer 'create the array

myArray(0) = 1

myArray(1) = 2

myArray(2) = 8

myArray(3) = 4

myArray(4) = 5

myArray(5) = 6

'code to search the array

intSearchValue = InputBox("Please type an integer to search")

For myInteger = 0 To 5

If intSearchValue = myArray(myInteger) Then

Print " A match for " + CStr(intSearchValue) + " found" & " at position " & myInteger + 1

Exit For

End If

Next

'Output if there is no match

If myInteger = 6 Then 'if looped 6 times meaning beyond the size of loop

Print " No match found"

End if

End Sub

The technique used here is called a **serial search**, because the integer elements of the array are compared one by one to the user input being looked for (userValue) until either a match is found or all elements of the array are examined without finding a match.

In the code above, if a match is found, the text “There is a match” is printed on the form and the loop exited. If no match is found, the program exits the loop and prints the text “No match found” if the variable for controlling the loop is equal to a value above the size of the array.

b) Binary Search

Binary search uses the concept of splitting your searchable array in two, discarding the half that does not have the element for which you are looking.

You place your items in an array and sort them. Then you simply get the middle element and test if it is <, >, or = to the element for which you are searching. If it is less than, you discard the greater half, get the middle index of the remaining elements and do it again. Binary search divides your problem in half every time you execute your loop.

Example

```
Private Sub cmdBinarySearch_Click()
```

```
    Dim myArray(5) As Integer 'create the array
```

```
    Dim low As Integer
```

```
    low = 0
```

```
    Dim high As Integer
```

```
    high = UBound(myArray)
```

```
    Dim i As Integer
```

```
    Dim intSearchValue As Integer
```

```
    myArray(0) = 1
```

```
    myArray(1) = 2
```

```
    myArray(2) = 8
```

```
    myArray(3) = 4
```

```
    myArray(4) = 5
```

```
    myArray(5) = 6
```

```
    intSearchValue = InputBox("Please type an integer to search")
```

```
    Do While low <= high
```

```
        i = (low + high) / 2
```

```
        If intSearchValue = myArray(i) Then
```

```
            result = True
```

```
            Print "There is a match"
```

```
            Exit Do
```

```
        ElseIf intSearchValue < myArray(i) Then
```

```
            high = (i - 1)
```

```
        Else
```

```
            low = (i + 1)
```

```
        End If
```

```
    Loop
```

```
    If low > high Then
```

whatsapp 0726 733 444

Print "No match"

End If

End Sub

Sorting Arrays

Sorting refers to arranging data into a particular order e.g. ascending or descending order. Bubble Sort, insertion sort and Shell Sort are some of the well-known sorting algorithms.

a) Bubble Sort

The algorithm scans through an array of elements. If the first index is greater than the second index the algorithm swaps the values. The procedure is repeated until all values are arranged in order. The technique uses nested loops to **make several passes through the array**. Each pass compares adjacent pairs of elements until the whole list is sorted. If a pair of values is equal, the bubble sort leaves the values as they are. The algorithm gets its name from the way smaller elements "bubble" to the top of the list.

A generalized Sub routine that implements the "pure" bubble sort algorithm to sort an array of integers is shown below. Note that nested For/Next loops are used to step through the array. The starting loop index for each For loop is the lower bound of the array to be sorted (LBound(myArray)). If the array was declared with the boundaries 1 To 5, then the lower bound would be 1. The ending loop index for each For loop is the upper bound of the array to be sorted, less 1 i.e. (UBound(myArray) - 1). Again, if the array was declared with the boundaries 1 To 5, then the upper bound less 1 would be 4. Note that the body of the inner loop performs the comparison of the current element of the array with its adjacent "neighbor", and if the value of the current element is greater than the adjacent element, the elements are exchanged.

Note the coding required to implement the exchange of items: the current item of the array (myArray (lngY)) is assigned to a temporary, or "hold" variable (intTemp). The adjacent item of the array (myArray (lngY + 1)) is then copied to the current item (at this point, both the current and the adjacent items contain the same value). The temporary variable is then assigned to the adjacent item, completing the exchange.

```
Private Sub cmdSort_Click()
```

```
'This procedure uses the Bubble Sort algorithm to sort an array of integers.
```

```
Dim intCtrl As Integer
```

```
Dim myArray(3) As Integer 'create the array
```

```
Dim lngX As Long
```

```
Dim lngY As Long
```

```
Dim intTemp As Integer
```

```
'get integers from user into the array
```

```
For intCtrl = 0 To 3
```

```
myArray(intCtrl) = InputBox("Enter an integer")
```

```
Next intCtrl
```

'Nested loops to sort integers (Bubble sort) in an ascending order

```
For lngX = LBound(myArray) To (UBound(myArray) - 1)
```

```
    For lngY = LBound(myArray) To (UBound(myArray) - 1)
```

```
        If myArray(lngY) > myArray(lngY + 1) Then
```

```
            ' exchange the items
```

```
            intTemp = myArray(lngY)
```

```
            myArray(lngY) = myArray(lngY + 1)
```

```
            myArray(lngY + 1) = intTemp
```

```
        End If
```

```
    Next
```

```
Next
```

'Loop to print sorted array elements

```
For intCtrl = 0 To 3
```

```
Print myArray(intCtrl)
```

```
Next
```

```
End Sub
```

NB: The outer loop will go to the first element (0) in the array and before lngY is incremented to 1, the inner loop executes by going also to the first element (0). The IF selection tests whether the first element is greater than the second element (myArray(lngY + 1)) and if true interchanges the two values. The inner loop iterates until it reaches the second last element (UBound(myArray) - 1) of the array and then cedes control to the outer loop. The outer loop then moves on to the second element which is then compared with the third element and this way, the loops make several passes over the elements until they are fully sorted.

b) INSERTION SORT

Insertion sort keeps making the left side of the array sorted until the whole array is sorted. It sorts the values seen far away and repeatedly inserts unseen values in the array into the left sorted array.

Although it has the same complexity as Bubble Sort, the insertion sort is a little over twice as efficient as the bubble sort.

Example

Original sequence : 34 8 64 51 32 21

The algorithm sees that 8 is smaller than 34 so it swaps.

- 8 34 64 51 32 21

51 is smaller than 64, so they swap.

- 8 34 51 64 32 21

The algorithm sees 32 as another smaller number and moves it to its appropriate location between 8 and 34.

- 8 32 34 51 64 21

The algorithm sees 21 as another smaller number and moves into between 8 and 32.

Final sorted numbers:

- 8 21 32 34 51 64

```
Public Sub cmdInsertionSort_Click()
```

```
    Dim t As Variant
```

```
    Dim i As Long
```

```
    Dim j As Long
```

```
    ' sort A[Lb..Ub]
```

```
    For i = Lb + 1 To Ub
```

```
        t = A(i)
```

```
        ' shift elements down until insertion point is found
```

```
        For j = i - 1 To Lb Step -1
```

```
            If A(j) <= t Then Exit For
```

```
            A(j + 1) = A(j)
```

```
        Next j
```

```
        ' insert
```

```
        A(j + 1) = t
```

```
    Next i
```

```
End Sub
```

Whatsapp 0726 733 444

C) SHELLSORT

ShellSort works by comparing elements that are distant rather than adjacent in an array or list where adjacent elements are compared.

Shellsort makes multiple passes through a list and sorts a number of equally sized sets using the insertion sort.

Shellsort improves on the efficiency of insertion sort by *quickly* shifting values to their destination.

The distance between comparisons decreases as the sorting algorithm runs until the last phase in which adjacent elements are compared

Advantage of Shellsort is that it's only efficient for medium size lists. For bigger lists, the algorithm is not the best choice. Fastest of all $O(N^2)$ sorting algorithms.

5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor.

As the example illustrates, the sub-arrays that Shellsort operates on are initially short; later they are longer but almost ordered. In both cases insertion sort works efficiently.

Shellsort is **unstable**: it may change the relative order of elements with equal values. It has "natural" behavior, in that it executes faster when the input is partially sorted.

Control Arrays

A collection of controls on a form that is used to provide capabilities for sharing event handlers, iterating through groups of controls and adding controls at run time.

A control array is nothing more than a list of controls, just as a variable array is a list of variables. The advantage of using a control array is the same as for using a variable array: You can step through several controls using a loop instead of having to name each individual control.

3142

Arrays within UDTs

UDT structures can include both static and dynamic arrays. Here's a sample structure that contains both types:

```
Type MyUDT
StaticArr(100) As Long
DynamicArr() As Long
End Type
...
Dim udt As MyUDT
' You must DIMension the dynamic array before using it.
ReDim udt.DynamicArr(100) As Long
' You don't have to do that with static arrays.
udt.StaticArr(1) = 1234
```

The memory needed by a static array is allocated within the UDT structure; for example, the StaticArr array in the preceding code snippet takes exactly 400 bytes. Conversely, a dynamic array in a UDT takes only 4 bytes, which form a pointer to the memory area where the actual data is stored. Dynamic arrays are advantageous when each individual UDT variable might host a different number of array items. As with all dynamic arrays, if you don't dimension a dynamic array within a UDT before accessing its items, you get an error 9—"Subscript out of range."

2. User-Defined Data Types in Visual Basic 6

Variables of different data types when combined as a single variable to hold several related pieces of information is called a User-Defined data type.

The following rules apply to UDTs:

- ✓ UDTs may be declared only at the module-level (you may not declare a UDT in an individual Sub or Function)
- ✓ UDTs may have Public (project-level) or Private (module-level) scope. If the keyword Public or Private is omitted, the default is Public.
- ✓ UDTs with Public scope may only be defined in standard modules, not form modules.

A Type statement is used to define a user-defined type in the General declaration section of a form or module. User-defined data types can only be private in a form while in standard modules they can be public or private. An example of a user defined data type to hold the product details is given below.

```
Private Type ProductDetails
ProdID as String
ProdName as String
Price as Currency
End Type
```

The user defined data type can be declared with a variable using the Dim statement as in any other variable declaration statement. An array of these user-defined data types can also be declared. An example to consolidate these two features is given below.

```
Dim ElectronicGoods as ProductDetails ' One Record
Dim ElectronicGoods(10) as ProductDetails ' An array of 11 records
```

A User-Defined data type can be referenced in an application by using the variable name in the procedure along with the item name in the Type block. Say, for example if the text property of a TextBox namely text1 is to be assigned the name of the electronic good, the statement can be written as given below.

```
Text1.Text = ElectronicGoods.ProdName
```

If the same is implemented as an array, then the statement becomes

```
Text1.Text = ElectronicGoods(i).ProdName
```

User-defined data types can also be passed to procedures to allow many related items as one argument.

```
Sub ProdData( ElectronicGoods as ProductDetails)
Text1.Text = ElectronicGoods.ProdName
Text1.Text = ElectronicGoods.Price
End Sub
```

3. ENUMERATIONS (Enums)

An Enumeration is basically a new data type (like String or Integer) that are user defined, which has an associated list of possible values (known as elements).

You create an enumeration with the Enum...End Enum statement:

```
Public Enum EnumName
    ConstantName1
    ConstantName2
    ...
    ContantNameN
End Enum
```

EnumName is the name of the enumeration following VB's usual variable naming rules. Each constant name (you can have as many as you like) becomes a member of the enumeration. By default, the constants are automatically assigned numerical values in order, starting with 0.

- ✓ The list of valid options will be shown in a pop-up list when you are writing code so you don't need to remember them or even type them - instead you can just select them with your mouse or keyboard.
- ✓ It is harder to make a typing error that causes bugs, as not only are the values shown in the list for you to pick from, but also valid entries you type will be changed to the case you declared them - so you can see when the case is not automatically corrected for you.
- ✓ If you use Option Explicit any typing error's will cause a message warning you about them.
- ✓ They take less memory than Strings, and code that uses them runs faster than the String equivalent.

To set up an Enumeration, use code like this in the "General" - "Declarations" section of a code file (at the very top):

Code:

```
Private Enum MyValues ' this is the name of our Enumeration
    First    '
    Second   ' these are the elements/values that will be allowed
    Third    ' (add as many as you like, there must be at least one!)
End Enum
```

To use it, declare a variable (or parameter, etc) using the Enum as the data type, and then when you are typing you will see a list of possible elements that you can select from (or type if you prefer):

```
Dim MyVariable As MyValues
MyVariable =
```

```
End Sub
```



To work with them, you can use the element names just as you would usually use a variable/const, eg:

```
Dim MyVariable As MyValues
MyVariable = First

If MyVariable = Second Then
    MsgBox "This message will not be shown!"
End If
```

Constants, Data Type Conversion, Visual Basic Built-in Functions

Constants

Constants are named storage locations in memory, the value of which does not change during program Execution. They remain the same throughout the program execution. When the user wants to use a value that never changes, a constant can be declared and created. The Const statement is used to create a constant. Constants can be declared in local, form, module or global scope and can be public or private as for variables. Constants can be declared as illustrated below.

Public Const gravityconstant As Single = 9.81

Keyword Used to Declare the Constant:	General Declarations Section of a Form (.frm) Module	General Declarations Section of a Standard (.bas) Module	Sub or Function procedure of a Form or Standard Module
Const	module-level scope	module-level scope	local-level scope
Private Const	module-level scope	module-level scope	not allowed
Public Const or Global Const	not allowed	project-level scope	not allowed

Predefined Visual Basic Constants

The predefined constants can be used anywhere in the code in place of the actual numeric values. This makes the code easier to read and write.

For example consider a statement that will set the window state of a form to be maximized.

Form1.Windowstate = 2

The same task can be performed using a Visual Basic constant

Form1.WindowState = vbMaximized

Data Type Conversion

Visual Basic functions either to convert a string into an integer or vice versa and many more conversion functions. A complete listing of all the conversion functions offered by Visual Basic is elucidated below.

Conversion To	Function
Boolean	Cbool
Byte	Cbyte
Currency	Ccur
Date	Cdate
Decimals	Cdec

Double	CDBl
Integer	Cint
Long	CLng
Single	CSng
String	CStr
Variant	Cvar
Error	CVErr

A conversion function should always be placed at the right hand side of the calculation statement.

Visual Basic Built-in Functions

Many built-in functions are offered by Visual Basic fall under various categories. These functions are procedures that return a value. The functions fall into the following basic categories that will be discussed in the following sections at length.

- Date and Time Functions
- Format Function
- String Functions

Date and Time Functions in Visual Basic 6

Not only does Visual Basic let you store date and time information in the specific Date data type, it also provides a lot of date- and time-related functions. These functions are very important in all business applications and deserve an in-depth look. Date and Time are internally stored as numbers in Visual Basic. The decimal points represents the time between 0:00:00 and 23:59:59 hours inclusive.

The system's current date and time can be retrieved using the Now, Date and Time functions in Visual Basic. The Now function retrieves the date and time, while Date function retrieves only date and Time function retrieves only the time.

To display both the date and time together a message box is displayed use the statement given below.

MsgBox "The current date and time of the system is" & Now

Here & is used as a concatenation operator to concentrate the string and the Now function. Selective portions of the date and time value can be extracted using the below listed functions.

Function	Extracted Portion
Year ()	Year (Now)
Month ()	Month (Now)
Day ()	Day (Now)

WeekDay ()	WeekDay (Now)
Hour ()	Hour (Now)
Minute ()	Minute (Now)
Second ()	Second (Now)

The calculation and conversion functions related to date and time functions are listed below.

Function	Description
DateAdd ()	Returns a date to which a specific interval has been added
DateDiff ()	Returns a Long data type value specifying the interval between the two values
DatePart ()	Returns an Integer containing the specified part of a given date
DateValue ()	Converts a string to a Date
TimeValue ()	Converts a string to a time
DateSerial ()	Returns a date for specified year, month and day

DateDiff Function

The DateDiff function returns the intervals between two dates in terms of years, months or days. The syntax for this is given below.

DateDiff (interval, date1, date2[, firstdayofweek[, firstweekofyear]])

Format Function

The format function accepts a numeric value and converts it to a string in the format specified by the format argument. The syntax for this is given below.

Format (expression[, format[, firstdayofweek[, firstweekofyear]]])

The Format function syntax has these parts:

Part	Description
Expression	Required any valid expression
format	Optional. A valid named or user-defined format expression.
firstdayofweek	Optional. A contant that specifies the first day of the week.
firstweekofyear	Optional. A contant that specifies the first week of the year

SUB-PROGRAMS

A procedure performs a specific task in a program and can be called within another procedure.

A Sub program is:

- a part of a program that performs **one or more** related tasks
- has its own name
- written as an independent part of the program

Benefits of using sub-procedures in programming are:

- Sub-programs help to break programs into several but logical sections. The smaller programs/routines make **programming, debugging** and subsequent **maintenance** easier.
- They also help in **coding repeated operations** such as frequently used calculations, text etc thus making programming less repetitive and faster.
- Procedures used in one program can act as building blocks for other programs with slight modifications – **code reuse**.
- Programmers working on large project can divide the workload by working on different functions.

A Procedure can be Sub, Function or Property Procedure.

The general form of a sub-program

```
Public/Private Sub ProcedureName (optional parameters)
    Statements(s)
End Sub
```

We can **invoke** a sub procedure with the **call** statement

```
Call ProcedureName (optional arguments)
```

A simple example of a sub procedure:

```
Private Sub ShowMessage()
MsgBox (" Hello There !"), vbInformation
End Sub
```

To call the sub procedure elsewhere,

```
Private Sub cmdSubPro_Click()
Call showMessage
End Sub
```

All calls from outside the form module must point to the form module containing the procedure. If the procedure named showMessage is in a form module called Form1, then you can call the procedure in Form1 from another form by using this statement:

Call Form1.showMessage(arguments)

Types of Sub-Procedures

According to **scope**, there are two types of sub-procedures: **Private and public**. **By default**, procedures are **public**—they can be called from any other procedure in any module within your application. On the other hand, procedures that have been declared **private** can only be referenced by other procedures within the same module. E.g.

```
Public Sub ReturnVersion(varCurrentAppVersion As Variant)
```

```
    statements
```

```
End Sub
```

As in

```
Public Sub SquareArea()
```

```
    Dim dblSide As Double
```

```
    Dim dblArea As Double
```

```
    dblSide = Val(txtSide.Text)
```

```
    dblArea = dblSide * dblSide
```

```
    txtArea.Text = dblArea
```

```
End Sub
```

Which can be called as in

```
Private Sub cmdCalcArea_Click()
```

```
    Call Form1.SquareArea
```

```
End Sub
```

According to how procedures are invoked, there are two types of Sub-procedures namely **general procedures** and **event procedures**.

1. Event Procedures

These are invoked when the related event occurs for example the click event. An event procedure contains the control's actual name, an underscore(_), and the event name. Event Procedures are Private by default.

```
Private Sub cmdCountUp_Click()
```

```
    Static count As Integer
```

```
count = count + 1
Label1.Caption = count
End Sub
```

2. General Procedures

A general procedure is declared when several event procedures need to perform the same actions. It is a good programming practice to write common statements in a separate procedure (general procedure) and then call them in the event procedure. E.g.

```
Public Sub SquareArea()
    Dim dblSide As Double
    Dim dblArea As Double

    dblSide = Val(txtSide.Text)
    dblArea = dblSide * dblSide

    txtArea.Text = dblArea
End Sub
```

A general procedure can either be private or public. A public procedure can be invoked outside the module while a Private procedure can be invoked only from within the module.

Function Procedures

Functions are like sub procedures, except they return a value to the calling procedure. They are especially useful for taking one or more pieces of data, called arguments and performing some tasks with them. Then the function returns a value that indicates the results of the tasks complete within the function. The following function procedure calculates the third side or hypotenuse of a right triangle, where A and B are the other two sides. It takes two arguments A and B (of data type Double) and finally returns the results.

```
Function Hypotenuse (A As Double, B As Double) As Double
    Hypotenuse = sqr (A^2 + B^2)
End Function
```

Or

```
Private Function computeVAT(curItemPrice As Currency) As Currency
    computeVAT = curItemPrice * 0.16
End Function
```

The above function procedure is written in the general declarations section of the Code window. A function can also be written by selecting the *Add Procedure* dialog box from the Tools menu and by choosing the required scope and type.

CALLING THE FUNCTION

```
Private Sub cmdTaxCalc_Click()
```

```
Dim curPrice As Currency
curPrice = CCur(txtItemPrice.Text)
lblVat.Caption = computeVAT(curPrice) 'calling a function
End Sub
```

Property Procedures

A property procedure returns and assigns values of properties on objects or modules. It is used to create read only properties for Forms, Standard modules and Class modules. Visual Basic provides three kind of property procedures-**Property Let** procedure that sets the value of a property, **Property Get** procedure that returns the value of a property, and **Property Set** procedure that sets the references to an object.

Arguments/Parameters

A value that is supplied to a procedure is called an argument. These values are declared inside the parenthesis after the procedure's name. When such a procedure is called, the calling procedure must also provide the values for these parameters. E.g.

```
Public Sub myTax(curX As Integer)
```

```
curResult = curX * 0.16
```

```
End Sub
```

When called,

```
Private Sub cmdTaxCalc_Click()
```

```
Dim curMySalary As Currency
```

```
curMySalary = CCur(Text1.Text)
```

```
Call frmDebugLogical.myTax(curMySalary)
```

```
Label1.Caption = curResult
```

```
End Sub
```

Parameters/Formal Arguments

These are the arguments/parameters in a function declaration. The scope of formal arguments is local to the procedure definition in which they are used and they are a copy of the actual arguments. A change in the formal arguments would not be reflected in the actual arguments.

When calling a procedure using the call keyword, you must use parenthesis around the parameter list. The data type of the actual arguments must match that of the formal arguments in the called procedure.

Actual Arguments

The arguments that are passed in a procedure call are called actual arguments. These arguments are defined in the calling procedure. Calling procedures pass information to called procedures in actual arguments. When a procedure is called, each formal argument is initialized with its corresponding actual argument from the calling procedure.

Variables

Variables are the memory locations which are used to store values temporarily during the execution of a program. A variable contains a value that is expected to change for example a person's age which will always change as days pass by i.e. variables are used to store values that are expected to change over the course of the program.

Variables can store data that originated from:

- a) An expression, e.g. `strNewMessage = "You have new mail"`
`intCalcValue = intTotal - intReadMail`
- b) Input from the user e.g. `strUserName = txtUserName.Text`
- c) A property value e.g. `strColour = frmMainForm.BackColor`
- d) An object e.g. `Set currentForm = frmMainForm`
- e) A value returned by a method or function e.g. `varDate = CDate(txtDate.Text)`

Declaring variables

Declaring a variable tells Visual Basic to reserve space in memory. It is not a must that a variable should be declared before using it. Automatically, whenever Visual Basic encounters a new variable, it assigns the default variable type (variant) and value (" ") i.e. "Empty". This is called **implicit declaration**. Though this type of declaration is easier for the user but it can lead to errors that may not be recognized at run time.

It is therefore advisable to declare all variables explicitly i.e. **Explicit Declaration**. To make sure that this rule is followed, start the Declarations with the Option Explicit clause. Variables are declared to let the program know about it in advance. It is declared with the Dim statement, followed by a unique name. The As type clause in the Dim statement allows one to define the data type or object type of the variable.

e.g. `Dim variablename [As type]`

Dim Total2 As Integer 'Explicit declaration

Dim Amount5, Amount6, Amount7 As Long 'Explicit declaration

Total = value1 + value2 'Implicit declaration

NB: With the second example, only the third variable, Amount7, is assigned the **Long** data type; the first two are variants i.e. the default data type. To correct the situation, one should explicitly declare every variable e.g.

Dim Amount5 As Long, Amount6 As Long, Amount7 As Long

A variable name:

- must begin with a letter
- cannot contain an embedded period e.g. a full stop.
- Must not exceed 255 characters
- Must be unique within the same scope.

NB: Explicit declaration of a variable is done in the Declarations Section of a module or at the beginning of a Procedure.

Variables have a lifetime i.e. the period of time during which they retain their value. Variables declared with **Dim** exist only while the procedure in which they are declared is executing. I.e. when a procedure is finished executing, the values of its local variables are not preserved and the memory used by the local variables is reclaimed. The next time the procedure is executed all its local variables are reinitialized.

Static variables – declared with **Static** keyword-, on the other hand, are not reinitialized each time a procedure is invoked and therefore retains or preserves value even when a procedure ends.

Scope of Variables

The scope of a variable, procedure or object determines which parts of the code in our application are aware of the variable's existence. There are local, module level and global variables.

Local Variables

A local variable is one that is declared inside a procedure. This variable is only available to the code inside the procedure and can be declared using the Dim statements as given below.

Dim intSum As Integer

The local variables exist as long as the procedure in which they are declared, is executing. Once a procedure is executed, the values of its local variables are lost and the memory used by these variables is freed and can be reclaimed. Variables that are declared with keyword Dim exist only as long as the procedure is being executed. A local variable can be declared with *Dim* and *Static* keywords.

Static variables are not reinitialized each time a procedure is invoked and therefore retain or preserves value even when a procedure ends.

e.g. Static intPermanent As Integer

Module Level Variables

A module level variable is available to all the procedures in the module and is declared in the Declarations section of the module. Such variables are visible only from within the module they belong to and can't be accessed from the outside. E.g.

Option Explicit

Private strHouseMaster As String, strMyName As String

Dim col1 As ColumnHeader

Global Variables

When a variable or constant is declared as **global** it is visible to all procedures and modules that comprise the application. Global variables and constants, as with Module level variables, must be declared outside of any procedures in the Declarations section of a standard module and must use the **Public** keyword. The syntax for declaring a global variable is as follows:

Public *variableName* **As** *dataType*

<i>Scope level</i>	<i>Example</i>
Local	Private Sub cmdFirst_Click()

	<pre> Dim counter As Integer Dim count As Integer count = 8 counter = count + 1 ' Increment the value of counter by 1 Print counter ' Print the value of the counter to the form End Sub </pre>
Module	<pre> Option Explicit Dim count As Integer Private Sub cmdFirst_Click() Dim counter As Integer count = 8 counter = count + 1 ' Increment the value of counter by 1 Print counter ' Print the value of the counter to the form Print count End Sub </pre>
Global	Modify the above program by adding a standard module (.BAS file) to your project and put Public count As Integer in the module. To add a standard module, click the Module icon on the toolbar, or choose Project, Add Module.

NB: A general rule: *always declare variables as locally as possible.*

ERROR HANDLING AND DEBUGGING

Error Handling enables programmers to write clearer, more robust, more fault-tolerant programs. Error handling enables the programmer to **attempt to recover (i.e., continue executing) from infrequent fatal errors** rather than letting them occur and suffering the consequences (such as loss of application data). If an error is severe and recovery is not possible, the program can be exited "gracefully"-all files can be closed and notification can be given that the program is terminating. The recovery code is called an error handler.

Error handling is designed for dealing with **run-time** errors such as an **attempt to divide by 0** (that occurs as the program executes the divide instruction). Other common examples of run-time errors are **memory exhaustion**, an **out-of-bound array index**, and **arithmetic overflow**. Error handling provides the programmer with a disciplined set of capabilities for dealing with these types of errors.

Error-handling code varies in nature and amount among software systems depending on the application and whether or not the software is a product for release. Products tend to contain much more error-handling code than is contained in "casual" software.

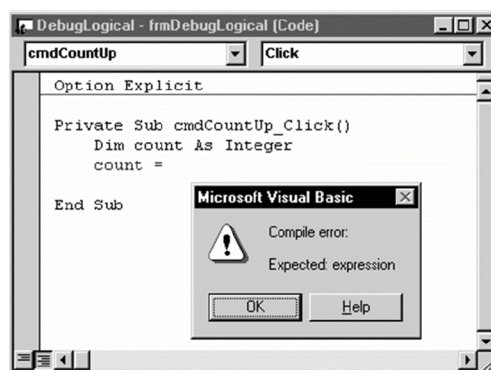
Usually, error-handling code is interspersed throughout a program's code. **Errors are dealt with in the places in the code where errors are likely to occur.** The advantage of this approach is that a programmer reading the code can see the error handling in the immediate vicinity of the code and determine if the proper error handling has been implemented.

Types of Programming Errors

There are three types of programming errors: *syntax errors*, *runtime errors* and *logical errors*. Each of the programming errors is described in the following sections.

1. Syntax Errors

Syntax errors are **generated when you stray from the rules of the programming language**. The **most common** bug is the syntax error, and it is also the **easiest to find**. Most syntax errors are **found by the Visual Basic** interpreter as soon as you type a line of code. Others show up as soon as you try to run your code.



Example of Syntax Error

They involve **misspelling keywords**, inserting the wrong number or type of arguments to a function. During design time, if you enter code with syntax errors, you are **prompted with a message informing you of the nature of the error**. **Colour coding**, provided by the Visual Basic editor, also helps to catch this type of error. Syntax errors can be prevented by choosing the Tools menu, Options and checking the *Auto Syntax Check* box on the Editor Tab.

2. Runtime Errors

The second type of error is the runtime errors. **These occur if code attempts to execute an instruction that is impossible to carry out**. For example, if you attempt to save a file to a drive that is not ready, a runtime error will be generated as shown in Fig. 8-2.

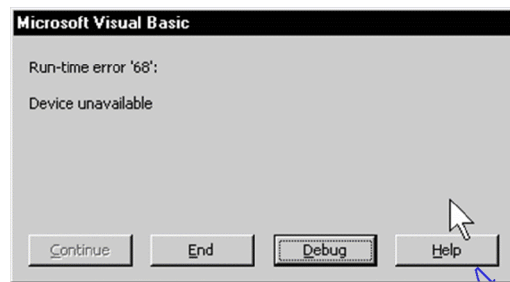


Fig. 8-2 Example of Runtime Error

Another common run-time error is misused block structures, such as a *For...Next* loop missing its *Next* statement. Again, this error **cannot be determined at design time**, but are **revealed when you attempt to run the code** and check whether blocks are incorrectly set up. For example,

```
Private Sub Command1_Click()  
    Dim x As Integer  
    For x = 0 To 10  
        MsgBox x  
    End Sub
```

Runtime errors most often represent a logic error in that you forgot to do something or wrote code that could not be executed. You can **trap these runtime errors** and handle them with a message to the user.

3. Logical/Semantic Errors

Logical errors occur when the computer is supplied with instructions that are logically incorrect i.e. the computer does not give the expected results. When you are lucky, the cause is obvious and you fix it. More often, **they are hard to find**, because code seems to execute correctly, but produces incorrect results. Visual Basic's debugging tools can assist you in tracking down errors of logic by letting you observe the code as it executes step by step.

Exercise: Finding a logical error

The following exercise illustrates a logical bug — see whether you can find it.

1. Start Microsoft Visual Basic from the **Start** menu and choose **Standard Exe**.

2. Draw a command button on the form and change its **caption** to *Count Up* and its name to *cmdCountUp*. Include a label control also.
3. Type in the following code in the code window.

```
Private Sub cmdCountUp_Click()
    Dim count As Integer
    count = count + 1
    Label1.Caption = count
End Sub
```

This procedure will be executed each time you click the command button. When it is executed, it increments the value of *count* by 1 and assigns the number in the *Caption* property of Label1; thus the Label shows the number each time you click at the command button.

4. Save the form as *frmDebugLogical.frm* and the project as *DebugLogical.vbp*.
5. Run the program and click the *Count Up* button.

What happens? The Label keeps showing 1 all the time. This is not what you told the computer to do — or is it? In the next few sections, you will learn a few methods of tracking down the problem.

Locating Logical Errors

There are a few steps involved in locating a logical error. They are:

1. Know the flow of your program. This involves gathering all the information you have about it and knowing what the program is doing.
2. Ask yourself these questions: "What is the error?" and "What causes it to occur?"
3. Using the knowledge you have for the above 2 steps, try to estimate the most probable cause of the error and then check that piece of code.
4. Try to narrow down the root of the problem to a few or even one procedure. You can make use of Visual Basic's debugging tools to help you to narrow down the root of the problem.
5. Keep gathering information and looking at the code until you find out what is happening.

Run-Time Error Trapping and Handling

Run-time errors are trappable. That is, Visual Basic recognizes an error has occurred and enables you to trap it and take corrective action. If an error occurs and is not trapped, your program will usually end in a rather unceremonious manner.

• Error trapping is enabled with the On Error GoTo statement:

Any time a run-time error occurs following this line, **program control is transferred** to the line labeled errlabel. On Error statement has three options:

1) On Error Goto label

This form of the On Error statement redirects program execution to the line label specified. When you use this form of On Error, a block of error handling code is constructed following the label.

2) **On Error Resume Next**

This form of the On Error statement tells VB to continue with the line of code following the line where the error occurred. With this type of error trap, you would normally test for an error at selected points in the program code where you anticipate that an error may occur.

3) **On Error Goto 0**

On Error Goto 0 disables any error handler within the current procedure. A run-time error that occurs when no error handler is enabled or after an On Error Goto 0 is encountered will be handled using VB's default error handling logic.

- The best way to explain how to use error trapping is to look at an outline of an example procedure with error trapping.

```
Sub SubExample()
```

```
[Declare variables, ...]
```

```
On Error GoTo HandleErrors
```

```
[Procedure code]
```

```
Exit Sub
```

```
HandleErrors:
```

```
Error handling code]
```

```
End Sub
```

Once you have set up the variable declarations, constant definitions, and any other procedure preliminaries, the **On Error statement is executed to enable error trapping**. Your normal procedure code follows this statement. **The error handling code goes at the end of the procedure**, following the HandleErrors statement label. This is the code that is executed if an error is encountered anywhere in the Sub procedure. Note you must exit (with Exit Sub) from the code before reaching the HandleErrors line to avoid inadvertent execution of the error handling code.

- Once a run-time error occurs, we would like to know what the error is and attempt to fix it. This is done in the error handling code.
- Visual Basic offers help in **identifying** run-time errors. The **Err object** returns, in its Number property (Err.Number), the number associated with the current error condition. The **Error() function** takes this error number as its argument and returns a string description of the error.
- Once an error has been trapped and some action taken, control must be returned to your application. That control is returned via the Resume statement. There are three options:
 1. **Resume** - Lets you retry the operation that caused the error. That is, control is returned to the line where the error occurred. This could be dangerous in that, if the error has not been corrected (via code or by the user), an infinite loop between the error handler and the procedure code may result.
 2. **Resume Next** - Program control is returned to the line immediately following the line where the error occurred.

3. **Resume label** - Program control is returned to the line labeled label. **A labeled line is simply a line with the label followed by a colon** e.g. Resume ExitLine

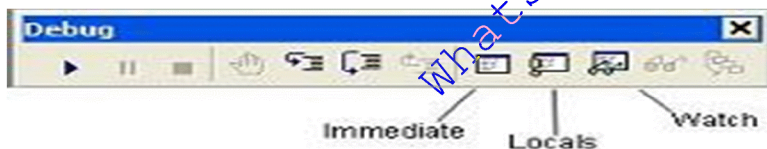
Error Number Error Description

6	Overflow
9	Subscript out of range
11	Division by zero
13	Type mismatch
16	Expression too complex
20	Resume without error
52	Bad file name or number
53	File not found
55	File already open
61	Disk full
70	Permission denied
92	For loop not initialized

Eliminating Logic Errors

These are errors that don't prevent an application from running, but cause incorrect or unexpected results. Visual Basic provides an excellent set of debugging tools to aid in this search.

The interface between your application and the debugging tools is via three different debug windows: the Immediate Window, the Locals Window, and the Watch Window. These windows can be accessed from the View menu (the Immediate Window can be accessed by pressing Ctrl+G). Or, they can be selected from the Debug Toolbar (accessed using the Toolbars option under the View menu):



- All debugging using the debug windows is done when your application is in **break mode**. You can enter break mode by setting breakpoints, pressing Ctrl+Break, or the program will go into break mode if it encounters an untrapped error or a Stop statement.
- Once in break mode, the debug windows and other tools can be used to:
 1. Determine values of variables
 2. Set breakpoints
 3. Set watch variables and expressions
 4. Manually control the application
 5. Determine which procedures have been called
 6. Change the values of variables and properties

Using the Debugging Toolbar

The Debug Toolbar



The Buttons on the Debug Toolbar as they follow one another:

Name	Description
Start	Runs the application
Break	Stops execution of a program temporarily
End	Stops running the program and returns to design mode
Toggle Breakpoint	Creates or removes a breakpoint.
Step Into	Runs the next executable line of code, stepping through each line of code that follows. If the code calls another procedure, Step Into steps through each line in that procedure also.
Step Over	Runs the next executable line of code, stepping through each line of code that follows. If the code calls another procedure, that procedure runs completely (rather than line by line) before stepping to the next line of code in the first/calling procedure.
Step Out	Executes the remainder of a called procedure and resumes at the next line of the calling procedure i.e. it is used to "step out" of a "called" procedure being stepped into.
Locals Window	Displays the current value of all declared variables
Immediate Window	It provides an immediate "view" into the memory by allowing you to execute code while the application is in Break Mode. You can monitor declared variables during Break Mode.
Watch Window	Displays the values of selected expressions in the code. With an expression, you can monitor an expression or variable as it changes or reaches a value that you define.
Quick Watch	Displays the value of the expression the cursor is on while in the Break Mode.
Call Stack	Displays the Call Stack dialog box which lists all the current procedures that have been called but not yet run to completion.

Setting Breakpoints

You can break a running code by either:

- Pressing Ctrl + Break , or
- Selecting **Break** from the **Run** menu, or
- Using the **Stop** statement in your code, or

- Clicking the **Toggle Breakpoint** button next to the **Start** button on the toolbar.
- Clicking on the margin indicator bar next to the line where you want to set the break point



The status of Break Mode is shown on the title bar

You can easily tell which mode you are in by looking at the title bar of Microsoft Visual Basic. When a program is in Break mode, you can examine code in the code window, print values in the Immediate window, change the values of variables, and even change code. You can change code without having to worry about the performance of the program.

Using the Immediate Window

You can go to the **Immediate Window** by pressing the Immediate Window button or press Ctrl + G. This window is the one you will work with frequently in debugging operations. **It provides an immediate "view" into the memory.**

The first step in debugging is to set a breakpoint in code. You can set as many breakpoints as you want. This is done by positioning the cursor at the line where you would like the break to begin and press F9 or click the **Toggle Breakpoint** button (the hand icon on the Debug toolbar). When the code is executed, Visual Basic will stop at the breakpoint so that you can step through your code. By **stepping into the code**, you will be able to narrow down to the root of the problem and thus solve the problem. Let's use the Break mode to find out what is wrong with the above program (*frmDebugLogical.frm*).

Exercise 8.2: Using the **Break Mode** to **Step Through** the program

1. Open the project called *DebugLogical.vbp* and hence *frmDebugLogical.frm*.
2. The problem lies in *cmdCountUp_Click* since it is the only procedure in the form, so open the code window, position the cursor at **Private Sub cmdCountUp_Click()** and press F9 or the Toggle Breakpoint button. The statement is highlighted in red colour, as shown in Fig. 8-6.

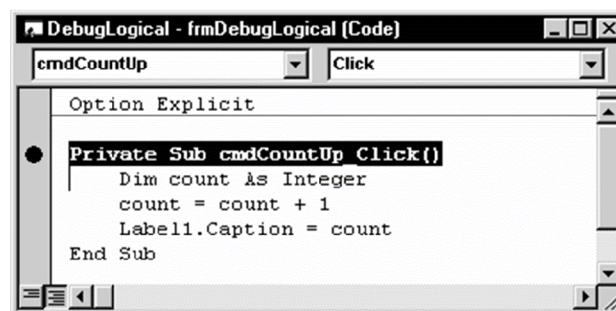


Fig. 8-6 Setting Breakpoint

3. Press F5 to run the program and click the *Count Up* button. The program stops at the breakpoint and displays the Code window. You will see that statement is highlighted in yellow colour with an arrow pointing at that line. It represents the code that will be executed next as you step through the procedure.
4. Go to the Immediate Window by pressing the Immediate Window button or press Ctrl + G. In the Immediate window, type **Print Me.Caption**. This will display the **Caption** property of the form. Next, type **Print cmdCountUp.Caption**. What do you see?
5. To step into the code, simply press F8 or click the **Step Into** button on the Debug toolbar. Press F8 now and you will notice the yellow highlighter goes down to the next line of code. Now move your mouse over the **count** variable. What do you see? It is **0 (zero)**, right? What is the reason?
6. Press F8 twice and move the mouse to the **count** variable and **Label1.Caption**. What do you see? Do you see the same as Fig. 8-7?

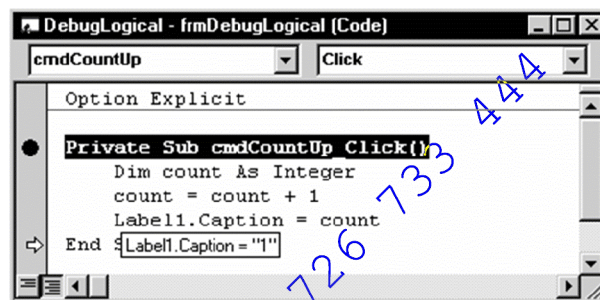


Fig. 8-7 Finding the Value of Label1.Caption

7. Press F8 once more to end the procedure and the program will now wait for you to click the button again.
8. Repeat Steps 3 to 7 a few times. Record down your observation.

Now stop the program by clicking the **Stop** button and change this statement **Dim count As Integer** to **Static count As Integer**. Repeat all the above steps again and record down your observation. You can discuss your observation together with the class.

Static variables retain their values even after the procedure has finished executing.

Clearing the Breakpoint(s)

A breakpoint remains in your code until you clear it. Clearing a breakpoint is the same as setting a breakpoint. You can clear it by either pressing F9 or the **Toggle Breakpoint** button again when the cursor is on the breakpoint. If you wish to clear one or more breakpoints in your code, choose **Clear All Breakpoints** from the **Debug** menu or press Ctrl + Shift + F9.

Using the Stop Statement

You can also break a running program by using the **Stop** statement in your code. This will have the same effect as setting the breakpoints. Modify the `cmdCountUp_Click` procedure as follows.

```
Private Sub cmdCountUp_Click()  
    Dim count As Integer  
    Stop  
    count = count + 1  
    Label1.Caption = count  
End Sub
```

Clear all breakpoints you have in the code and run the program. You will notice that the program will be in the Break mode once you click the *Count Up* button and the program will wait for you to step into the code in that procedure.

The Locals Window

Besides the Immediate window, there is another debugging window called **Locals** window. **The Locals window shows the values of all locally declared variables within the procedure being examined during the Break mode.** These values can be examined and even changed from the Locals window to stimulate the program to run differently from the normal. Changing the value of the variables will not affect the code at all. The Locals window can greatly accelerate diagnosing errors in your code. A quick scan of variable values often indicates the source of the problem.

You can get the Locals window by clicking the **Locals Window** button on the Debug toolbar.

Exercise 8.3: Using the Locals window

We will use the same program again (*DebugLogical.vbp*). Remove the **Stop** statement, if any, and set the breakpoint at `count = count + 1`.

1. Run the program until you hit the breakpoint. Click the **Locals Window** button on the Debug toolbar. You should see the window as shown in Fig. 8-8.

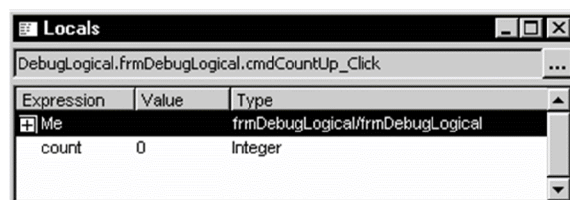


Fig. 8-8 The Locals window

The Locals window shows one variable called *count* and the implicit variable *Me*, which refers to the form module.

2. In the Locals window, select the value of *count*. You will see a cursor and the value is selected. Change its value to 10.
3. Step into the code by pressing **[F8]**. You should see that the value of *count* is now 11.
4. Stop the program and re-run the program. You should see that the value of *count* is back to 0 (zero).

Using the Watch Expression

Having watch expressions as part of debugging tools makes debugging easier and creates a more powerful method of solving the problem. **With a watch expression, you can monitor an expression or variable as it changes or reaches a value that you define.** You can create a watch expression at design time as well as in the Break mode. You can instruct Visual Basic to enter Break mode without setting a breakpoint or using the **Stop** statement by entering a certain criteria in the watch expression.

There are two types of watch expressions: a **regular** watch and a **quick** watch. **A regular watch allows you to watch the value of an expression**, which is similar to the Immediate window and the Locals window. A **quick watch** also enables you to check the value of the variables while in break mode; however, the watch is not retained in the Watches window.

You can add a watch expression by either choosing **Add Watch** from **Debug** menu or right clicking within the code window and select **Add Watch**.

Exercise 8.4: Using Regular and Quick Watch

1. Open the project called *DebugLogical.vbp* and hence *frmDebugLogical.frm*.
2. Add another command button and set its **Caption** property to "Using Watch" and its **Name** property to "cmdUsingWatch".
3. Type in the following code for the new command button.

```
Private Sub cmdUsingWatch_Click()  
    Dim x As Integer  
    For x = 0 To 10  
        Print x  
    Next x  
End Sub
```

4. Clear all breakpoints and set a new breakpoint at the line **For x = 0 To 10**.
5. Choose **Add Watch** from **Debug** menu. A dialog box will appear and change the dialog box so that it conforms to the one shown in Fig. 8-9. You will watch the *x* variable.

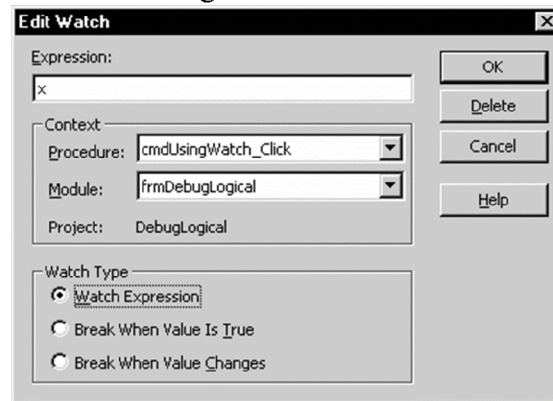


Fig. 8-9 The Add Watch dialog box

6. Click the **OK** button. The **Watches** window appears with the variable value that you intend to watch (see Fig. 8-10).

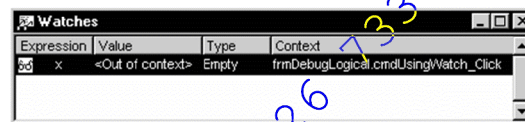


Fig. 8-10 The Watches Window

7. Run the program.
8. Click the new command button. When the program enters the Break mode, step into (press F8) the next line of code and notice the value of *x* in the Watches window. Its value is incremented with each loop iteration.
9. Stay within the **cmdUsingWatch_Click** procedure, and selects the variable *x*. Press the **Quick Watch** button on the Debug toolbar or press Shift + F. A dialog box appears and shows the value of *x*. You can have the choice of adding it to the Watches window or Cancel. Click the **Cancel** button because we have the *x* variable in the Watches window already.
10. Stop the program and proceed to the next exercise.

Exercise 8.5: Entering a Certain Criteria in the Watch Expression

1. Clear all breakpoints and remove all watch expressions by selecting them in the Watches window and pressing the *Delete* button.
2. Add the following watch expression.

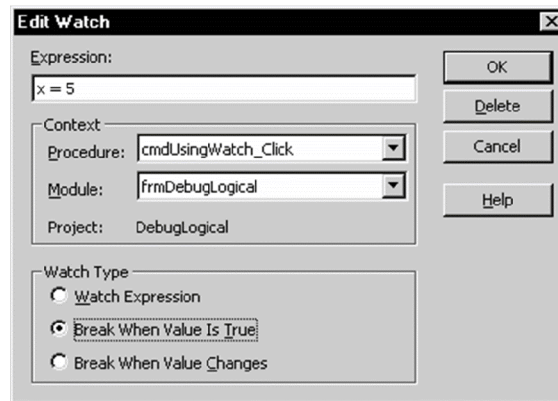


Fig. 8-11 Another Add Watch dialog box

3. Run the program and click the new command button, cmdUsingWatch.
4. The program will enter the Break mode once the value of *x* equals to 5. Take a look at the Watches window and you will see that the value is **True**. To get the precise value of *x*, point your mouse to *x*.
5. Stop the program.

One thing to remember about adding a watch expression, i.e. **you must select the right expression to evaluate.**

8.2.2 *The Debug Object*

Instead of breaking the flow of the program, i.e. going into the Break mode, you can make use of Visual Basic's **Debug** object and its **Print** method to **monitor the values of variables**. You just need to enter a **Debug.Print** request directly into your code and the results will be shown at the Immediate window for examination after the code has stopped executing. When you are ready to make an executable file, you can remove it from your code.

The following code shows the use of **Debug.Print** to show the result onto the Immediate window.

```
Private Sub cmdUsingWatch_Click()
    Dim x As Integer
    For x = 0 To 10
        Debug.Print x    ' Print the value of x to the Debug window
    Next x
End Sub
```

SEQUENTIAL FILES

In many applications, it is helpful to have the capability to read and write information to a disk file. This information could be some computed data or perhaps information loaded into a Visual Basic object.

Visual Basic provides the capability of processing three types of files:

1. **Sequential files** - Files that must be read in the same order in which they were written i.e. one after the other with no skipping
2. **Binary files** - "unstructured" files which are read from or written to as series of bytes, where it is up to the programmer to specify the format of the file
3. **Random files** - files which support "direct access" by record number

Sequential files can handle both text data and variable values. Sequential access is best when dealing with files that have lines with mixed information of different lengths. They are used to transfer data between applications.

Writing to Sequential Files

The following code snippet would allow one to write text to a file in VB 6.0. This can help if someone wants to log any kind of information to a log file.

```
Dim FileNo As Integer
Dim FileExists As String
Dim DirExists As String, strPathName As String
FileNo = FreeFile
'Check for the directory's existence
DirExists = Dir(strPathName, vbDirectory)
If DirExists = "" Then
MkDir (strPathName)
End If
```

```
FileExists = Dir(strPathName & "\" & strFileName, vbNormal)
```

```

'Check for the file existance
'If it's not there create it and write the line of text into it
If FileExists = "" Then
Open strPathName & "\" & strFileName For Output As FileNo
Print #FileNo, strLineToWrite
Close FileNo
Else
'If it is there then append data to end of file
Open strPathName & "\" & strFileName For Append As FileNo
Print #FileNo, strLineToWrite or intAge, strName, strCourse
Close FileNo
End If

```

NB/ Dir() function takes the pathname and Attributes as its arguments.

It returns the name of the file as its return value if the file is found and an empty string if not

Reading From Sequential Files

Sequential File-Related Statements and Functions

Open	Prepares a file to be processed by the VB program.
Path	Supplies the path of your application
FreeFile	Supplies a file number that is not already in use
Input #	Reads fields from a comma-delimited sequential file
Line Input #	Reads a line (up to the carriage return) from a sequential file
EOF	Tests for end-of-file
Write #	Writes fields to a sequential file in comma-delimited format
Print #	Writes a formatted line of output to a sequential file but, unlike write method, does not enclose string variables in quotes
Close #	Closes a file

As you know, a data file consists of records, which consist of fields.

Field	Data Type
Employee Name	String
Department Number	Integer
Job Title	String
Hire Date	Date
Hourly Rate	Single

Suppose there were five records in the file. A graphic representation of the file populated with the five data records follows (the field names are not stored in the file):

Employee Name	Dept #	Job Title	Hire Date	Hourly Rate
ANDY ANDERSON	100	PROGRAMMER	3/4/1997	25.00

BILLY BABCOCK	110	SYSTEMS ANALYST	2/16/1996	33.50
CHARLIE CHEESEMAN	100	COMPUTER OPERATOR	3/1/1996	15.00
DARLENE DUNCAN	200	RECEPTIONIST	10/11/1998	12.75
ERNIE EACHUS	300	MAIL ROOM CLERK	8/19/1997	10.00

Please note that the data types for these fields are the data types of the variables into which these fields will be stored. On the sequential file, all fields will be represented as a string of characters.

Comma-Delimited Format

Each field is separated by a comma. Both string and numeric fields are "trimmed" (contain no extraneous spaces or zeroes). String fields are enclosed in quotes (Note: The quotes enclosing the string fields are optional, VB and other applications that can read comma-delimited files will access the string fields properly with or without the quotes. The only time a string field MUST be enclosed in quotes is when it contains an embedded comma.) If Date fields are enclosed in pound signs (#), VB will automatically recognize the field as the Date data type. If the Date fields are enclosed in quotes instead, you need to use the **CDate** function to convert the date from string format to the Date data type.

"ANDY ANDERSON",100,"PROGRAMMER",#3/4/1997#,25
 "BILLY BABCOCK",110,"SYSTEMS ANALYST",#2/16/1996#,33.5
 "CHARLIE CHEESEMAN",100,"COMPUTER OPERATOR",#3/1/1996#,15
 "DARLENE DUNCAN",200,"RECEPTIONIST",#10/11/1998#,12.75
 "ERNIE EACHUS",300,"MAIL ROOM CLERK",#8/19/1997#,10

VB Statements and Functions for Sequential File Processing

1. The Open Statement

The Open statement prepares a file to be processed in the VB program. It identifies the Windows-system file that will be used in the program and assigns the file a file number that will be used to reference that file for the remainder of the program. The general format is:

Open <filename> [For mode] As [#] <filenumber>

- filename* is a legal Windows-system filename, which may include a drive and path; the filename can be specified in the Open statement as either a string constant or a string variable
- Mode* is one of the following three keywords: **Input**, **Output**, or **Append**.

When a file is opened for Input, that file must already exist.

When a file is opened for Output, if it does not exist, it will be created; if it does exist, its previous contents will be overwritten.

When a file is opened for Append, if it does not exist, it will be created, if it does exist, records will be added to the file after the last record in the file (the previous contents of the file will not be overwritten).

The **Input #** and **Line Input #** statements may only be used on files opened in the Input mode; the **Write #** and **Print #** may only be used on files open in the Output or Append modes.

- c. **filenumber** is an integer from 1 to 511 which is used to associate the Windows-system filename with a number; this number will be used to reference the opened file in all further VB file processing statements in the program.

Examples:

```
Open "C:\Program Files\EmpMaint\EMPLOYEE.DAT" For Input As #1
Open "A:\EMPLOYEE.DAT" For Input As #1
```

Using App.Path

In order to avoid "hard-coding" the path of a file in your VB program, it is recommended that you use App.Path to reference the path of the file. This way, as long as the file resides in the same directory in which your program is running, the correct path will always be referenced.

For example, if both your program and the data file reside in **C:\Program Files\EmpMaint**, then that is what App.Path would refer to. So if you concatenate App.Path with a backslash and the name of your data file, then you have a complete reference for your file, which can be used in the Open statement.

Examples:

```
Open App.Path & "\EMPLOYEE.DAT" For Input As #1
```

You could also use a string variable to hold the filename, as in the following example:

```
Dim strEmpFileName As String
```

```
strEmpFileName = App.Path & "\EMPLOYEE.DAT"
```

```
Open strEmpFileName For Input As #1
```

A special situation comes up if your program and the data file reside in the root directory of a drive (for example A:). If you concatenate App.Path with a backslash and the filename, you'll come up with an invalid file reference, such as:

```
A:\EMPLOYEE.DAT
```

To cover both situations and alleviate the pesky "extra backslash" problem, you can use code like the following (the new statements are shown in bold):

```
Dim strEmpFileName As String
```

```
Dim strBackSlash As String
```

```
strBackSlash = If (Right$(App.Path, 1) = "\", "", "\")
```

```
strEmpFileName = App.Path & strBackSlash & "EMPLOYEE.DAT"
```

```
Open strEmpFileName For Input As #1
```

Using FreeFile

Instead of hard-coding the file number, you can use the VB function FreeFile to supply you with a file number that is not already in use by the system. The FreeFile function takes no arguments and returns an integer. To use it, declare an integer variable, then assign FreeFile to it, as follows:

```
Dim intEmpFileNbr As Integer  
intEmpFileNbr = FreeFile
```

In the Open statement (and any other statement that refers to this file), use the integer variable rather than the hard-coded number.

2. The Input # Statement

The Input # statement reads a series of fields (usually one "record's worth") from a comma-delimited sequential file, and stores the contents of those fields into the specified variables. The general format is:

Input #<filename>, <variable list>

- a) **filename** refers to the file that was **Opened As** that number (for Input) in the Open statement
- b) **variable list** is a list of variables, separated by commas, into which the data fields from the file will be stored

Example:

Recall the comma-delimited version of the employee file shown earlier:

```
"ANDY ANDERSON",100, "PROGRAMMER",#3/4/1997#,25  
"BILLY BABCOCK",110, "SYSTEMS ANALYST",#2/16/1996#,33.5  
"CHARLIE CHEESEMAN",100, "COMPUTER OPERATOR",#3/1/1996#,15  
"DARLENE DUNCAN",200, "RECEPTIONIST",#10/11/1998#,12.75  
"ERNIE EACHUS",300, "MAIL ROOM CLERK",#8/19/1997#,10
```

Assume you declare the following variables in your program:

```
Dim strEmpName As String  
Dim intDeptNbr As Integer  
Dim strJobTitle As String  
Dim dtmHireDate As Date  
Dim sngHrlyRate As Single
```

the statement

Input #intEmpFileNbr, strEmpName, intDeptNbr, strJobTitle, dtmHireDate, sngHrlyRate

would cause **ANDY ANDERSON** to be stored in **strEmpName**, **100** to be stored in **intDeptNbr**, **PROGRAMMER** to be stored in **strJobTitle**, **3/4/1997** to be stored in **dtmHireDate**, and **25** to be stored in **sngHrlyRate** the first time that the statement was executed. The second time the statement was executed, **BILLY BABCOCK**, **110**, **SYSTEMS ANALYST**, **2/16/1996**, and **33.5** would be stored respectively in **strEmpName**, **intDeptNbr**, **strJobTitle**, **dtmHireDate**, **sngHrlyRate**; and so on. As VB reads each field into its respective variable, it automatically performs the conversion to the correct data type (Integer, Date, Single, etc.). As mentioned earlier, VB will only convert an incoming field to the Date data type if that field is enclosed in pound signs (#) – if the field was enclosed in quotes, it would be treated as a string and the CDate function would have to be used to convert it to a Date.

VB "knows" that the data is to be read from the "EMPLOYEE.DAT" file because the Input # statement is referring to file #intEmpFileNbr, and file #intEmpFileNbr was associated with "EMPLOYEE.DAT" in the Open statement.

3. The EOF function

The operating system automatically appends a special character, called the end-of-file marker, to the end of a sequential file. VB can sense the presence of this end-of-file marker with the EOF function.

A programming language will generally recognize EOF at either one of two times: (1) after the last record has been read – OR – (2) at the same time that the last record has been read. COBOL falls into the first category, VB falls into the second.

FYI: This discussion applies only to how VB processes sequential files – because when VB processes the rows of a database table, it actually handles EOF "the COBOL way".

In a language that recognizes EOF after the last record in the file has been read (such as COBOL), the "input" or "read" loop is set up similar like a prompted dialog loop: with a priming read outside the loop; all subsequent reads occur at the bottom of the loop. The pseudocode might be written as follows:

```
READ (first) RECORD
DO UNTIL EOF
PROCESS THE RECORD
READ (next) RECORD
LOOP
```

In a language that recognizes EOF when the last record is read (such as VB), the "input" or "read" loop must be modified so that there is NO PRIMING READ and the read occurs as the FIRST statement in the body of the processing loop. The pseudocode might be written as follows:

```
DO UNTIL EOF
READ A RECORD
PROCESS THE RECORD
LOOP
```

The syntax of the EOF function is **EOF(*n*)** where *n* is a number corresponding to the file number of the file from which you want to read data. *n* can either be a hard-coded number or an integer variable, depending on whether or not you used FreeFile in the Open statement.

The EOF function can be used anywhere that a conditional expression can be used; as such, it must always follow keywords such as UNTIL, WHILE, and IF. The EOF function can also be preceded by the keyword NOT: for example, **Do Until EOF(1)** is equivalent to **Do While Not EOF(1)**.

The main loop to process the employee file might look like this (note that there is no "priming" read and that the input is done at the top of the loop):

```
Do Until EOF(intEmpFileNbr)
Input #intEmpFileNbr, strEmpName, intDeptNbr, strJobTitle, dtmHireDate, sngHrlyRate

' Processing for the record would go here – for example, load some of these

' fields into an element of an array or list box, print a line of a report, etc...
```

Loop

Building on what has been discussed thus far, the "full-blown" procedure to process a comma-delimited sequential file for input might look like this:

```
Dim strEmpFileName As String
Dim strBackSlash As String
Dim intEmpFileNbr As Integer
```

```
Dim strEmpName As String
Dim intDeptNbr As Integer
Dim strJobTitle As String
Dim dtmHireDate As Date
Dim sngHrlyRate As Single
strBackSlash = IIf (Right$(App.Path, 1) = "\", "", "\")
strEmpFileName = App.Path & strBackSlash & "EMPLOYEE.DAT"
intEmpFileNbr = FreeFile
```

```
Open strEmpFileName For Input As #intEmpFileNbr
```

```
Do Until EOF(intEmpFileNbr)
Input #intEmpFileNbr, strEmpName, intDeptNbr, strJobTitle, dtmHireDate, sngHrlyRate

Print strEmpName; _

Tab(25); Format$(intDeptNbr, "@@@@"); _

Tab(35); strJobTitle; _

Tab(55); Format$(dtmHireDate, "mm/dd/yyyy"); _

Tab(70); Format$(Format$(sngHrlyRate, "Standard"), "@@@@@@")
```

Loop

```
Close #intEmpFileNbr
```

4. The Close Statement

When you are finished using a file in your program, you should Close that file. The Close statement tells VB that you are done using a file, and frees up the system resources needed to process that file.

The statement

Close #1

frees the resources used by the file referenced as #1, and also terminates the association between the Windows-system file and the file number “ so at this point, if you wanted to, you could Open some other file AS #1.

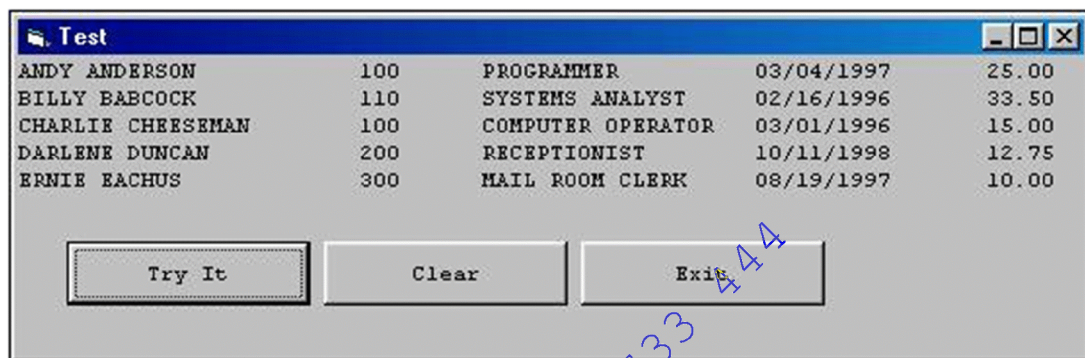
If you have more than one file open in a program, you can close multiple files with one Close statement by separating the file numbers with commas:

Close #1, #2, #68

The statement

Close

by itself on one line closes all files that are currently open.



LINKING TO DATABASES

All business applications need to store large volumes of data organized in a format so that information can be retrieved efficiently and quickly as and when required. With the help of a DBMS (Database Management System), managing the data becomes easy. A DBMS is a system that manages the storage and retrieval of data in a database.

Further Microsoft Visual Basic provides tools for creating and accessing a variety of RDBMS (Relational Database Management System). An RDBMS stores and retrieves information according to the relationship defined. In a RDBMS, the data is the container of the tables in which all data is stored in the relationships is formed by data values.

A database is a collection of data that is related one to another to support a common application. For example Employee details - Name, Address, etc. Each of these collections of data continue a database.

database accessing methods are as follows;

1. Jet Engine - Accessing Microsoft Access and Visual Basic databases.
2. ODBC (Open Database Connectivity) - Allow access to the client server databases on a network.
3. ISAM (Index Sequential Access Method) - Used to access flat databases such as dBase, FoxPro, ParaDox.

ADO (ActiveX Data Object) data control



• The ADO (ActiveX Data Object) data control is the primary interface between a Visual Basic application and a database. It can be used without writing any code at all! Or, it can be a central part of a complex database management system. This icon may not appear in your Visual Basic toolbox. If it doesn't, select Project from the main menu, then click Components. The Components window will appear. Select Microsoft ADO Data Control, then click OK. The control will be added to your toolbox.

• As mentioned in Review and Preview, previous versions of Visual Basic used another data control. That control is still included with Visual Basic 6.0 (for backward compatibility) and has as its icon:



Make sure you are not using this data control for the work in this class. This control is suitable for small databases. You might like to study it on your own.

• The data control (or tool) can access databases created by several other programs besides Visual Basic (or Microsoft Access). Some other formats supported include Btrieve, dBase, FoxPro, and Paradox databases.

• The data control can be used to perform the following tasks:

1. Connect to a database.
2. Open a specified database table.
3. Create a virtual table based on a database query.
4. Pass database fields to other Visual Basic tools, for display or editing. Such tools are **bound tools** (controls), or **data aware**.
5. Add new records or update a database.
6. Trap any errors that may occur while accessing data.
7. Close the database.

DATA CONTROL PROPERTIES:

Align Determines where data control is displayed.

Caption Phrase displayed on the data control.

ConnectionString Contains the information used to establish a connection to a database. E.g. Provider = Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files (x86)\Microsoft Visual Studio\VB98\BIBLIO.MDB;Persist Security Info=False

LockType Indicates the type of locks placed on records during editing (default setting makes databases read-only).

Recordset A set of records defined by a data control's ConnectionString and RecordSource properties. Run-time only.

RecordSource Determines the table (or virtual table) the data control is attached to.

• **As a rule, you need one data control for every database table**, or virtual table, you need access to. One row of a table is accessible to each data control at any one time. This is referred to as the current record.

• When a data control is placed on a form, it appears with the assigned caption and four arrow buttons:



The arrows are used to navigate through the table rows (records). As indicated, the buttons can be used to move to the beginning of the table, the end of the table, or from record to record.

Assigning Tables

- Once the ADO data control is connected to a database, we need to assign a table to that control. Recall each data control is attached to a single table, whether it is a table inherent to the database or the virtual table we discussed. Assigning a table is done via the **RecordSource** property.

- Tables are assigned by making queries of the database. The language used to make a query is **SQL** (pronounced 'sequel,' meaning structured query language). SQL is an English-like language that has evolved into the most widely used database query language. You use SQL to formulate a question to ask of the database. The data base 'answers' that question with a new table of records and fields that match your criteria.

- A table is assigned by placing a valid SQL statement in the **RecordSource** property of a data control. We won't be learning any SQL here. There are many texts on the subject - in fact, many of them are in the BIBLIO.MDB database we've been using. Here we simply show you how to use SQL to have the data control 'point' to an inherent database table.

- Click on the ellipsis next to **RecordSource** in the **property** box. A Property Pages dialog box will appear. In the box marked **Command Text (SQL)**, type this line:

SELECT * FROM TableName

This will select all fields (the * is a wildcard) from a table named **TableName** in the database. Click **OK**.

- Setting the **RecordSource** property also establishes the **Recordset** property, which we will see later is a very important property.

Data Bound Tools

- Most of the Visual Basic tools we've studied can be used as bound, or data-aware, tools (or controls). That means, certain tool properties can be tied to a particular database field. To use a bound control, one or more data controls must be on the form.

- Some bound data tools are:

Label - Can be used to provide display-only access to a specified text data field.

Text Box - Can be used to provide read/write access to a specified text data field. Probably, the most widely used data bound tool.

Check Box - Used to provide read/write access to a Boolean field.

Combo Box - Can be used to provide read/write access to a text data field.

List Box - Can be used to provide read/write access to a text data field.

Picture Box - Used to display a graphical image from a bitmap, icon, or metafile on your form. Provides read/write access to a image/binary data field.

Image Box - Used to display a graphical image from a bitmap, icon, or metafile on your form (uses fewer resources than a picture box). Provides read/write access to a image/binary data field.

- There are also three 'custom' data aware tools, the **DataCombo** (better than using the bound combo box), **DataList** (better than the bound list box), and **DataGrid** tools, we will look at later.

Bound Tool Properties:

DataChanged - Indicates whether a value displayed in a bound control has changed.

DataField - Specifies the name of a field in the table pointed to by the respective data control.

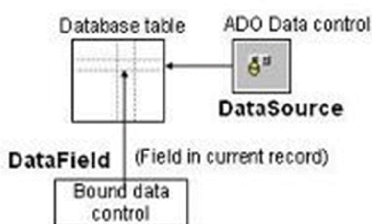
DataSource - Specifies which data control the control is bound to.

If the data in a data-aware control is changed and then the user changes focus to another control or tool, the database will automatically be updated with the new data (assuming LockType is set to allow an update).

- To make using bound controls easy, follow these steps (in order listed) in placing the controls on a form:
 1. Draw the bound control on the same form as the data control to which it will be bound.
 2. Set the **DataSource** property. Click on the drop-down arrow to list the data controls on your form. Choose one.
 3. Set the **DataField** property. Click on the drop-down arrow to list the fields associated with the selected data control records. Make your choice.
 4. Set all other properties, as required.

By following these steps in order, we avoid potential data access errors.

- The relationships between the bound data control and the data control are:



ADO CURSORS AND LOCKS

A database **cursor** is a **control structure** that enables **traversal** over the records in a database. Cursors facilitate subsequent processing in conjunction with the traversal, such as retrieval, addition and removal of database records. The database cursor characteristic of traversal makes cursors akin to the programming language concept of iterator.

Cursors are used by database programmers to process individual rows returned by database system queries. Cursors enable manipulation of whole result sets at once. In this scenario, a cursor enables the rows in a result set to be processed sequentially.

In SQL procedures, a cursor makes it possible to define a result set (a set of data rows) and perform complex logic on a row by row basis. By using the same mechanics, a SQL procedure can also define a result set and return it directly to the caller of the SQL procedure or to a client application.

CURSOR TYPE CONSTANTS AND DESCRIPTIONS

In ADO, when we talk about cursors, we are essentially talking about a set of rows. When you execute a query that returns rows of data, such as `SELECT * FROM mytable`, the resulting data is handled using a cursor. A cursor can be located either on the client with the `adUseClient` argument, or on the server with the `adUseServer` argument. In addition, there are 4 types of cursor: `adOpenForwardOnly`, `adOpenStatic`, `adOpenDynamic`, and `adOpenKeyset`.

The different types and locations will be discussed in further detail below. Your choice of cursor type and cursor location will affect what you are able to do with the data you retrieve, and how changes made to the data by other users are reflected in your copy of the data.

Cursor Location

The `ADODB.Connection` object (the ADO object used to broker all data exchanges between the VB application and the DATABASE) has a property known as `CursorLocation` which is used to set/retrieve the cursor location that will be used by any recordset objects that access their data through the connection object.

The `CursorLocation` property can only be set while the connection is closed, and the property will be inherited by any recordset objects that access their data through the given connection object. Recordset objects can also explicitly set a `cursorlocation` different than the connection objects `cursorlocation` as long as it is set before the recordset is open. The two options available for this property are `adUseClient` and `adUseServer`, with `adUseServer` being the default property.

adUseServer

When using the `adUseServer` server-side `cursorlocation`, responsibility for handling the data generated by a query lies with the database server. **The benefit of server-side cursors is that we gain access to the dynamic cursor type. This allows us to see any changes to the data that are made by other users in the data our application is accessing.**

For example: let's say we are selling tickets to a concert with our application, we need to know that a given seat is available for sale in real-time to ensure we do not double-book the seat. With a server-side cursor, **we can be sure that the data we are manipulating is the most current possible. In addition, we have the ability to lock the data we are working on as we edit it, to make sure our changes are going to be posted to the database successfully.**

With a server-side cursor (`adUseServer`), we have access to the `adOpenDynamic` and `adOpenForwardOnly` cursor types, and all four of the recordset lock types, which will be discussed below.

It should be noted that using a server-side cursor, and the **adOpenDynamic** cursor in particular, **will result in a significant performance loss**, and should be avoided if at all possible. In addition, **certain functionality, such as the **RecordCount** property of a Recordset will fail or return abnormal results when used with a server-side cursor.**

adUseClient

Client-side cursors, specified with the **adUseClient** keyword, are handled internally by ADO. **These cursors offer more functionality than their server-side counterparts**, and also **result in less load being placed on the server**. Most advanced ADO functionality is designed for use with client-side cursors.

When using a client-side **adUseClient** cursor, only the **adOpenStatic** cursor is available, and we cannot use the **adLockPessimistic** lock type (see below).

Client-side cursors also help decrease load on the server, since with a static cursor data is sent to the client and then the server has no further communications with the client. This allows your server to scale a lot better than with server-side cursors.

Cursor Types

In addition to the two cursor locations, there are four cursor types, three of which are supported under Connector/ODBC:

- **adOpenStatic** (Client-Side)
- **adOpenForwardOnly** (Server-Side)
- **adOpenDynamic** (Server-Side)

The different cursor types support different functionality and features.. The fourth cursor type, **adOpenKeySet**, is not currently supported by MySQL / MyODBC.

adOpenStatic

The static cursor is the only cursor type that is currently available when using **adUseClient** as your cursor location. **With a static cursor, the server will send the result set to the client, after which there will be no further communication from the server to the client.** The client may communicate with the server to send changes back to the server. This makes the static cursor more resource-intensive for the client and less resource-intensive for the server, as the result set is stored in the client's memory instead of the server's.

If a different client makes changes to the underlying data after the query results are sent, the original client will receive no notification of the change. A static cursor is **bi-directional**, meaning that your application can move forwards and backwards through the recordset. The following methods are available to a recordset using a static cursor and the **adLockOptimistic** lock type (more on lock types later):

- AddNew
- Delete
- Find
- MoveNext
- MoveLast
- Resync

- MoveFirst
- MovePrevious
- Update
- UpdateBatch

adOpenForwardOnly

The **adForwardOnly** cursor type is the fastest performing cursortype, and also the most limited. The forward-only cursor does not support the RecordCount property, and does not support the **MovePrevious** methods of the recordset object.

The most efficient way to access data for display to the screen output to a file is to use a **adOpenForwardOnly** cursor with a **adLockReadOnly** lock type when opening a recordset. This combination is often referred to as a Firehose Cursor. A firehose cursor bypasses a lot of handling code between the client and server and allows for very fast data access when moving sequentially through the resulting rows.

The following recordset methods are supported when using a forward-only cursor with an optimistic lock:

- AddNew
- Delete
- Find
- Update
- UpdateBatch

With every call to the recordset's **MoveNext** method, the previous row is discarded and the next row is queried from the server.

adOpenDynamic

While the forward-only cursor is the most efficient of the cursor types, the dynamic cursor, specified AS **adOpenDynamic**, is the least efficient.

As there is no native support for dynamic, server-side cursors in MySQL, every call to a row-moving method(**MoveNext**, **MovePrevious**, etc.) results in the Connector/ODBC driver converting your method call to a SQL query, posting the query, and returning the resulting row.

The dynamic cursor supports the following recordset methods when opened with a optimistic lock:

- AddNew
- Delete
- Find
- MoveFirst
- MovePrevious
- Update
- UpdateBatch

Lock Types

While cursor locations and cursor types specify how data is going to be handled, the **lock type property specifies how we are going to lock the underlying data to protect any changes we make and ensure they are processed**. There are four different lock types, and the locktype is set in the recordset object as part of the open method (it can also be set using the LockType property of the recordset object). The four locktypes are: **adLockReadOnly** (default), **adLockOptimistic**, **adLockPessimistic**, and **adLockBatchOptimistic**. All four locktypes are available to a server-side cursor, the **adLockPessimistic** locktype is unavailable to a client-side cursor.

adLockReadOnly

The default lock type is **adLockReadOnly**. A read-only lock is the most efficient when accessing data, as there is no checking for data changes and therefore no extra traffic between the client and server while loading records.

As the name implies, using a read-only lock will block you from making any changes to the table.

adLockOptimistic

An optimistic lock is used for modifications that either happen in a low-concurrency environment, or where having multiple users making changes to the same records is not a major concern. With an optimistic lock, the table or row locks will occur when the update method of the recordset object is called. This will ensure the change is successfully made, but will not prevent other users from changing the underlying data while you are modifying it in VB.

adLockBatchOptimistic

When using the **adBatchOptimistic** lock type, your changes will be cached locally until the recordset's **UpdateBatch** method is called. When **UpdateBatch** is called, all changes will be pushed to the server in a group. This can make the bulk insert of a large number of records more efficient.

adLockPessimistic

In a situation of high concurrency, with multiple users modifying the same data, you may need a pessimistic lock type. With **adLockPessimistic**, the underlying rows (or table) will be locked as soon as you begin making changes to the current record, and will not be unlocked until the Update method is called.

While this will ensure that you do not have overlapping changed with other users, it could cause performance issues.

Data Manager in Visual Basic 6 (VB6)

The Visual Data Manager is a Visual Basic Add-In that **allows the creation and management of databases**. It is simple to use and can create a database compatible with the Microsoft Jet (or Access) database engine.

- To examine an existing database using the Data Manager, follow these steps:

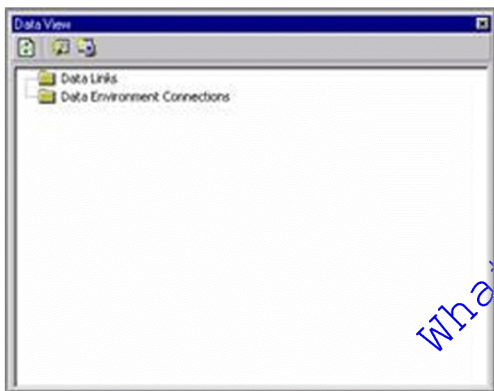
1. Select Visual Data Manager from Visual Basic's Add-In menu (you may be asked if you want to add SYSTEM.MDA to the .INI file - answer No.)
2. Select Open Database from the Data Manager File menu.
3. Select the database type and name you want to examine.

Once the database is opened, you can do many things. You can simply look through the various tables. You can search for particular records. You can apply SQL queries. You can add/delete records. The Data Manager is a DBMS in itself. You might try using the Data Manager to look through the BIBLIO.MDB example database.

• To create a new database, follow these steps:

1. Select **Visual Data Manager** from Visual Basic's **Add-In** menu.
2. Select **New** from the Data Manager **File** menu. Choose database type (Microsoft Access, Version 7.0), then select a directory and enter a name for your database file. Click **OK**.
3. The Database window will open. Right click the window and select **New Table**. In the **Name** box, enter the name of your table. Then define the table's fields, one at a time, by clicking **Add Field**, then entering a field name, selecting a data type, and specifying the size of the field, if required. Once the field is defined, click the **OK** button to add it to the field box. Once all fields are defined, click the **Build the Table** button to save your table.

Data view Window



Data View enables developers to examine all databases used in a project, and then quickly and easily integrate database objects into an application. Developers can use Data View to view table structures, database views, stored procedures, and other database objects directly within the development environment, as well as to view and change the actual data in a convenient grid pane.

The Data View component provides a visual interface in which you can view:

1. Live connections to the databases in each project
2. Objects available in each database, organized into folders, including:
 - Database diagrams
 - Tables
 - Triggers

- Views
- Stored procedures

Example - Accessing the Books Database

1. Start a new application. We'll develop a form where we can skim through the books database, examining titles and ISBN values. Place an ADO data control, two label boxes, and two text boxes on the form.
2. If you haven't done so, create a data link for the BIBLIO.MDB database following the steps given under Data Links in these notes.
3. Set the following properties for each control. For the data control and the two text boxes, make sure you set the properties in the order given.

Form1:

BorderStyle - 1-Fixed Single
Caption - Books Database
Name - frmBooks

Adodc1:

Caption - Book Titles
ConnectionString - BIBLIO.UDL (in whatever folder you saved it in - select, don't type)
RecordSource - SELECT * FROM Titles
Name - dtaTitles

Label1:

Caption - Title

Label2:

Caption - ISBN

Text1:

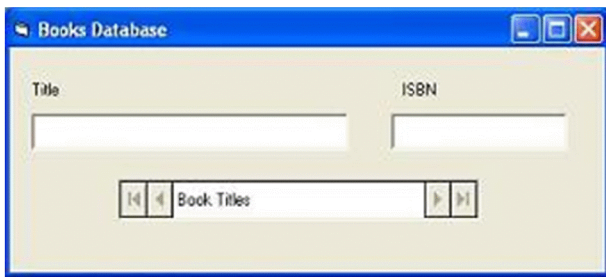
DataSource - dtaTitles (select, don't type)
DataField - Title (select, don't type)
Locked - True
MultiLine - True
Name - txtTitle
Text - [Blank]

Text2:

DataSource - dtaTitles (select, don't type)
DataField - ISBN (select, don't type)
Locked - True
Name - txtISBN
Text - [Blank]

Whatsapp 0726 733 444

When done, the form will look something like this (try to space your controls as shown; we'll use all the blank space as we continue with this example):



4. Save the application. Run the application. Cycle through the various book titles using the data control. Did you notice something? You didn't have to write one line of Visual Basic code! This indicates the power behind the data tool and bound tools.

Finding Specific Records

- In addition to using the data control to move through database records, we can write Visual Basic code to accomplish the same, and other, tasks. This is referred to as **programmatic control**. In fact, many times the data control **Visible** property is set to **False** and all data manipulations are performed in code. We can also use programmatic control to find certain records.

- There are four methods used for moving in a database. These methods replicate the capabilities of the four arrow buttons on the data control:

MoveFirst - Move to the first record in the table.

MoveLast - Move to the last record in the table.

MoveNext - Move to the next record (with respect to the current record) in the table.

MovePrevious - Move to the previous record (with respect to the current record) in the table.

- When moving about the database programmatically, we need to test the **BOF (beginning of file)** and **EOF (end of file)** properties. The BOF property is True when the current record is positioned before any data. The EOF property is True when the current record has been positioned past the end of the data. If either property is True, the current record is invalid. If both properties are True, then there is no data in the database table at all.

- These properties, and the programmatic control methods, operate on the **Recordset** property of the data control. Hence, to move to the first record in a table attached to a data control named **dtaExample**, the syntax is:

dtaExample.Recordset.MoveFirst

e.g. Adodc1.Recordset.MoveFirst

- There is a method used for searching a database:

Find - Find a record that meets the specified search criteria.

This method also operates on the Recordset property and has three arguments we will be concerned with. To use Find with a data control named dtaExample:

dtaExample.Recordset.Find Criteria,NumberSkipped,SearchDirection

as in

```
dtaTitles.Recordset.Find "Author >= '" + cmdLetter(Index).Caption + "'", 0,
```

- The search **Criteria** is a string expression like a **WHERE** clause in SQL. Simply put, the criteria describes what particular records it wants to look at.

Criteria = "Title >= 'S'"

Note the use of single quotes around the search letter. Single quotes are used to enclose strings in Criteria statements. Three logical operators can be used: equals (=), greater than (>), and less than (<).

- The **NumberSkipped** argument tells how many records to skip before beginning the Find. This can be used to exclude the current record by setting NumberSkipped to 1.
- The **SearchDirection** argument has two possible values: **adSearchForward** or **adSearchBackward**. Note, in conjunction with the four Move methods, the **SearchDirection** argument can be used to provide a variety of search types (search from the top, search from the bottom, etc.)
- If a search fails to find a record that matches the criteria, the Recordset's **EOF** or **BOF** property is set to **True** (depending on search direction). Another property used in searches is the **Bookmark** property. This allows you to save the current record pointer in case you want to return to that position later.

Sample Search Code

```
Private Sub cmdLetter_Click(Index As Integer)
```

```
Dim BookMark1 As Variant 'Mark your place in case no match is found
```

```
BookMark1 = dtaTitles.Recordset.Bookmark 'Move to top of table to start search
```

```
dtaTitles.Recordset.MoveFirst
```

```
dtaTitles.Recordset.Find "Author >= '" + cmdLetter(Index).Caption + "'", 0, adSearchForward
```

```
If dtaTitles.Recordset.EOF = True Then
```

```
    dtaTitles.Recordset.Bookmark = BookMark1
```

```
End If
```

```
txtAuthor.SetFocus
```

```
End Sub
```

Creating a Data Report in Visual Basic 6 (VB6)

- Data reports are a good way for obtaining printed or displayed information from your data. The process of obtaining such information is known as creating a data report.
- There are two steps to creating a data report. First, we need to create a Data Environment. This is designed within Visual Basic and is **used to tell the data report what is in the database**. Second, we create the Data Report itself. This, too, is done within Visual Basic. The Data Environment and Data Report files then become part of the Visual Basic project developed as a database management system.
- The Visual Basic 6.0 data report capabilities are vast and using them is a detailed process. The use of these capabilities is best demonstrated by example. We will look at the rudiments of report creation by building a tabular report for our phone database.

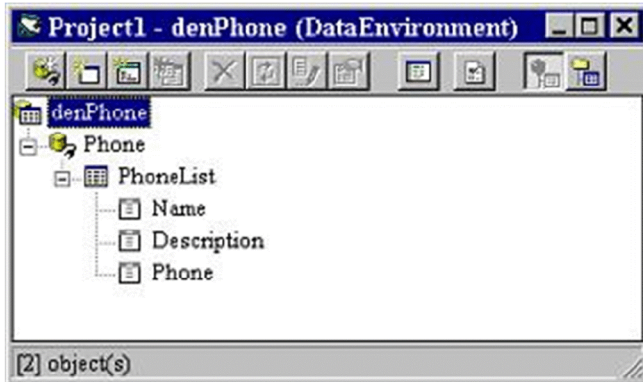
Example - Phone Directory - Building a Data Report

We will build a data report that lists all the names and phone numbers in our phone database. We will do this by first creating a Data Environment, then a Data Report. We will then reopen the phone database management project and add data reporting capabilities.

Creating a Data Environment

1. Start a new **Standard EXE** project.
2. On the Project menu, click **Add Data Environment**. If this item is not on the menu, click **Components**. Click the **Designers** tab, and choose **Data Environment** and click **OK** to add the designer to your menu.
3. We need to point to our database. In the **Data Environment** window, right-click the **Connection1** tab and select **Properties**. In the **Data Link Properties** dialog box, choose **Microsoft Jet 3.51 OLE DB Provider**. Click **Next** to get to the **Connection** tab. Click the **ellipsis** button. Find your phone database (mdb) file. Click **OK** to close the dialog box.
4. We now tell the **Data Environment** what is in our database. Right-click the **Connection1** tab and click **Rename**. Change the name of the tab to Phone. Right-click this newly named tab and click **Add Command** to create a **Command1** tab. Right-click this tab and choose **Properties**. Assign the following properties:

 Command Name - PhoneList
 Connection - Phone
 DataBase Object - Table
 ObjectName - PhoneList
5. Click **OK**. All this was needed just to connect the environment to our database.
6. Display the properties window and give the data environment a name property of denPhone. Click **File** and **Save** denPhone As. Save the environment in an appropriate folder. We will eventually add this file to our phone database management system. At this point, my data environment window looks like this (I expanded the PhoneList tab by clicking the + sign):



Creating a Data Report

Once the Data Environment has been created, we can create a Data Report. We will drag things out of the Data Environment onto a form created for the Data Report, so make sure your Data Environment window is still available.

1. On the Project menu, click Add Data Report and one will be added to your project. If this item is not on the menu, click Components. Click the Designers tab, and choose Data Report and click OK to add the designer to your menu.

2. Set the following properties for the report:

Name - rptPhone

Caption - Phone Directory

DataSource - denPhone (your phone data environment - choose, don't type)

DataMember - PhoneList (the table name - choose don't type)

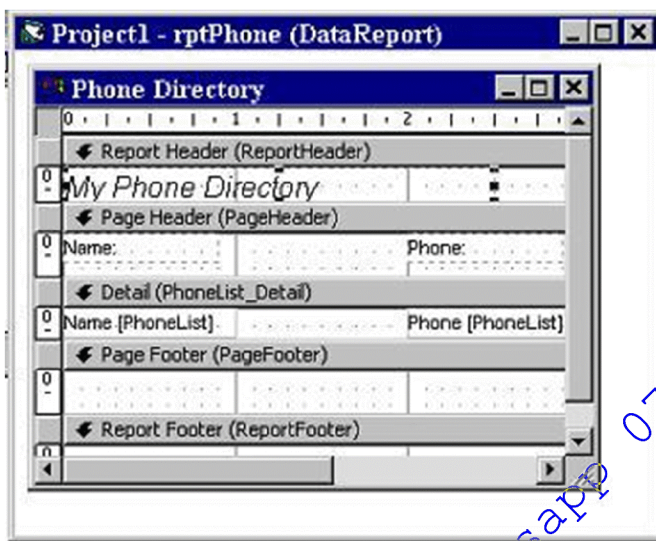
3. Right-click the **Data Report** and click **Retrieve Structure**. This establishes a report format based on the **Data Environment**.

Notice that a Data Report is divided into several sections, much like reports in Microsoft Access. These sections display the following types of information:

- a) **Report Header.** Information displayed once, at the top of the report
- b) **Page Header.** Information displayed at the top of every page
- c) **Detail Section.** The part of the report that is repeated for each record in the ADO data source
- d) **Page Footer.** Information displayed at the bottom of every page
- e) **Report Footer.** Information displayed at the very end of the report

To place information in one of these regions, right-click the selected region, click Add Control, then choose the control you wish to place. These controls are called data report controls and properties are established just like you do for usual controls. Try adding some headers.

- i. The **Detail section** is used to lay out the information you want printed for each record in your database. We will place two field listings (Name, Phone) there. Click on the Name tab in the Data Environment window and drag it to the Detail section of the Data Report. Two items should appear: a text box Name and a text box Name (PhoneList). The first text box is heading information. Move this text box into the Page Header section. The second text box is the actual value for Name from the PhoneList table. Line this text box up under the Name header. Now, drag the Phone tab from the Data Environment to the Data Report. Adjust the text boxes in the same manner. Our data report will have page headers Name and Phone. Under these headers, these fields for each record in our database will be displayed. When done, the form should look something like this:



In this form, I've resized the labels a bit and added a Report Header. Also, make sure you close up the Detail section to a single line. Any space left in this section will be inserted after each entry.

6. Click File and Save rptPhone As. Save the environment in an appropriate folder. We will now reopen our phone database manager and attach this and the data environment to that project and add capabilities to display the report.

Accessing the Data Report

1. Reopen the phone directory project. Add a command button named cmdReport and give it a Caption of Show Report. (There may be two tabs in your toolbox, one named General and one named DataReport. Make sure you select from the General tools.)
2. We will now add the data environment and data report files to the project. Click the Project menu item, then click Add File. Choose denPhone and click OK. Also add rptPhone. Look at your Project Window. Those files should be listed under Designers.
3. Use this code in cmdReport_Click:

```
Private Sub cmdReport_Click()  
rptPhone.Show  
End Sub
```

4. This uses the Show method to display the data report.

5. Save the application and run it. Click the Show Report button and this should appear:



You now have a printable copy of the phone directory. Just click the Printer icon. Notice the relationship with this displayed report and the sections available in the Data Report designer.

STRING FUNCTIONS

Function:	Len
Description:	Returns a Long containing the length of the specified string
Syntax:	Len(<i>string</i>) Where <i>string</i> is the string whose length (number of characters) is to be returned.
Example:	lngLen = Len("Visual Basic") ' lngLen = 12

Function:	Left() & Right()
	Returns a substring containing a specified number of characters from a string starting from the left or the Right.
Syntax:	Left(string,length) or Right(string,length) Str1stChar= Left(txt2ndName,1)
Function:	Mid (or Mid)
Description:	Returns a substring containing a specified number of characters from any part of a string.
Syntax:	Mid(string, start, length) The Mid\$ function syntax has these parts: string Required. String expression from which characters are returned. start Required; Long. Character position in string at which the part to be taken begins. If start is greater than the number of characters in string, Mid returns a zero-length string (""). length Optional; Long. Number of characters to return. If omitted or if there are fewer than length characters in the text (including the character at start), all characters from the start position to the end of the string are returned.
Example:	strSubstr = Mid\$("Visual Basic", 3, 4) ' strSubstr = "sual" <i>Note: Mid\$ can also be used on the left side of an assignment statement, where you can replace a substring within a string.</i> strTest = "Visual Basic" Mid\$(strTest, 3, 4) = "xxxx" 'strTest now contains "Vixxxx Basic" In VB6, the Replace\$ function was introduced, which can also be used to replace

	characters within a string.
Function	StrReverse(String)
Description	Returns a string in which the character order of a specified string is reversed.
Example	<pre>strTest = StrReverse\$("Visual Basic") strTest = "cisaB lausiV"</pre>

String-Related Functions

The string-related functions manipulate and analyze string data. One of Visual Basic's greatest strengths over other programming languages, thanks to Visual Basic's BASIC background, is the strong support for string data.

a) Len() Determines Length

Len() is one of the few functions that can take either a numeric variable or a string for its argument—although you'll use Len() primarily for string data. Len() returns the number of memory bytes needed to hold its argument.

Here's the format of Len():

Len(Expression)

Len() accepts any string value (variable, literal, or expression). However, only numeric variables, not numeric literals or expressions, work as Len() arguments.

Len() returns the length (number of characters) of the string variable, string constant, or string expression inside its parentheses. The following MsgBox() function displays a 6 as its output:

```
intMsg = MsgBox(Len("abcdef"))
```

If the string contains Null, Len() returns a value of 0. Testing for a null string lets you test to see whether a user entered data in response to an InputBox() function or a control value.

Converting Strings

Several conversion functions work with string data. Table 8.4 describes each string-conversion function used in the following examples.

The string-conversion functions convert to and from the String data type.

CStr() Changes its argument to a string **Str()** Converts its numeric argument to a string (actually, to a Variant data type that you can use as a string) **Val()** Converts its string argument to a number, assuming that you pass **Val()** a string-like number **CStr()** and **Str()** convert their arguments to string values. The only difference is that **CStr()** doesn't add a leading blank before positive numbers converted to strings; **Str()** does. Listing 8.5 demonstrates the difference between **CStr()** and **Str()**. **Str()** adds a leading blank before positive numbers that **CStr()** does not add.

```
1: Private Sub convStr ()
2: Dim str1 As String, s2 As String
3: Dim intMsg As Integer ' For button clicked
4: str1 = CStr(12345)
5: str2 = Str(12345)
6: intMsg = MsgBox("****" & str1 & "****")
7: intMsg = MsgBox("****" & str2 & "****")
8: End Sub
```


Line 6 produces a message box that displays `***12345***`, and Line 7 displays `*** 12345***`. Notice the blank before the number that `Str()` added.

The ASCII Functions

Use `Chr()` and `Asc()` to convert strings to and from their numeric ASCII values. (ASCII – American Standard Code for Information Interchange). The ASCII table lists every possible characters available on the PC and assigns a sequential number (an *ASCII code*) to each character. `Asc("A")` would return 65 and `Asc("a")` would return 97. Capital Z ends at 90 and lower case z at 122.

By putting a number inside the `Chr()` parentheses, you can produce the character that corresponds to that number in the ASCII table. By using `Chr()`, you can generate characters for variables and controls that don't appear on your computer's keyboard but that do appear in the ASCII table.

The `Asc()` function is a mirror image of `Chr()`. Whereas `Chr()` takes a numeric argument and returns a string character, `Asc()` requires a string argument and converts that argument to its corresponding ASCII table number.

Therefore, an A is stored in `strVar` in the following assignment statement because the ASCII value of A is 65:

```
strVar = Chr(65) ' Stores an A in aVar
```

Of course, it makes more sense to store an A directly in the `strVar` variable in the preceding example's statement. But what if you want to ask a Spanish question inside a message box? Spanish questions always begin with an upside-down question mark, and no upside-down question mark appears on your keyboard. Therefore, you can resort to using `Chr()` as follows:

```
' Chr(241) produces an n with a tilde over it
strMyQuest = Chr(191) & "Se" & Chr(241) & "¿o, como esta?"
intMsg = MsgBox(strMyQuest)
```

`Asc()` returns the ASCII number of the character argument you give it. You can find the ASCII numbers by searching Visual Basic's online help. The argument must be a string of one or more. If you pass `Asc()` a string of more than one character, it returns the ASCII number of the first character in the string.

The following code demonstrates a good use for `Asc()`:

```
strAns = InputBox("Do you want to see the name")
If ((Asc(strAns) = 89) Or (Asc(strAns) = 121)) Then
b = MsgBox("The name is " + aName)
End If
```

Use ASCII characters to display characters not on the keyboard.

The user can answer the prompt with y, Y, Yes, or YES. The `If...Then` test works for any of those input values because 89 is the ASCII value for Y, and 121 is the ASCII value of y. `Asc()` returns the ASCII value of its string argument's first letter.

The Substring Functions Practical

The substring functions return parts of strings.

1. **Right()**

Returns characters from the right side of a string. E.g.

Right(stringValue, numericValue)

2. **Left()**

Returns characters from the left side of a string. E.g.

Left(stringValue, numericValue)

3. **Mid()**

Lets you pick characters from the middle of a string.

Mid(stringValue, startPosition[, length])

The following section of code demonstrates Left():

```
strA = "abcdefg"  
partSt1 = Left(strA, 1) ' Stores a  
partSt2 = Left(strA, 3) ' Stores abc  
partSt3 = Left(strA, 20) ' Stores abcdefg
```

If you try to return more characters from the left of the string than exist, Left() returns the entire string.

Right() works in the same manner as Left(), except that it returns the rightmost characters from a string:

```
strA = "abcdefg"  
partSt1 = Right(strA, 1) ' Stores g  
partSt2 = Right(strA, 3) ' Stores efg  
partSt3 = Right(strA, 20) ' Stores abcdefg
```

Mid() accomplishes what Left() and Right() can't—it returns characters from the middle of a string. Mid() uses three arguments: a string followed by two integers. The first integer determines where Mid() begins stripping characters from the string (the position, starting at 1), and the second integer determines how many characters from that position to return. If you don't specify two integers, Mid() uses 1 as the starting position.

Mid() can pull any number of characters from anywhere in the string. The following example shows how the Mid() function works:

```
strA = "Visual Basic FORTRAN COBOL C Pascal"  
lang1 = Mid(strA, 1, 12) ' Stores Visual Basic  
lang2 = Mid(strA, 14, 7) ' Stores FORTRAN  
lang3 = Mid(strA, 22, 5) ' Stores COBOL  
lang4 = Mid(strA, 28, 1) ' Stores C  
lang5 = Mid(strA, 30, 6) ' Stores Pascal
```

If you don't specify the length argument, VB returns all the characters to the right of the starting position.

If the length is longer than the rest of the string, VB ignores the length argument.

Mid() works both as a command and a function. It works as a command when it appears on the left side of an assignment statement's equal sign; it's a function when it appears anywhere else. Following is its format:

Mid(string, start[, length])

When you use the Mid() statement, Mid() changes the contents of the string used inside the statement's parentheses. The following code initializes a string with three words and then changes the middle word with Mid():

```
strSentence = "Rain in Spain"  
' Change the middle word  
Mid(strSentence, 6, 2) = "on"  
' After the change intMsg = MsgBox("After change: " & strSentence)  
' Prints Rain on Spain
```

Converting to Uppercase and Lowercase

The **UCase()** function returns its string argument in all uppercase letters. **LCase()** returns its string argument in all lowercase letters. The following **MsgBox()** function displays VISUAL BASIC:
`intMsg = MsgBox(UCase("Visual Basic"))`

Justifying and Trimming Strings

LTrim() and **RTrim()** trim spaces from the beginning or end of a string. **LTrim()** returns the argument's string without any leading spaces. **RTrim()** returns the argument's string without any trailing spaces. **The Trim() function trims leading and trailing spaces.**

Here are the formats of the string-trimming functions:

`LTrim(stringExpression)`

`RTrim(stringExpression)`

`Trim(stringExpression)`

The following statements trim spaces from the beginning, end, or both sides of strings:

`st1 = LTrim(" Hello")` ' Stores Hello

`st2 = RTrim("Hello ")` ' Stores Hello

`st3 = Trim(" Hello ")` ' Stores Hello

Without the trimming functions, the spaces are copied into the target variables as well as the word Hello.

Str() always converts positive numbers to strings with a leading blank (where the imaginary plus sign appears); therefore, you can combine **LTrim()** with **Str()** to eliminate the leading blank. The first of the following two statements stores the leading blank in `str1`. The second uses **LTrim()** to get rid of the blank before storing the string into `str2`:

`str1 = Str(234)` ' Stores " 234"

`str2 = LTrim(Str(234))` ' Stores "234"

The following **StrReverse()** function includes several of the string functions described in today's lesson.

This function, shown in Listing 8.6, reverses a certain number of characters within a string.

This function reverses a string by using the string functions.

```
Public Function StrReverse (strS As String, ByVal n As Integer) As _String
```

```
    ' Accepts: a string and an integer indicating the number of
```

```
    ' characters to reverse
```

```
    ' Purpose: reverses the specified number of characters in the
```

```
    ' specified string
```

```
    ' Returns: the modified string
```

```
    ' Reverses the first n characters in s.
```

```
Dim strTemp As String, intI As Integer
```

```
If n > Len(strS) Then n = Len(strS)
```

```
For intI = n To 1 Step -1
```

```
    strTemp = strTemp + Mid(strS, intI, 1)
```

```
Next intI
```

```
StrReverse = strTemp + Right(strS, Len(strS) - n)
```

```
End Function
```

Suppose that the **StrReverse()** function is called with the following statement:

```
newStr = StrReverse ("Visual Basic", 6)
```

If all goes well, the string named newStr will hold the characters lausiV Basic (the first six characters are reversed). Line 10 declares two local variables, the first of which, a string variable named strTemp, holds the reversed string as it's being built. The second variable, intl, is used in the For loop.

Starting with version 6, Visual Basic now includes a string function named StrReverse() that returns the reversed string of its single string argument. Listing 8.6, although certainly not as efficient as using StrReverse(), does help demonstrate the Mid() function.

Line 12's If statement ensures that the integer passed to StrReverse() isn't larger than the length of the string passed. It's impossible to reverse more characters than exist in the string. If more characters are passed, the If statement ensures that the entire string is reversed by changing the length to reverse to the exact length of the string via the Len() function.

In line 13, the For loop then counts down, from the position to reverse (stored in n) to 1. By using the Mid() function in line 14, Visual Basic concatenates one character from the string, at position n, to the new string being built. As n reaches 1, the reversed characters are sent to the new string in line 14. After all the characters that need to be reversed are reversed, the rightmost portion of the passed string is concatenated as is to the reversed characters.

Counting Number of Words In A String

```
Dim a As Integer, words As Integer
words = 1
For a = 1 To Len(Text1.Text)
If Mid(Text1.Text, a, 1) = " " Then
words = words + 1
End If
Next
MsgBox ("No of words: " & words)
```

Whatsapp 0726 733 444