GUI (Graphical User Interface):

A graphical user interface, often abbreviated as GUI, is a type of user interface using graphics for interaction with a computer or other media formats which employs graphical images, widgets, along with text to represent the information and actions available to a user. The actions are usually performed through direct manipulation of the graphical elements.

A program interface that takes advantage of the computer's graphics capabilities to make the program easier to use. Well-designed graphical user interfaces can free the user from learning complex command languages. On the other hand, many users find that they work more effectively with a command-driven interface, especially if they already know the command language.

Features of GUI:

Graphical user interfaces, such as Microsoft Windows and the one used by the Apple Macintosh, feature the following basic components:

- ***Pointer :*** A symbol that appears on the display screen and that you move to select objects and commands. Usually, the pointer appears as a small angled arrow. Text - processing applications, however, use an I-beam pointer that is shaped like a capital *I*.
- ***Pointing device :*** A device, such as a mouse or trackball, that enables you to select objects on the display screen.
- ***Icons :*** Small pictures that represent commands, files, or windows. By moving the pointer to the icon and pressing a mouse button, you can execute a command or convert the icon into a window. You can also move the icons around the display screen as if they were real objects on your desk.
- ***Desktop :*** The area on the display screen where icons are grouped is often referred to as the desktop because the icons are intended to represent real objects on a real desktop.
- ***Windows:*** You can divide the screen into different areas. In each window, you can run a different program or display a different file. You can move windows around the display screen, and change their shape and size at will.
- ***Menus :*** Most graphical user interfaces let you execute commands by selecting a choice from a menu.

In addition to their visual components, graphical user interfaces also make it easier to move data from one application to another. A true GUI includes standard formats for representing text and graphics. Because the formats are well-defined, different programs that run under a common GUI can share data. This makes it possible, for example, to copy a graph created by a spreadsheet program into a document created by a word processor.
Many DOS programs include some features of GUIs, such as menus, but are not graphics based. Such interfaces are sometimes called graphical character-based user interfaces to distinguish them from true GUIs.

Refers to software and hardware that treat objects on a display screen as bit maps or geometrical shapes rather than as characters. In contrast, character-based systems treat everything as ASCII or extended ASCII characters.
All graphics software is by definition graphics based. Systems that manipulate text can also be graphics based; for example, desktop publishing systems are essentially graphics-based word processors.
Traditionally, most DOS applications -- word processors, spreadsheets, and database management systems -- have been character based. Windows and the Mac OS are graphics-based.

Murphy's Law:
Murphy's law is a popular adage(an edge is a short) in Western culture that most likely originated at Edwards Air Force Base in 1948. The Law broadly states that things will go wrong in any given situation, if you give them a chance. "If there's more than one way to do a job, and one of those ways will result in disaster, then somebody will do it that way." It is most often cited as "Whatever can go wrong, will go wrong" (or, alternately, "Whatever can go wrong will go wrong, and at the worst possible time," or, "Anything that can go wrong, will," or even, "If anything can go wrong, it will, and usually at the most inopportune moment"). However, this interpretation has given rise to confusion with Sod's law and Finagle's law.
Per the 1948 theory, in American culture the law was named somewhat sarcastically[1][2] by Stapp's Team working on Project MX981 at Edwards Air Force Base after Major Edward A. Murphy, Jr., a development engineer contributing support measurement technology for a brief time on rocket sled experiments done by the United States Air Force in 1948 with inveterate adage collector and the law's undoubted popularizer Doctor/Colonel John Paul Stapp, a former next-door neighbor and friend of Murphy.

The correct, original Murphy's Law reads: "If there are two or more ways to do something, and one of those ways can result in a catastrophe, then someone will do it." This is a principle of defensive design. For example, you don't make a two-pin plug symmetrical and then label it "THIS WAY UP"; if it matters which way it is plugged in, then you make the design asymmetrical.

Edward A. Murphy, Jr. was one of McDonnell-Douglas's test engineers on the rocket-sled experiments that were done by the U.S. Air Force in 1949 to test human acceleration tolerances (USAF project MX981). One experiment involved a set of 16 accelerometers mounted to different parts of the subject's body. There were two ways each sensor could be glued to its mount, and somebody methodically installed all 16 in a replacement set the wrong way around. Murphy then made the original form of his pronouncement, which the test subject mis-quoted (apparently in the more general form "Whatever can go wrong, will go wrong)" at a news conference a few days later.

Icons:
A small picture that represents an object or program. Icons are very useful in applications that use windows, because with the click of a mouse button you can shrink an entire window into a small icon. (This is sometimes called minimizing.) To redisplay the window, you merely move the pointer to the icon and click (or double click) a mouse button. (This is sometimes called restoring or maximizing)
Icons are a principal feature of graphical user interfaces.

Graphics:
Refers to any computer device or program that makes a computer capable of displaying and manipulating pictures. The term also refers to the images themselves. For example, laser printers and plotters are graphics devices because they permit the computer to output pictures. A graphics monitor is a display monitor that can display pictures. A graphics board (or graphics card) is a printed circuit board that, when installed in a computer, permits the computer to display pictures.
Many software applications include graphics components. Such programs are said to support graphics. For example, certain word processors support graphics because they let you draw or import pictures. All CAD/CAM systems support graphics. Some database management systems and spreadsheet programs support graphics because they let you display data in the form of graphs and charts. Such applications are often referred to as business graphics.

The following are also considered graphics applications :

- Paint programs : Allow you to create rough freehand drawings. The images are stored as bit maps and can easily be edited.
- Illustration/design programs: Supports more advanced features than paint programs, particularly for drawing curved lines. The images are usually stored in vector-based formats. Illustration/design programs are often called draw programs.
- Presentation graphics software : Lets you create bar charts, pie charts, graphics, and other types of images for slide shows and reports. The charts can be based on data imported from spreadsheet applications.
- Animation software: Enables you to chain and sequence a series of images to simulate movement. Each image is like a frame in a movie.
- CAD software: Enables architects and engineers to draft designs.
- Desktop publishing : Provides a full set of word-processing features as well as fine control over placement of text and graphics, so that you can create newsletters, advertisements, books, and other types of documents.

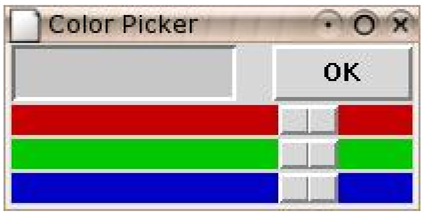
In general, applications that support graphics require a powerful CPU and a large amount of memory. Many graphics applications—for example, computer animation systems—require more computing power than is available on personal computers and will run only on powerful workstations or specially designed graphics computers. This is true of all three-dimensional computer graphics applications.
In addition to the CPU and memory, graphics software requires a graphics monitor and support for one of the many graphics standards. Most PC programs, for instance, require VGA graphics. If your computer does not have built-in support for a specific graphics system, you can insert a video adapter card.
The quality of most graphics devices is determined by their resolution—how many points per square inch they can represent—and their color capabilities.

Identifying Visual Cues:
A cue is a signal of something or a reminder of something. It brings to mind something from past knowledge or previous experience that provides a framework of meaning that can be used to interpret the sign. The concept of cueing is very important to visual communication because much of past experience is filed in memory as a visual element. In other words, while cues can and do work on the semantic level for certain types of information, perceptual psychologists focus more on the tremendous role of visual imagery in the cueing process based as it is on experiential knowledge.
Advertising, with its highly condensed message formats, uses a shortcut form of information processing. Through association, two thoughts--usually a product and a selling message--are connected in the mind. If this works successfully, when you think about the selling message with its visual cues, you recall the product and vice versa. Clearly this message strategy is heavily dependent on the successful functioning of the cueing process. There simply isn't time in most advertisements for elaborated message development, so the message designers depend upon cues to elicit the associated meanings. In other words, cueing drives the process of association.
A message is encoded by the source, transmitted through a channel and decoded by a receiver. In advertising the message is put into words and pictures by a creative team, approved by a client, distributed through a medium like television or magazines, and--assuming it gets attention--it is decoded by the targeted audience. In order for the audience to make sense of the information, however, the message has to use appropriate signs and symbols to stimulate the individual's perceptual system into action.

Clear Communication:
Put the options on push buttons. Buttons and menus can handle longer phrases than pull-downs. Get clear communication for casual and new users. You can position the push button on the screen at the place the user needs it, matching the task flow as it goes in a left-right, top-down pattern. If a command appears on several screens, try to design a task flow that allows a consistent position for the buttons.

When using pull-downs, never change menu bar names! (Play the slots with a slot machine, not a menu bar.) If an option doesn't apply in a particular situation, deactivate it by dimming it rather than "offing" it.

Color Selection
Left clicking on any color Widget/Option position will bring up the Color Selection Menu shown below.



The color is selected by varying the RGB settings using the lower sliders. The current color will appear in the upper left channel as the settings are changed. When the desired color is achieved click on OK to set it.
To copy the color in one setting to another, left click on the setting with the color to be copied. Click OK on the Color Selection Menu, then right click on the destination button. When Color Selection Menu appears if should hold the color of the previous selection. Click OK to set the color.

Widget:
RADIO AND CHECK BUTTONS: RADIO and CHECK BUTTONS are binary selection widgets. These are widgets which represent an on/off state. The difference between the two classifications is in a set of linked Radio Buttons only one of the associated buttons is allowed to be in the on state, where as with check buttons any or all of the buttons may be in the on state.
SLIDERS: SLIDERS are scroll buttons which are generally associated with List and Text Boxes.
ENTRY BOX: ENTRY BOXES define areas where text can be manually inserted.
BUTTONS: BUTTONS covers all classes of buttons outside the Check and Radio button classes. This includes, Menu, Push, and Toggle Buttons.
MENU OPTIONS: MENU OPTIONS are the list of options popped up whenever a menu button is activated.
LIST BOX: LIST BOXES are lists of text from which items can be selected (hilighted) and deselected (un-hilighted).
LABELS: LABELS are simple text widgets.
TEXT BOX: TEXT BOXES are areas in which text or information is displayed.
FRAMES: FRAMES define areas of a menu into which various widgets have been placed.

GUI Standard:

- Platform independent.
- Supports several interaction styles.
- Captures a lot of the usual User Interface Management System (UIMS) abstractions.
- Powerful graphics model.
- Object-oriented with lots of reuse.
- Simple user interfaces are quick and easy to implement.
- Powerful command line interaction model.
- Presentation types are a nice building block for very interactive user interfaces via context sensitive input.
- Reference implementation is written mostly in portable Common Lisp plus some backend magic.
- Supports formatted output for lists, tables, graphs, ...
- Supports native gadgets.
- Some adaption to the platforms look and feel.
- Output recording captures graphics output.
- Incremental redisplay to minimize (re-)drawing.
- Layout descriptions.
- Several backends have been written over time (Lisp Machine, X11/Motif, CAPI, Windows, MacOS, Postscript and a partial backend to web pages).

GUI Design:

- Use widgets according to their function (e.g., radio buttons should not trigger actions).
- Differentiate clearly between control and indicator (read-only) fields. Do not use a text entry field to display read-only information (#1 blooper).
- Use widgets as appropriate for the input data type.
  - This can have the added advantage of automatic input validation and minimizing parsing or use of regular expressions.
- Use an action label for all buttons (e.g., Open, Display).
- Do not use toggle buttons (e.g., text changes on the button depending upon state).
- Color should be used very sparingly.
  - Color alone should not be used to distinguish important items (e.g., You can use color in conjunction with shape or text.)
- LBT Color Convention (ask for exact settings)
  - Grey – OFF, OK/normal/nominal
  - Green – ON, OK/normal/nominal
  - Red – ON or OFF, error/alarm indicator
  - The only Red present on LBT GUIs should be for alarms.
  - Yellow – ON or OFF, warning indicator
  - Yellow and Black cross hatch – PANIC BUTTONS
- Do not hardcode colors in your program.
  - There will be a predefined alternative set of colors to accommodate color blindness.
- Only use ellipses ("…") when more information is needed before the request can be fulfilled.
- Use bold, Helvetica font (Helvetica or Sans Serif is the default on Linux/Unix systems).
- If an action is in progress, keep the user informed (e.g., use a progress bar, slow flashing, etc.).
- Clearly label items and indicate the units where appropriate.
- Minimize mouse motion.
- Implement mouse-less navigation and "key press" action.
- Implement "short cuts" and "accelerator keys".

- Group items, as appropriate, for clarity.
- Make sure all dialogs have a “Close” or “OK/Cancel” to shutdown the dialog. Do not make the user rely on the “X”.
- Use variable names which reflect purpose. In particular, the prefix (last few letters) should indicate the widget used. For example, vdOpenAllBut – ventilation door, Open All button

Visual Programming:

A Visual programming language (VPL) is any programming language that lets users specify programs by manipulating program elements graphically rather than by specifying them textually. A VPL allows programming with visual expressions, spatial arrangements of text and graphic symbols. Most VPLs are based on the idea of "boxes and arrows," that is, boxes or circles or bubbles, treated as screen objects, and connected by arrows, lines or arcs.
VPLs may be further classified, according to the type and extent of visual expression used, into icon-based languages, form-based languages, and diagram languages. Visual programming environments provide graphical or iconic elements which can be manipulated by users in an interactive way according to some specific spatial grammar for program construction.
A visually transformed language is a non-visual language with a superimposed visual representation. Naturally visual languages have an inherent visual expression for which there is no obvious textual equivalent.
Current developments try to integrate the visual programming approach with dataflow languages to either have immediate access to the program state resulting in online debugging (i.e. Lab VIEW) or automatic program generation and documentation (i.e. visual paradigm). Dataflow languages also allow automatic parallelization, which is likely to become one of the greatest programming challenges of the future.

- A programming language that uses a visual representation (such as graphics, drawings, animation or icons, partially or completely)
- A visual language manipulates visual information or supports visual interaction, or allows programming with visual expressions.
- Any system where the user writes a program using two or more dimensions.
  A visual language is a set of spatial arrangements of text-graphic symbols with a semantic interpretation that is used in carrying out communication actions in the world.

Software Component:

A software component is a system element offering a predefined service and able to communicate with other components.

- Multiple-use
- Non-context-specific
- Composable with other components
- Encapsulated i.e., non-investigable through its interfaces
- A unit of independent deployment and versioning

A simpler definition can be: A component is an object written to a specification. It does not matter what the specification is: COM, Java Beans, etc., as long as the object adheres to the specification. It is only by adhering to the specification that the object becomes a component and gains features like reusability and so forth.
Software components often take the form of objects or collections of objects (from object-oriented programming), in some binary or textual form, adhering to some interface description language (IDL) so that the component may exist autonomously from other components in a computer.

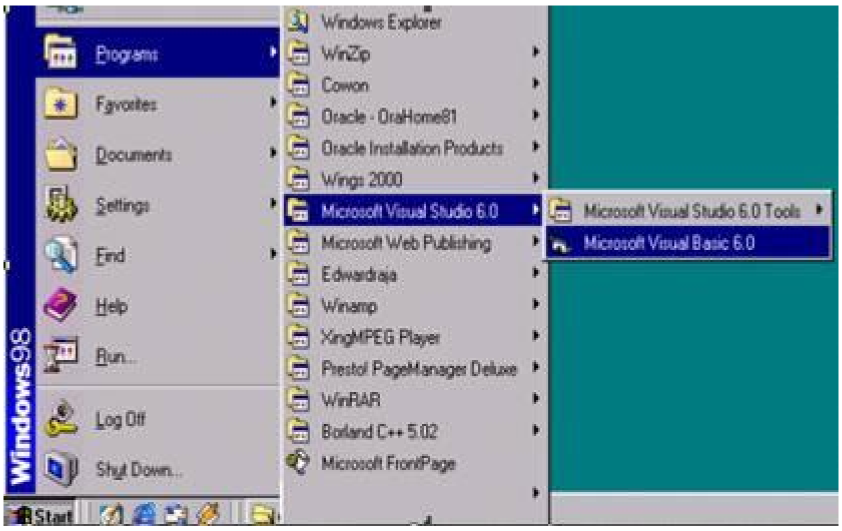
When a component is to be accessed or shared across execution contexts or network links, some form of serialization (also known as marshalling) is employed to turn the component or one of its interfaces into a bit stream.

Getting started.

Hardware Requirements.

Visual Basic 6.0 for windows requires Microsoft Windows 95/Windows NT 4.0, 486 processor and a minimum of 16 MB RAM.

To open Visual Basic application, Click Start -> Programs -> Microsoft Visual Studio 6.0 a Microsoft Visual Basic 6.0. as shown below.



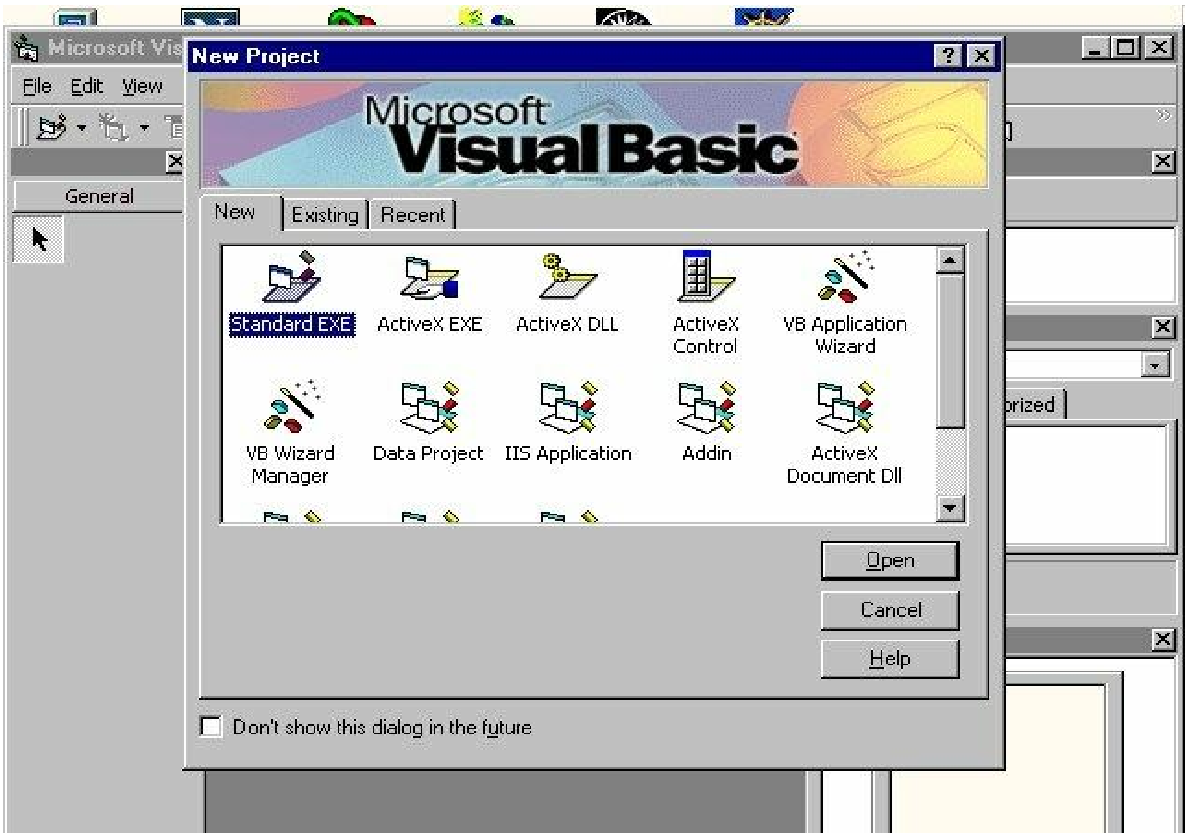
On start up, Visual Basic 6.0 will display the following dialog box as shown in figure.

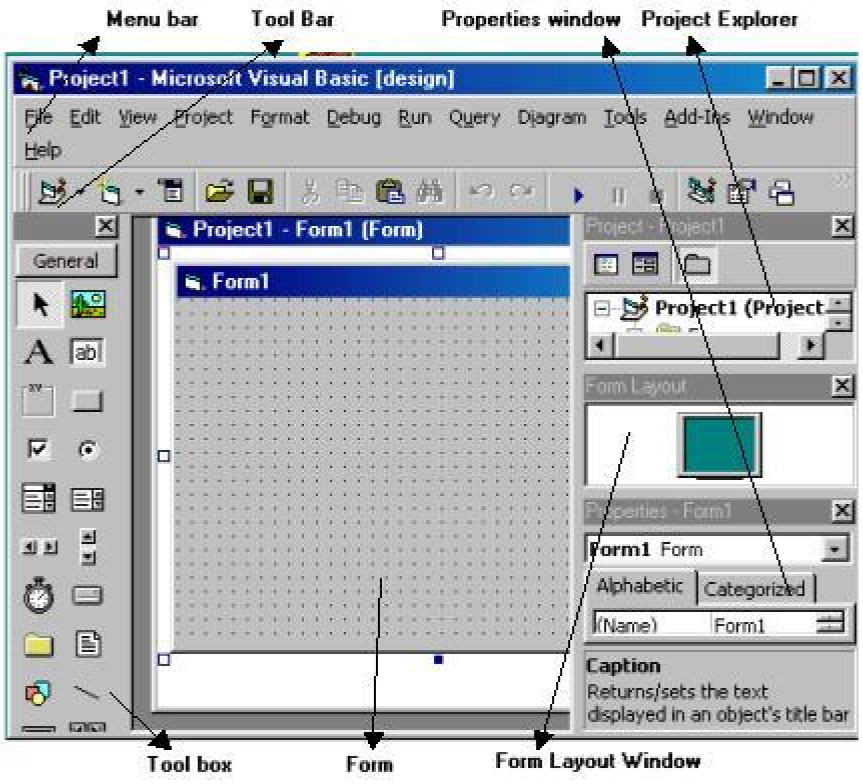
The opening screen has three tab options:

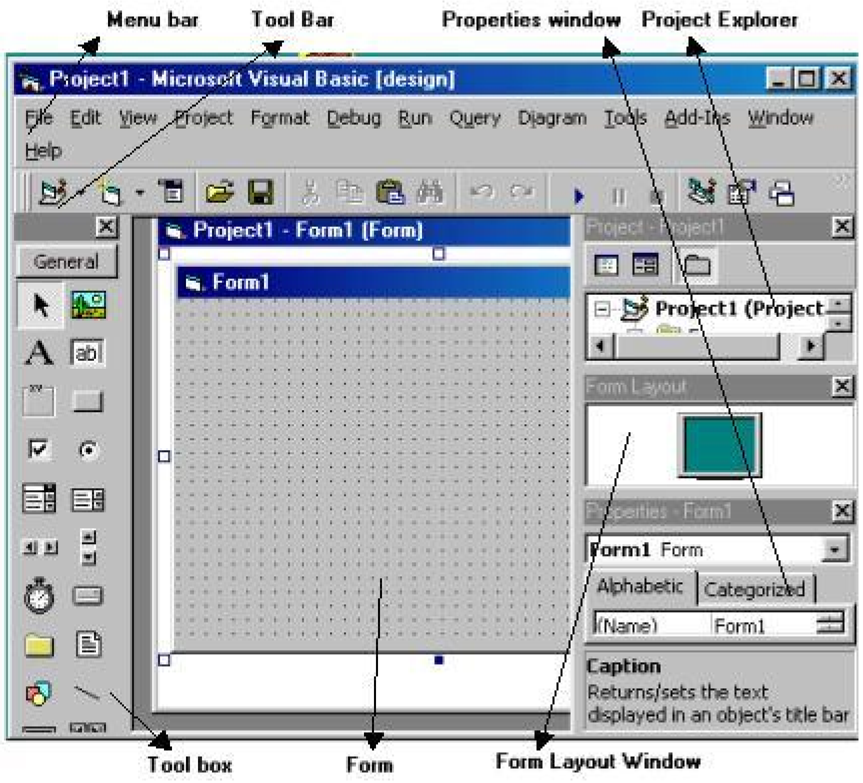
- New,opens a new Project.
- Existing,opens a dialog box from which the users can select the existing Visual Basic Projects.
- Recent, shows a list of Projects created in Visual Basic that have been recently accessed or created.

A project is a collection of files,that make up your application. There are various types of applications, one could create, however, let us concentrate on creating, Standard EXE rograms. (EXE means executable program).

Click on the Standard EXE icon, to go into the actual Visual Basic programming environment.


New Project
Microsoft Visual Basic
New
Existing
Recent
Standard EXE
ActiveX EXE
ActiveX DLL
ActiveX Control
VB Application Wizard
VB Wizard Manager
Data Project
IIS Application
Addin
ActiveX Document Dll
Open
Cancel
Help
Don't show this dialog in the future


Menu bar
Tool Bar
Properties window
Project Explorer
Project1 - Microsoft Visual Basic [design]
File Edit View Project Format Debug Run Query Diagram Tools Add-Ins Window Help
General
Project1 - Form1 (Form)
Form1
Project - Project1
Project1 (Project
Form Layout
Properties - Form1
Form1 Form
Alphabetic
Categorized
(Name) Form1
Caption
Returns/sets the text displayed in an object's title bar
Tool box
Form
Form Layout Window



In the above figure, the Visual Basic Environment consists of,

- The Project window,displays the files, that are created in your application.

It also includes a Toolbox, that consists of all the controls essential for developing a VB Application. Controls are tools such as boxes, buttons, labels and other objects draw on a form to get input or display output. They also add visual appeal.

Interface.

Tool Bar.

Provides a quick access to commonly used commands in the programming environment.

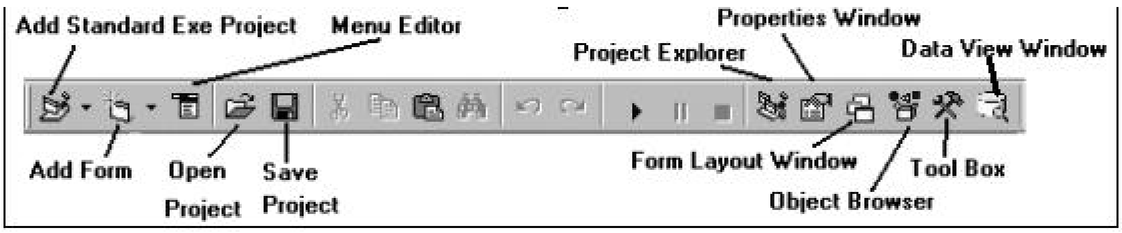
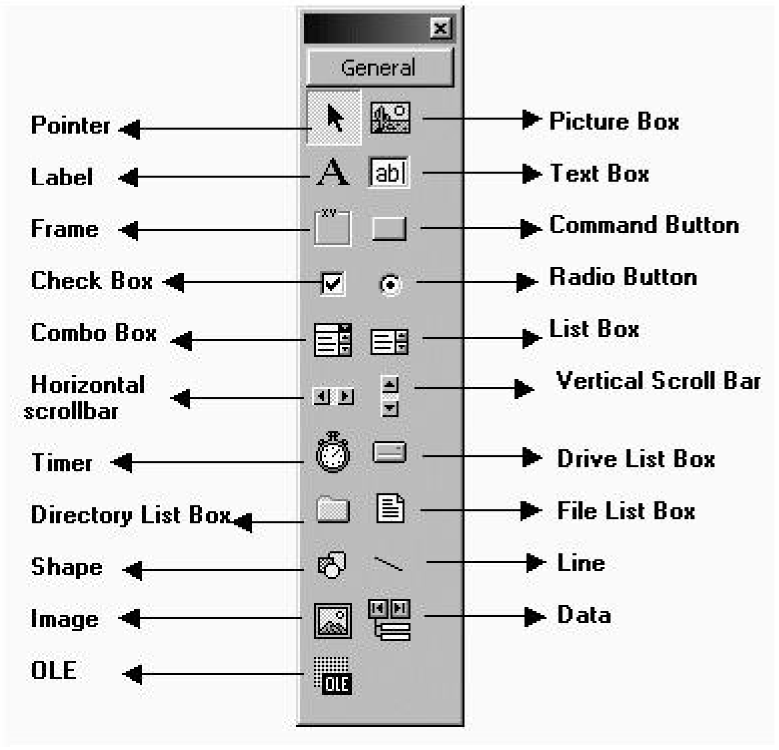


Figure Shows ToolBar.

Tool Box.

Provides a set of tools that the user uses at design time on a Form. Each tool in the toolbox represents a control.



Form.

In the **Blank Form** window which you can design your application's interface. You can add controls, graphics and pictures to a form to create the look you want.

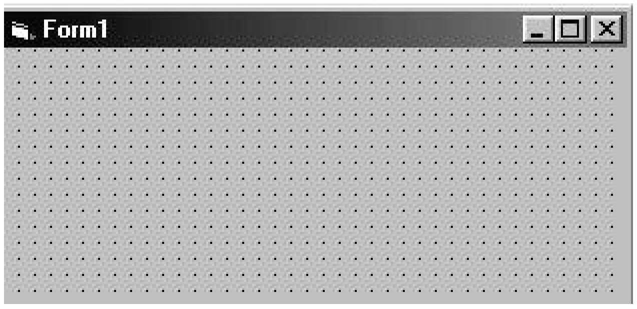


Figure Shows The Blank Form

Project Explorer.

This window lists the Forms and Modules in the current project. A project is a collection of files that we use to build an application.

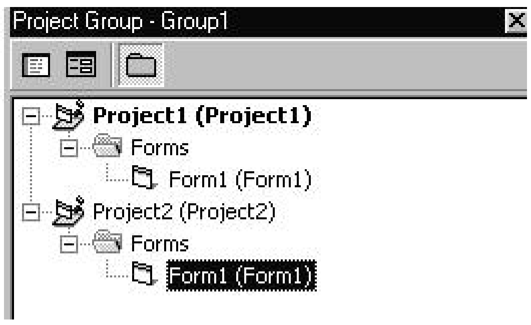
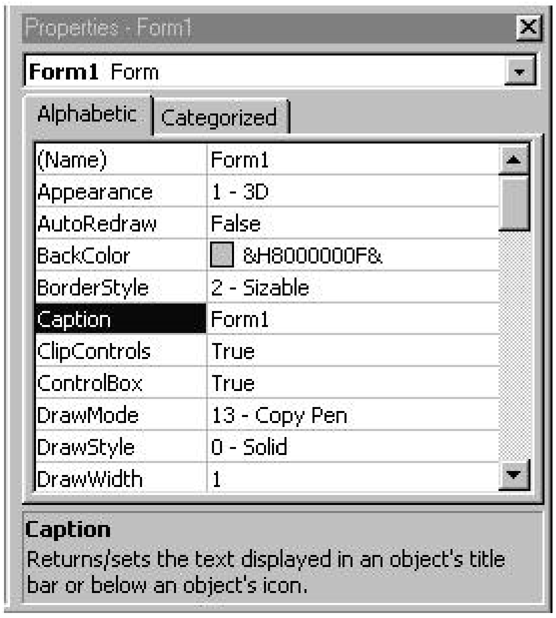


Figure Shows Project Explorer.

The Properties Window.

The **Properties** window, displays the properties of various, controls and objects that are created in your applications.



(Name)	Form1
Appearance	1 - 3D
AutoRedraw	False
BackColor	&H8000000F&
BorderStyle	2 - Sizable
Caption	Form1
ClipControls	True
ControlBox	True
DrawMode	13 - Copy Pen
DrawStyle	0 - Solid
DrawWidth	1

Caption
Returns/sets the text displayed in an object's title bar or below an object's icon.

Figure Shows Properties Window .

Form Layout View.

The view displays a sample layout of the Form being designed.

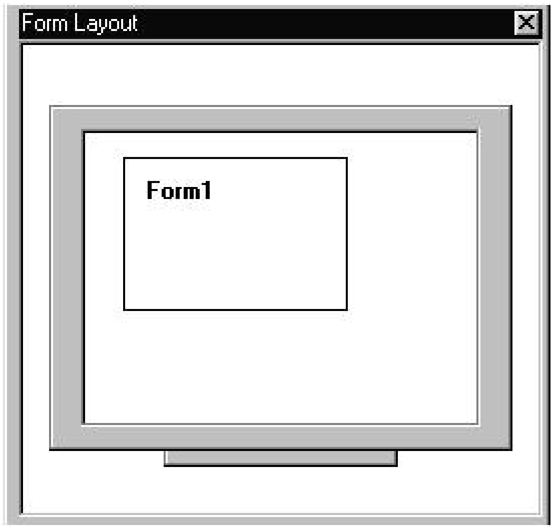


Figure Shows Form Layout View.

Context Menu.

Provides Visual Basic development environment that contains shortcuts to frequently performed actions. To open a contex menu, right click on the object being used correctly. The context menu displayed in response to this action is displayed below. This allows us to display the custom controls dialog box, hide the toolbox, or toggle it.

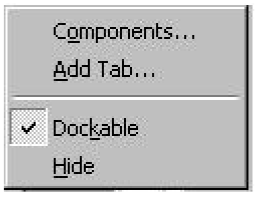


Figure Shows Context Menu.

Event-Driven Programming.

Applications using traditional or procedural programming is used to control the portions of code to be executed. Execution starts with first line of code and follows a pre-defined path through the application, calling procedures as needed.

In an event-driven application, the code doesn't follow a predetermined path – it executes different code sections in response to events. Events can be triggered by the user's actions, by messages from the system or other applications, or even from the application itself. The sequence of these events determines the sequence in which the code executes, thus the path through the application's code differs each time the program runs.

In traditional programming, the application development process can be divided into three steps – writing, compiling and testing code. In most of the languages if one makes a mistake while writing a code, the error is caught by the compiler at the time of compilation. One must find and fix the error and begin the compile cycle again, repeating the process for each error found.

The above limitations of traditional programming are eliminated in the event-driven programming. Unlike traditional languages, Visual Basic with the concept of event-driven programming uses an interactive approach to development. It interprets the code as one enters it; hence partial compiling is done while code is being entered. If the compiler finds an error, it is highlighted in the code. One can fix the error and continue compiling without having to start all over. Because of the interactive nature of Visual Basic one can find oneself running the application frequently as one develops it.

Actually, while programming in Visual Basic, it must first be decided how the application interacts with the user. In other words, it must be decided how each control reacts to user actions, such as the click of a mouse, keystrokes and so on and these reactions must be programmed. ***This is called Event-driven programming*** because the application does not determine the flow, instead, the events caused by the user determine the flow of the application.

Various Graphical User Interface Components.

Various Graphical User Interface components of Visual Basic are listed below:

- **Objects -** A general term used to describe all the forms and controls that make up a program.
- **Forms -** are Customizable windows that serve as the interface for an application or as dialog boxes that are used to gather information from the user.
- **Project -** Visual Basic application is comprised of one or more components that are arranged under a project. The project is stored as a file with .VBP extension.
- **Controls -** Graphical representation of objects, such as text boxes, labels, scroll bars, command buttons that users manipulate to provide information to the application.
- **Properties -** Each component's characteristic is specified by a property. Properties include names, captions, size, position, and contents. Visual Basic applies default properties. In Visual Basic it is also possible to change the properties at *design time or run time.*
- **Methods -** Built-in procedure that can be invoked to impart some action to a particular object.
- **Event -** Actions recognized by a form or control. Event occurs as the user, Operating System, or application interacts with the objects of a program.
- **Event Procedures -** Program code related to some object. This is the program code that is executed when a particular event occurs.
- **General Procedures -** Program code not related to objects. This program code must be invoked by the application.
- **Modules -** Collection of general procedures, variable declarations, and constant definitions used by application. In other words the modules are the files that hold procedure code. These files hold the form and its code, is technically called the *form module*.

Introduction to Controls.

A control is an object that can be drawn on a form object to enhance user interaction with the application. They respond to events initiated by the user or triggered by the system. They are classified into 3 categories:

- **Standard Controls:** Command Button, Text Box, Label, List Box etc. which appear by default in the Tool Box.
- **ActiveX (or) Custom Controls:** MS common dialog control, MS flex grid control, MS win sock control etc.
- **Insert able Objects:** MS Excel, MS Word, MS Power Point, Paint Brush etc.

Standard Controls.

Let us discuss some of the standard controls in Visual Basic.

Label Control .

It is used to display a text that a user cannot change directly. In addition, label control displays read-only text as far as the user is concerned, though one can alter the caption at run time.

Properties.

Name.

Used to give a name to a label control. Default value is 'Label1' for the first label that is created.

Capti on.

Used to set a value to display in the label.

Auto size .

Decides the adjustment of the text within the label. By default it is false and if set to true then text passed into the label gets adjusted automatically according to the label width.

Word Wrap.

As and when the display text is typed this property aligns the text to the width of the control.

Use Mnemonic.

It should be set to true if one wants to define a character in the caption property of the label as an access key.

Text Box Control .

A Text Box control is used to get input from the user or to display text. The Text Box is a small text editor

Properties.

Text.

The Text property is a string and can be used as an argument. This is the default property.

Locked.

By default it is false which enables the user to edit the content during run-time. If it is set as true, the contents cannot be edited.

PasswordChar.

It is used to indicate the characters that should be displayed when a user types in a Textbox. This property is used to create a password filed in a dialog box. This property does not affect the contents of the Textbox.

MaxLength.

It is zero, by default, which means that the text may be of any length. One can enter up to 3600 characters. If it is set, the number of characters entered is restricted to the number specified.

Multiline.

It determines whether the textbox control can hold single line or multiple lines of text. By default, it holds single line of text. It can accept 36000 characters.

Scroll Bars.

Scroll bars property allows to set a value indicating if the textbox has horizontal or vertical scroll bars. Single line text can have horizontal scroll bar so that the user can view any part of a long line of text. Multiline text box can have both the horizontal and vertical scroll bars. The possibility of the scroll bars setting would be None, Horizontal, Vertical or Both.

Tabindex .

It gives index for text boxes. Tabindex of a label must always be one less than the tabindex.

Methods.

Maxlength.

This property specifies the amount of text one can place in a Text Box control and is limited to approximately 64kb (a single line of text can hold only 255 characters).

Text Selection.

The TextBox control provides three properties for manipulating the text selected by the user: selText, selStart and selLength.

selText .

The selText property returns the selected text. To manipulate the currently selected text from within the code, the selText property is used. To delete the current selection, assign an empty string to the selText property.

selStart .

The selStart property returns or sets the position of the first character of the selected text.

selLength.

The SelLength property returns or sets the length of the selected text. The most common use of these properties is to extract the user’s selection or to select a piece of text from within the application.

Events.

Change.

Fires for every change made in the text box.

KeyPress.

Keys are trapped through this property.

Validate Event.

It is one of the most important event of the text box that has been introduced in Visual Basic 6.0. This fires before **lostfocus()** is fired.

Validate (cancel as Boolean).

If **cancel = false**, then cursor leaves the text box.

If **cancel = true**, then cursor does not leave the text box.

Validate event does not allow the cursor to leave the text box until the condition is false.

```
Private sub Text1_validate()

If text1 > 10 then

Cancel = false

Else

Cancel = true

End if

End sub
```

The priorities of occurrences of events of a text box are:

lostFocus – This event is fired when the control leaves the textbox.

gotFocus – This event is fired when the control enters the textbox.

Command Button .

A command button allows the user to interact with an application used to begin, interrupt or end a process. When chosen, a command button appears pushed in and hence is also called as a Push Button.

Properties:

- **Caption.** Displays text on a command button.
- **Back color.** Sets the command button's background color.
- **Fore color.** Set the color of text on a command button.
- **Name.** Reference name given to the command button.
- **Font.** Changes the Font of a command button's text.
- **Visible.** To make a command button visible or invisible at run-time.
- **TabIndex.** Always starts with zero. The control with least tabindex will get the first preference.Tab index is unique. No two controls can have the same tabindex.
- **Tabstop.** If the tabstop property is false the tab control does not enter into the textbox. Tabindex and tabstop work only with Tabkey.

Cancel .

The cancel property indicates whether a command button is the cancel button on a form. When a Command button control's cancel property setting is ' True and the form is the active form, the user can choose the command button by clicking it, pressing the ESC key or pressing ENTER when the button has the focus.

Default.

The default property is used to determine whether the command button control is the default command button on a form. When the command button's default property is true, the user can choose the command button by pressing the ENTER key provided the focus is not in any other command button. If the focus is on some other command button, then pressing ENTER chooses the command button that has the focus instead of the default command button.

Enabled: Activates or deactivates the command button function

Tool Tip Text: It is used to give a small text for the control, whenever mouse moves over that control.

Style: The style property determines if the command button is standard or graphical.

Standard:A command button is a standard command button without an associated graphic.

Graphical: Command button displays a picture that is associated with it in the picture property.

Default is standard: When made as graphical the following can be given:

Picture.

- Down Picture.
- Disabled Picture.

A down picture is displayed at run-time when the mouse is clicked down. On releasing it, the picture will become the existing one. Disabled picture fires only when the enabled property is set to false and it can be seen at run-time. When one gives a disabled picture the command button cannot be executed.

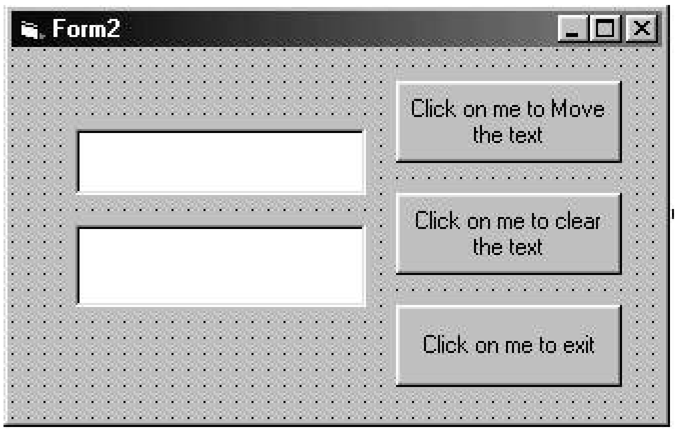
Example.

Drop 3 command buttons and name it as Move, Clear and Exit.

The 'Click on me to Move the text' button contains code to transfer the text from the textbox1 to textbox2.

The 'Click on me to Clear the text ' button clears the textboxes.

The 'Click on me to Exit' button terminates the application.

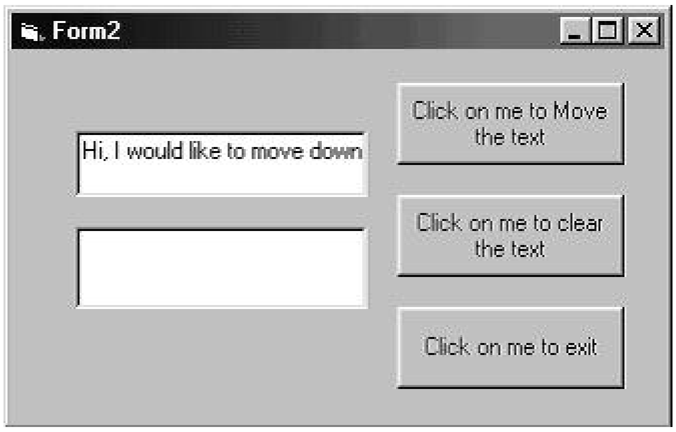


Demo with Textboxes and Command Buttons.

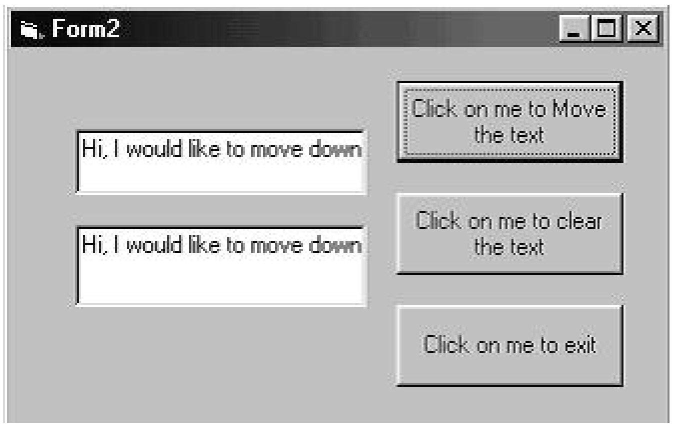
Type the code as given below in the respective control and event.

```
Private Sub CmdClear_Click()

Text1.Text = ""

Text2.Text = ""

End Sub

Private Sub CmdExit_Click()

End

End Sub

Private Sub CmdMove_Click()

Text2.Text = Text1.Text

End Sub
```

After execution you will see the following window.


Form2
Hi, I would like to move down
Click on me to Move the text
Click on me to clear the text
Click on me to exit

After entering text in the first Text box.


Form2
Hi, I would like to move down
Hi, I would like to move down
Click on me to Move the text
Click on me to clear the text
Click on me to exit

After Clicking on the First Command button.

Timer Control .

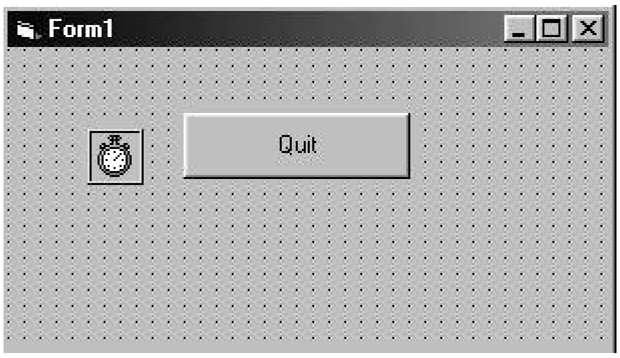
The timer control is one of the few controls always hidden at run time. The timer basically does just one thing. It checks the system clock and acts accordingly.

Interval : This property indicates the control to fire after the given number of milliseconds.

Example .

Drop a Timer Control and a Command Button.

Name the control and paste the code as given in the code window below.



Demo for Timer control.

The code given in the timer event of Timer1 control is fired for every 1 second. This is because the time interval is set for 1000 milliseconds; that is 1 second.

```
Private Sub CmdQuit_Click()

End

End Sub

Private Sub Form_Load()
```

```
Timer1.Interval = 1000

End Sub

Private Sub Timer1_Timer()

Print "I am called"

End Sub
```

Check Box Control .

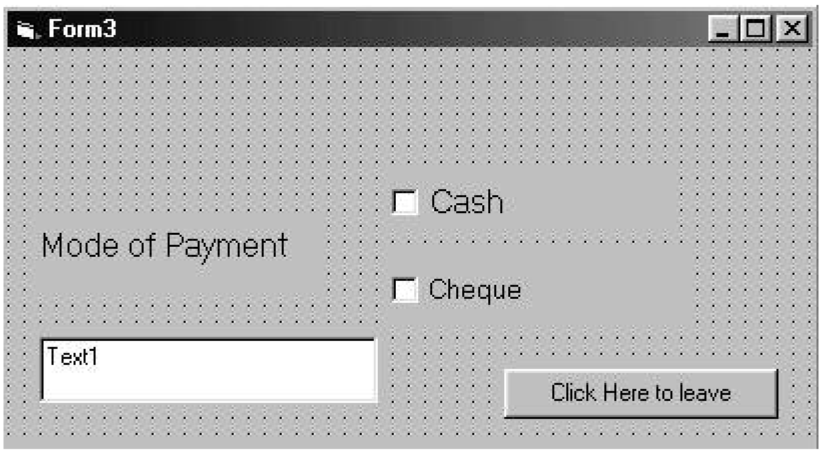
Check Box is used to select one or more options. A selected check box shows the selection with a checkmark and check boxes are never mutually exclusive. Therefore, the user can select one or more check boxes even if those check boxes reside in the same frame or on the same form.

Visual Basic added a new **Style** value to the Check Box control's property list. The available **Style** property values are **0-Standard** and **1-Graphical**. The graphical style value makes the check box look a lot like a command button that stays pressed (when selected) or unpressed (when not selected).

Example.

Drop a label, a text box, two Check boxes and a command button as shown below.

Name the controls with the appropriate caption as given and type the code respectively.

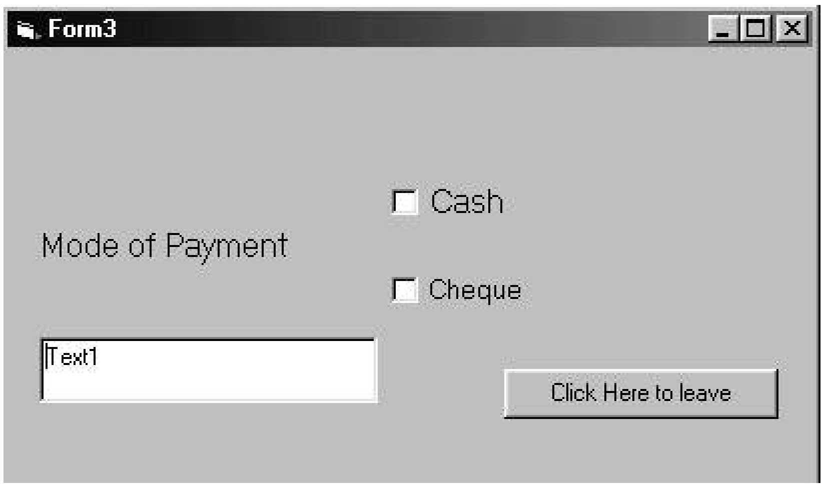


Demo for Checkbox.

```
Private Sub ChkCash_Click()

If ChkCash.Value = vbChecked Then

Text1.Text = "Cash is opted"

End If
End Sub

Private Sub ChkCheque_Click()

Text1.Text = "Cheque is opted"

End Sub

Private Sub CmdExit_Click()

End

End Sub
```

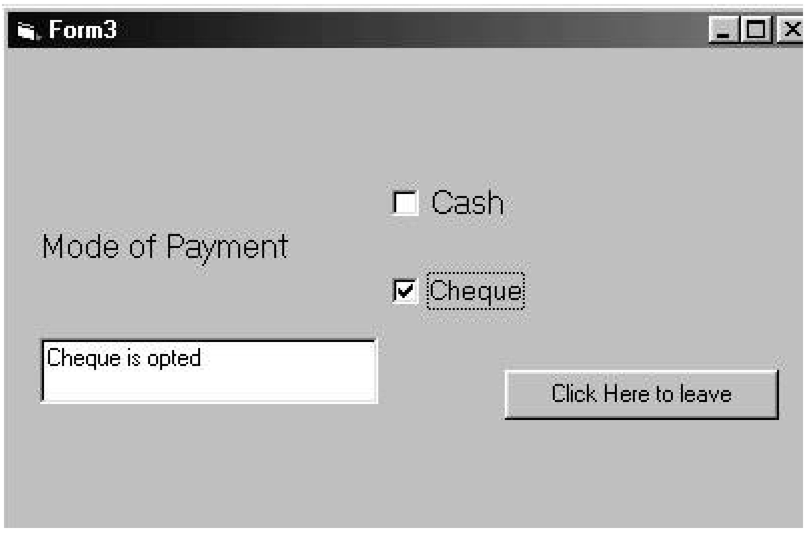
Before selecting Checkbox.

On execution, before clicking on cash or cheque check box you will see the following screen.



After selecting Checkbox.

After clicking on the Cheque Check box, the following output appears as shown Below.



Cheque is opted

Option Button Control:

Option buttons act in a mutually exclusive fashion. Only one option button can be selected at any one time. Option buttons are sometimes called **radio** buttons.

An option button gives the user a choice. By clicking the option button or by sending the focus to the option button and pressing the Spacebar to choose the option, the user selects or deselects an option button. The option button has a filled in circle inside its option button circle when selected.

The option button supports several properties such as the **Appearance** and **Alignment**. The **Alignment** property determines whether the option button text resides to the left or right of the option button. The **Appearance** determines whether or not an object is painted at run time with 3-D.

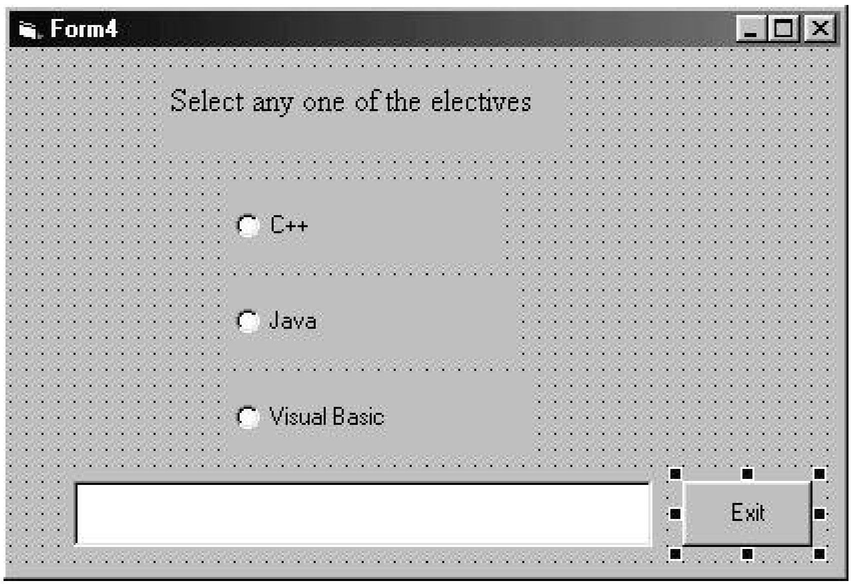
The **Value** property is perhaps the most important option button property because the **Value** property changes at runtime and determines whether the option button is currently selected.

Example.

Drop a Label, three option button, a text box and a command button.

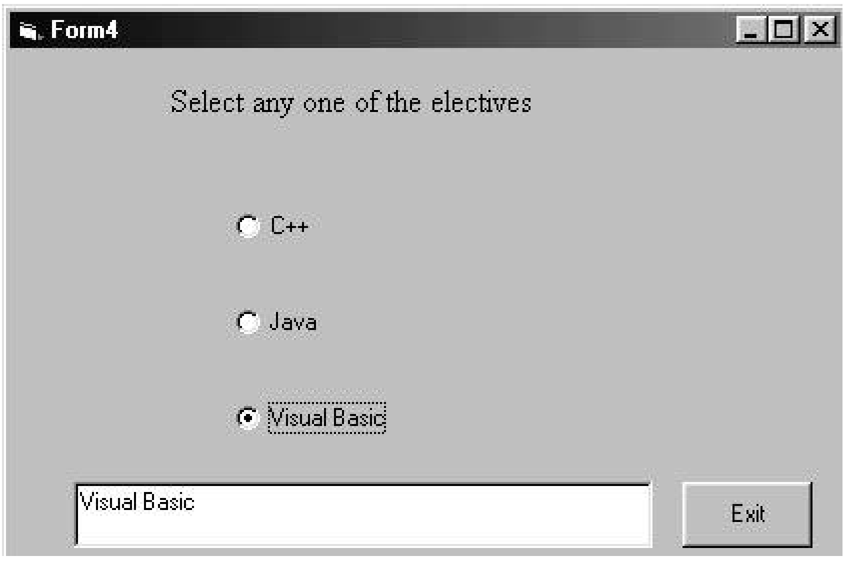
Name the controls as shown in the code window below and type the code respectively.

The user is allowed to have any one of the electives displayed. Even if he selects more than one elective, the earlier selected one will be deselected as the control used is Option Button.



Demo for Option Button.

```
Private Sub OptCpp_Click()

If OptCpp.Value = True Then

Text1.Text = OptCpp.Caption

End If

End Sub

Private Sub OptJava_Click()

If OptJava.Value = True Then

Text1.Text = OptJava.Caption

End If

End Sub

Private Sub OptVB_Click()

If OptVB.Value = True Then

Text1.Text = OptVB.Caption

End If

End Sub
```



Output window of Option Button Control.

Frames **and Option Buttons.**

Sometimes a form will need several sets of option buttons. In each set the user should be allowed to select only one option button. Only one option button inside a frame can be set at one time. A **frame** is a rectangular region on a form that holds other controls and groups the controls into a single set.

Once the controls are placed in a frame, the frames when moved all the frame's controls move with it. Therefore, adjusting framed controls is relatively easy to do. Always place a frame on the form before putting controls in the frame. If controls are moved from elsewhere on the form to the frame, the controls will not be in the frame but will exist simply on top of the frame. Visual Basic will not consider them framed together. To add additional controls to a frame with controls, click one of the framed controls before adding the new control.

Scrollbar Control.

Scrollbars allow users to control value changes. Rather than typing specific values, the user can move the scrollbars with the mouse to specify relative positions within a range of values. The toolbox includes both a **Horizontal Scrollbar** **and a Vertical Scrollbar control** .

Table contains a list of important scrollbar properties that determine the behavior of the scrollbar.

Property.	**Description.**
LargeChange.	Specifies the amount that the scrollbar changes when the user clicks within the scrollbar's shaft area.
Max.	Indicates the maximum number of units that the scrollbar value represents at its highest setting. The range is from 1 to 32767 (the default Max value).
Min.	Indicates the minimum number of units the scrollbar value represents at its lowest setting. The range is from 1 (the default Min value) to 32767 .
SmallChange.	Specifies the amount that the scrollbar changes when the user clicks an arrow at either end of the scrollbar.
Value.	Contains the unit of measurement currently represented by the position of

	the scrollbar.

Fundamental scrollbar properties .

When placing a scrollbar on a form, the user decides what range of values the scrollbar would represent. The scrollbar's full range can extend from **1** to **32767**. Set the **Min** property to the lowest value to be represented by the scrollbar. Set the **Max** property to the highest value to be represented by the scrollbar.

When the user eventually uses the scrollbar, the scrollbar arrows control small movements in the scrollbar's value determined by the **SmallChange** property. Clicking the empty part of the shaft on either side of the **scrollbox** produces a positive or negative change in the value represented by the **LargeChange** property. The user can drag the scrollbox itself to any position within the scrollbar shaft to jump to a specific location instead of changing the value gradually.

The figure below, shows an application that uses a vertical scrollbar to change the size of a label's font size. As the user clicks the top scrollbar arrow, the font size shrinks by the SmallChange value. As the user clicks the bottom scrollbar arrow, the font size increases by the SmallChange value. If the user clicks in the scrollbar's shaft on either side of the scrollbar's thumb, the LargeChange property value is either added to or subtracted from the font size.

Example.

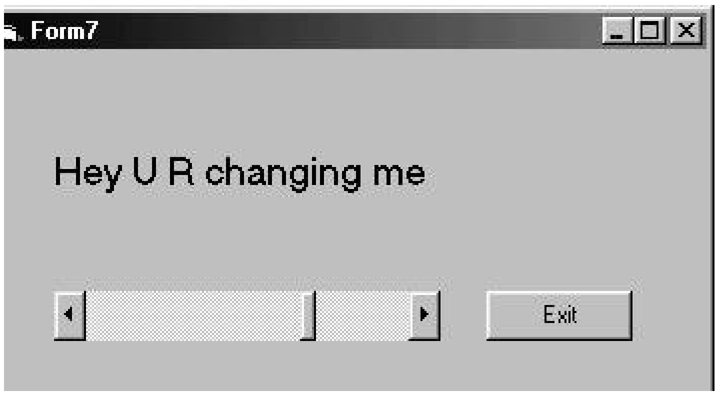
Drop a Horizontal scroll bar, a label and a command button.

Name the control and type the code in the respective event of the respective controls.

Set the Max property as 20.

On dragging the thumb of the scroll you can see the font size and caption of the label increasing as shown in the figure.

Demo for Horizontal Scroll Bar.



Output Scroll Bar Application .

Any time the user changes the scrollbar, the scrollbar's **change()** event procedure is executed.

```
Private Sub HScroll1_Change()

Label1.FontSize = HScroll1.Value
```

```
Label1.Caption = "Hey U R changing me"

End Sub

Private Sub CmdExit_Click()

End

End Sub
```

ListBox Control.

List Box Control displays a list of items from which the user can select the required one. By default, only one column is displayed but multiple columns can also be displayed. The properties of a List Box control are as follows:

Adding item in a list box.

At design time the items in a list box are added using the **ListProperty** from the properties of the List Box. Each new item in the List Property must be entered in a separate line. At runtime the items can be added using the **Additem( )** method.

The syntax for it is:

Object.Additemitem, index.

Where **item** is string that has to be added as a new item in the list box.

Index is an integer that indicates where the new item has to be placed.

Removing item from the list box.

To remove an item from the list box the **Removeitem( )** method is used.

The syntax is:

Object.Additem item, index.

List Index Property sets the index number of the currently selected item. The index number always starts with 0. The **List Index** returns –1 if no item is selected.

List Count property gives the total number of items in the list box.

New Index property gives the index number of the last item added to the list.

Multi Select allows the user to select multiple items from the list box.

It accepts three values:

- Multiple selections is not allowed (default state)
- To select a small range of items
- Extended Multi selection, that is, large range of items can be selected(press shift and click the mouse)

Selected:

This property is an array, similar to the List property, with elements that have a true or false, depending on the status of the corresponding list element.

Clear( ):

method removes all the items from the list box.

SelCount:

It is used to give number of selected items in the list box. It works only with Multiselect (1 and 2) properties.

Sorted:

It is false by default. If made true it will display the items entered in a sorted manner.

Example.

Drop a list box, a text box, six command buttons.

Name them as shown in the figure below .

The **Add** button adds the items entered in the text box to the list box.

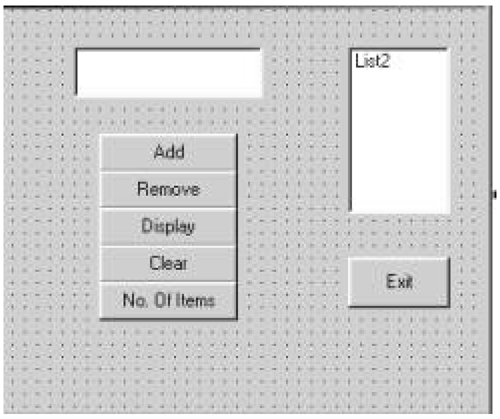
The **Remove** button removes the selected item from the listbox.

The **Display** button displays the selected item in the textbox.

The **Clear** button clears the listbox items.

The **No Of Items** button displays the number of items of the list box in the text box.

The **Exit** button is to exit from the application.

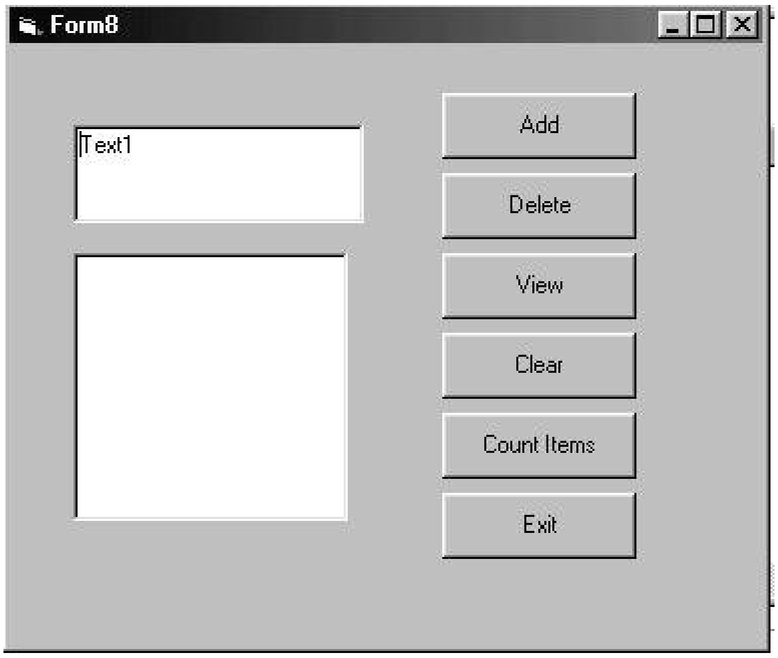


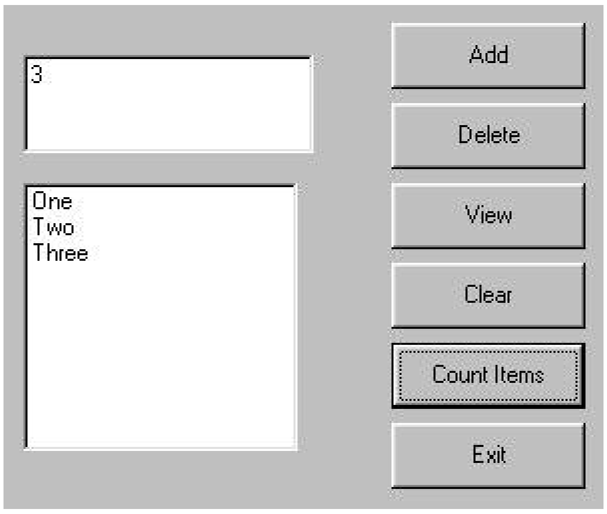
Design for listbox control.

Type the command in the respective command buttons and events as shown below.

```
Private Sub CmdAdd_Click()

List2.AddItem Text1.Text

End Sub

Private Sub CmdDelete_Click()

List2.RemoveItem List2.ListIndex

End Sub

Private Sub CmdView_Click()

Text1.Text = ""

Text1.Text = List2.List(List2.ListIndex)

End Sub

Private Sub CmdClear_Click()

List2.Clear

End Sub

Private Sub CmdCountItems_Click()

Text1.Text = List2.ListCount

End Sub

Private Sub CmdExit_Click()

End

End Sub
```

First Output screen on execution of the above code





After adding some values.

Combo Box Control .

It is an expandable list box. It is a collection of text box and list box properties. One can either type a text into combo box or select an item from the list. There are three types of combo boxes:

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

Dropdown Combo:

This is the default one. It appears only as 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. The user can either select a value or type a value in the edit area.

Simple Combo:

It displays an edit area always with an attached list box always visible immediately below the edit area. User can either select or type a value from the list.

Drop-down List Combo:

It turns the combo box into a drop-down list box. At run-time it looks like a drop-down combo. The user can select only one of the list items but can't type anything in the edit area. The difference between drop-down combo & drop-down list is that the edit area in the drop-down list is disabled.

Methods.

- Add Item: It is used to add values to the combo box.
- Remove Item: It is used to remove items from combo box.
- Clear: Both the text box and the list box are cleared.
- Locked: Default is false. Both list box and text box are locked When it is true.
- Sorted: Default value is false. The items in the list box are sorted out when sorted=true.

Example.

Drop four command buttons, a text box and a combo box.

The command button 'Add' adds the items entered in the text box to the combo box.

The 'Clear' button clears the combo box items and the text box item.

The 'No.of items' button displays the number of items of the combo box, in the text box.

Exit button is to terminate the application.

Type the code given below and run the application.

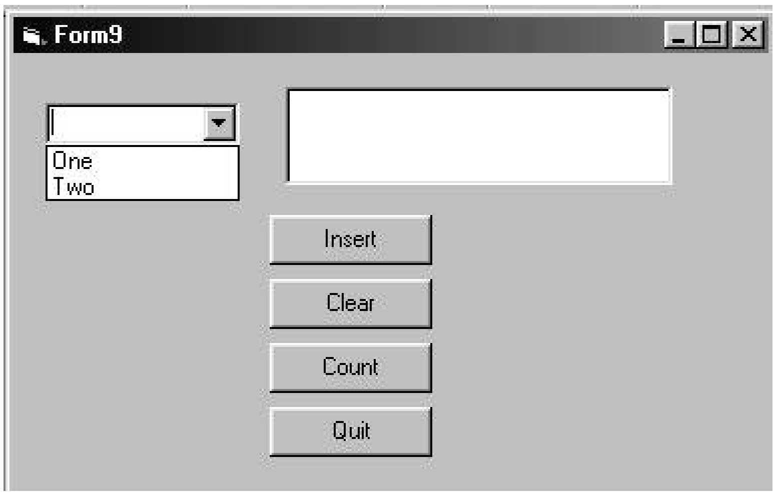
```
Private Sub CmdInsert_Click()

Combo1.AddItem Text1.Text

Text1.Text = ""

Text1.SetFocus

End Sub

Private Sub CmdClear_Click()

Text1.Text = ""

Combo1.Clear

End Sub

Private Sub CmdExit_Click()
```

```
End

End Sub

Private Sub CmdCount_Click()

Text1.Text = Combo1.ListCount

End Sub
```

The following output is shown.

Type a value in the text box. Click on the Insert button.

You can see the value being added to the combo list as shown in the figure below.



On execution of Combo list box application.

Picture Box Control .

It is used to display bitmap file (.bmp), icon files(.ico) and windows meta files. The picture at run-time can be changed using

picture1.picture = loadpicture (filepath)

Picture box is a container control as it can have any standard control such as text box, command button etc.

Align:

Using align property, a user can place a picture anywhere.

Autosize:

adjusts the picture automatically to the size of the control. It accepts two values: true or false. Default is false. One cannot enlarge a picture control as it is of standard size where as in an image control one can enlarge a picture using stretch property. Loading a picture in an image control is done as:

Image1.picture = loadpicture("filepath").

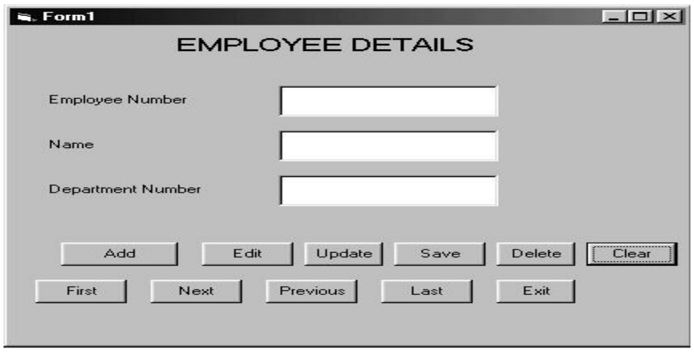
Here the picture can be controlled according to the size of the image control. Picture box is the only control that can be placed in MDI Form(directly), Data control and Timer Control.

Example.

The following example shows how to load a picture during runtime.

Drop a Picture and a command button control on the form as shown below:

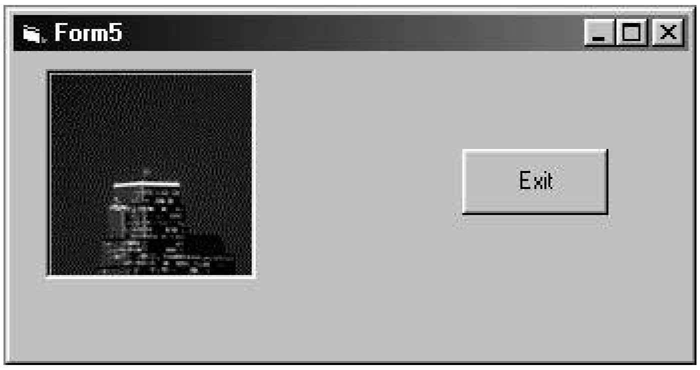
Name the control and type the code as given in the code window below.



Demo for the Picture Box control.

```
Private Sub CmdExit_Click()

End

End Sub

Private Sub Form_Load()

Picture1.Picture = LoadPicture("C:\Program Files\Microsoft          FrontPage\temp\city.gif")

 End Sub
```

On execution you will see the respective gif loaded into the control.



Output for the Picture Control .

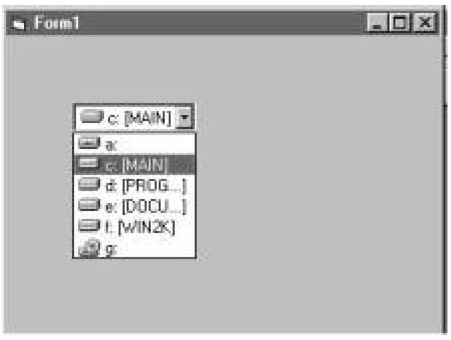
Image Box Control .

It is similar to the Picture Box Control. If the **Stretch** property is made true, then the image can be resized so that it can fill the area of the Image Box control. If it is stretch property is false then it behaves like Picture Box control with it's Autosize property set to true.

Drive List Box.

Use the Drive List Box control to let the user to select a disk drive. This control is smart enough to search the host computer and determine the drives –local and remote, floppy, hard and CD-ROM – that exist on each system. The control then displays these choices graphically when users open the drive list box.

A Drive List Box Example:

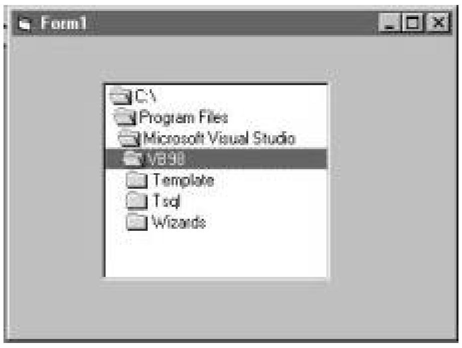


Directory List Box:

Use the directory list box control to let the user to select a directory folder. This control searches the host computer and determines the directories exist in the system. The directory list box will display these choices by using the standard window format..

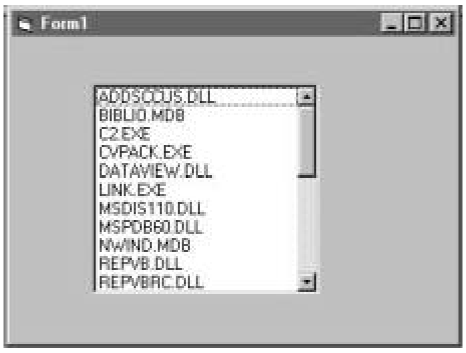
But the directory list box control can't determine the drive which is selected in the drive list box.

Directory List Example.



FileList Box.

Use the file list box control to let users select a file. The control searches the computer and determine the files that exist in the file system. The file list box then displays these choices using the standard window format.



```
ADDSCCUS.DLL
BIBLIO.MDB
C2.EXE
CVPACK.EXE
DATAVIEW.DLL
LINK.EXE
MSDIS110.DLL
MSPDB60.DLL
NWIND.MDB
REPVB.DLL
REPVBRC.DLL
```

A File List Box Example.

Example of File, Directory, Drive list Box .

The following program illustrates the use of the three list boxes and this program used to show the picture file.

Private Sub Dir1_Change().

File1.Path = Dir1.Path.

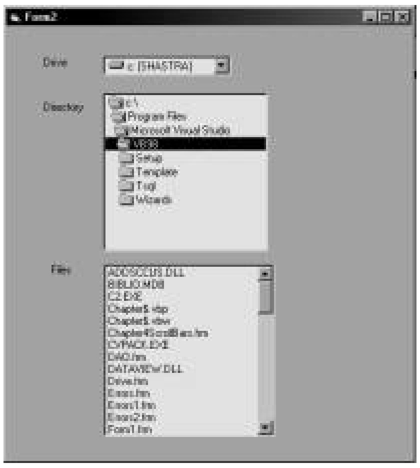
End Sub.

Private Sub Drive1_Change().

Dir1.Path = Drive1.Drive.

End Sub.

The output is as follows. When you change to another drive the directories in that drive are displayed. Similarly when you change to another directory the files in that directory are displayed.



```
c: [SHASTRA]

c:\
Program Files
Microsoft Visual Studio
VB98
Setup
Template
Tsql
Wizards

ADDSCCUS.DLL
BIBLIO.MDB
C2.EXE
Chapter5.vbp
Chapter5.vbw
Chapter4ScrollBars.frm
CVPACK.EXE
DAO.frm
DATAVIEW.DLL
Drive.frm
Errors.frm
Errors1.frm
Errors2.frm
Form1.frm
```

Other Controls

We still have to briefly discuss a few other controls in the Toolbox.

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.

The OLE Control

When OLE first made its appearance, the concept of Object Linking and Embedding seemed to most developers nothing short of magic. The ability to embed a Microsoft Word Document or a Microsoft Excel worksheet (see Figure 3-17) within another Windows application seemed an exciting one, and Microsoft promptly released the OLE control—then called the OLE Container control—to help Visual Basic support this capability.

In the long run, however, the *Embedding* term in OLE has lost much of its appeal and importance, and nowadays programmers are more concerned and thrilled about Automation, a subset of OLE that lets them control other Windows applications from the outside, manipulating their object hierarchies through OLE. For this reason, I won't describe the OLE control: It's a rather complex object, and a thorough description of its many properties, methods, and events (and quirks) would take too much space.

Active X Controls:

Visual basic 6.0 provides with a number of Custom Controls, some are data bound controls and some of them are Active X Controls. Since a majority of these controls happen to be Active X controls, these custom controls are also called as ***Active X* Controls**. To add the custom control into the form:

- Right click the toolbox.
- Select Components from the pop-up menu.

List of Active X Controls are:

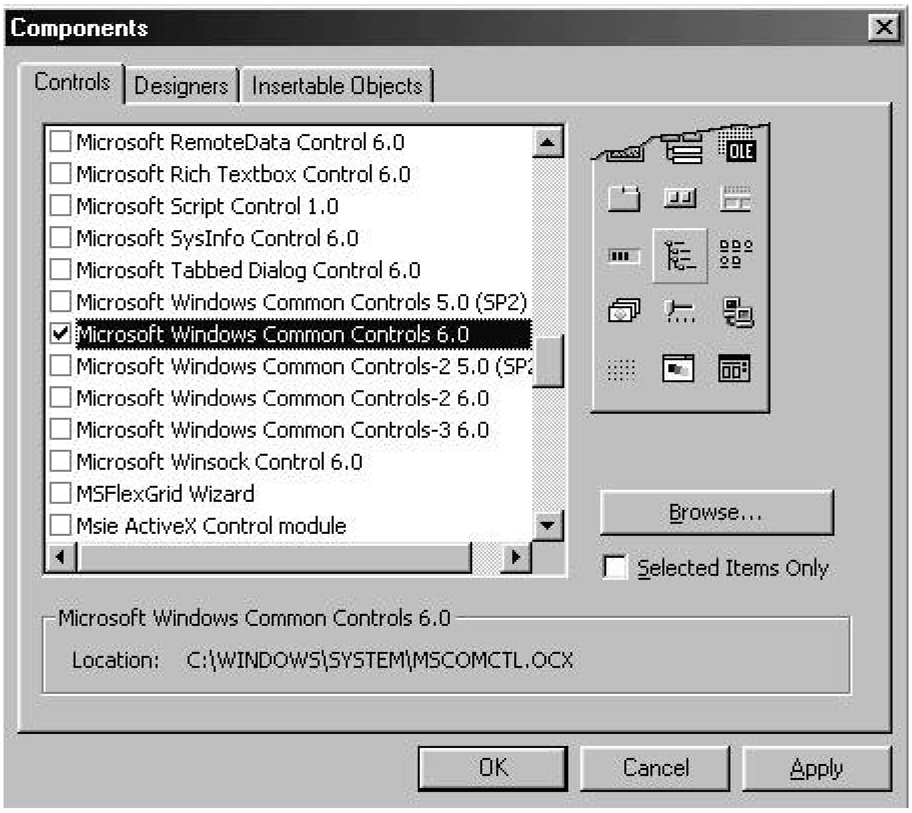
- Rich Text Box Control.
- Tabstrip control.
- Slider control.
- Progress Bar control.
- StatusBar Control.
- Toolbar control.
- Imagelist Control.
- ImageCombo Control.
- ListView Control.
- Treeview Control.

a check the box for

Microsoft Windows Common Controls x.x. (Where x is the

version of Visual Basic) as shown in the figure below and click on

the **Apply** button.



Components

Controls | Designers | Insertable Objects

- [ ] Microsoft RemoteData Control 6.0
- [ ] Microsoft Rich Textbox Control 6.0
- [ ] Microsoft Script Control 1.0
- [ ] Microsoft SysInfo Control 6.0
- [ ] Microsoft Tabbed Dialog Control 6.0
- [ ] Microsoft Windows Common Controls 5.0 (SP2)
- [x] Microsoft Windows Common Controls 6.0
- [ ] Microsoft Windows Common Controls-2 5.0 (SP2
- [ ] Microsoft Windows Common Controls-2 6.0
- [ ] Microsoft Windows Common Controls-3 6.0
- [ ] Microsoft Winsock Control 6.0
- [ ] MSFlexGrid Wizard
- [ ] Msie ActiveX Control module

Browse...

- [ ] Selected Items Only

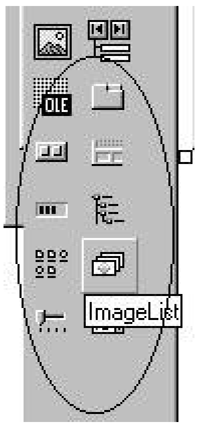
Microsoft Windows Common Controls 6.0

Location: C:\WINDOWS\SYSTEM\MSCOMCTL.OCX

OK | Cancel | Apply

Components window.

You will see a list of control added to your tool box as shown below.

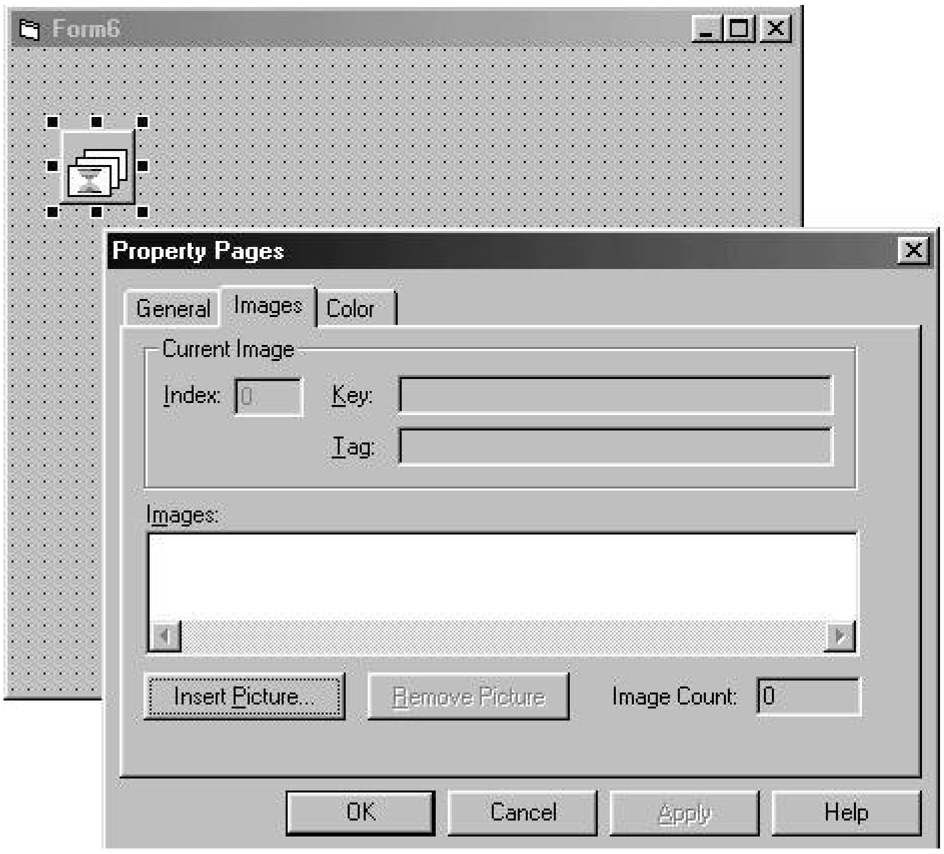


Tool box with more windows controls .

Adding Images to the Image List Control.

Select ImageList control in the tool box and add it to the form.

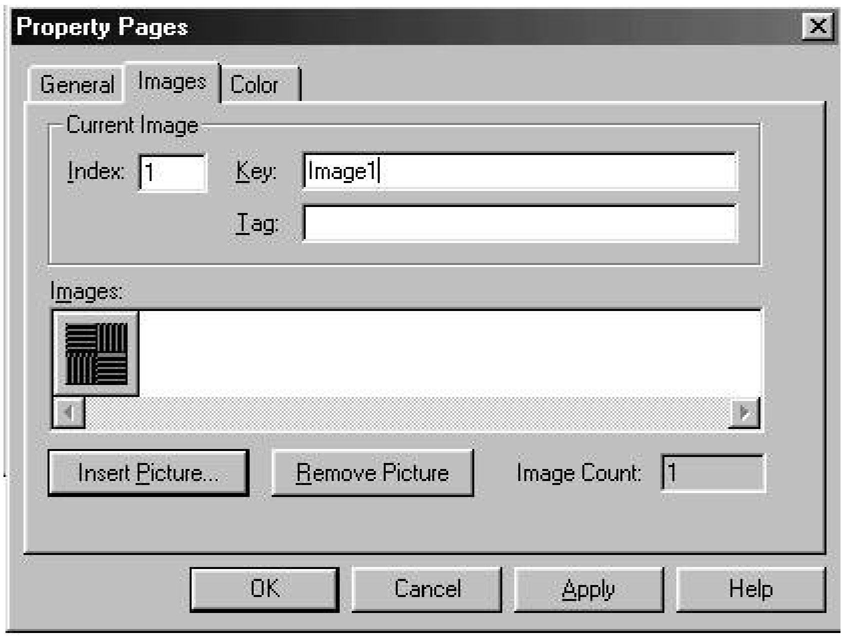
Right click on the **ImageList** control that is added to the form and select Properties. The Property Pages for the ImageList control will now appear as given in the figure below.



P roperty Pages window of ImageList control.

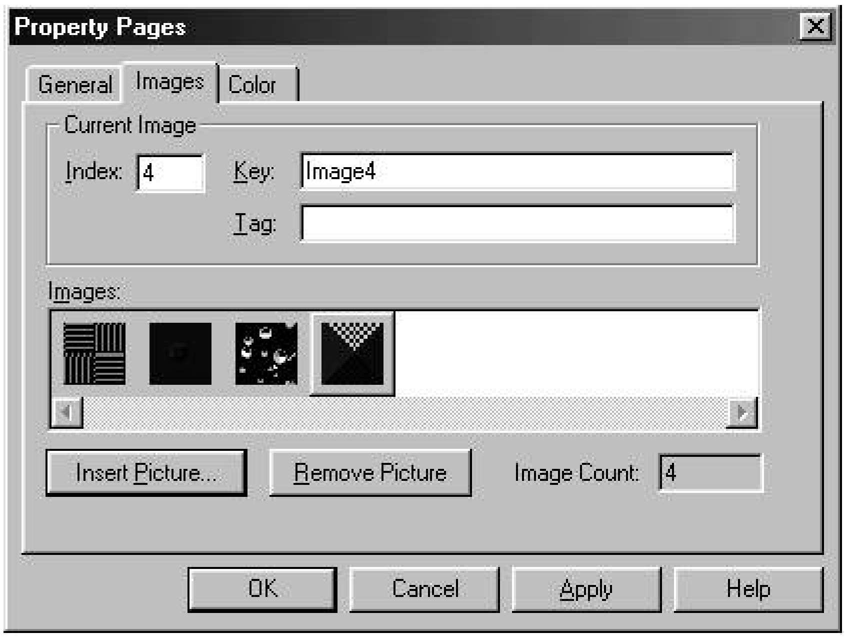
Insert the images by clicking on the Insert picture button and selecting the required image.

Give a key or name in the Key field to identify it as shown in the **figure** below. Similarly repeat for all the images and finally click on the **Apply** and then click on the **Ok** Button.



Inserting picture and Keys for ImageList control.

The Index for each image is automatically assigned as 1,2,3,4 and so on respectively for each image.



Property Page window.

When a Common Control is using the Images in the ImageList control (called being bound), user can only add images to the end. Also, once the ImageList is bound to another control, user cannot delete any images.

Using the images in the Image List Control.

- **Status Bar Control:**

A status bar control is a frame that consists of several panels, which inform the user about the status of an application. The control can hold upto 16 frames. It can be placed at the top or bottom or sides of an application.

- **Properties of Status Bar:**

- Style: The style of the status bar may be either Normal or simple.

- Mouse Pointer: The user can select type of mouse pointer from the given list.

- Panels: One can either insert/remove panels for a status bar and can also give key Value, tool tip text , tags , index. Index should always start with one.

- Alignment: The panel can be aligned to the left, center or right.

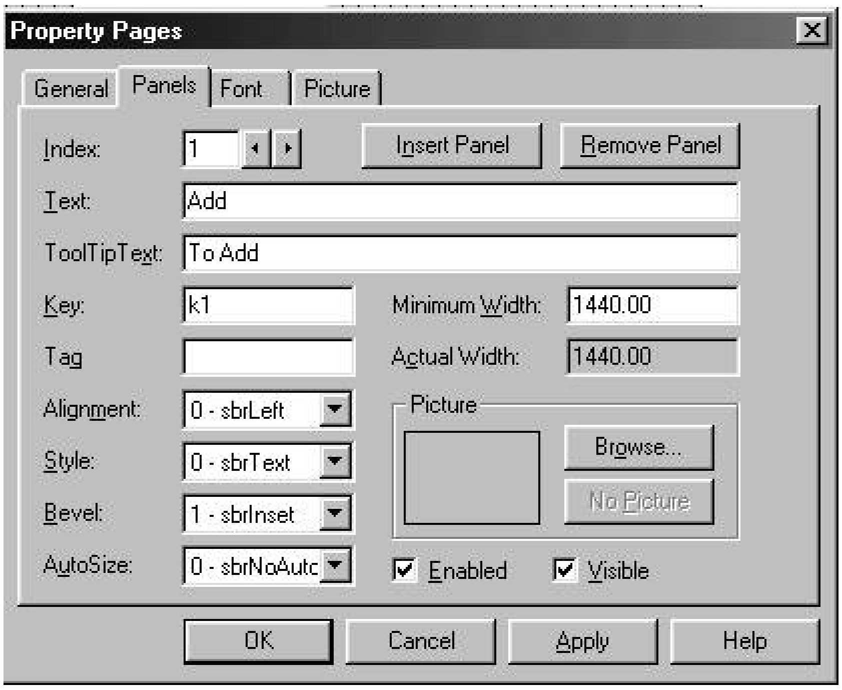
- Picture: We can also give pictures to panels.

Example:

The following dialog consists of 3 text boxes, a status bar with 3 panels.

Steps to insert panels in the status bar:

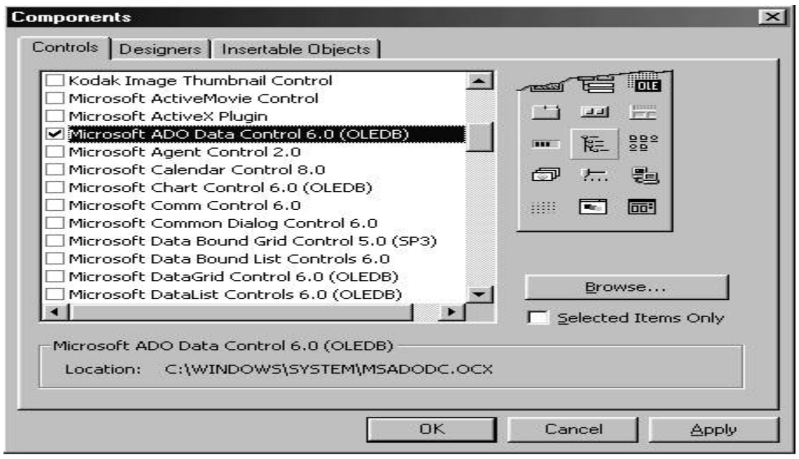
- Drop or place the status bar on the form, right click and select properties.
- Select Panels.
- Click on Insert panel.
- Assign key, text and tool tip for each panel after clicking the Insert panel button.
- Click Apply and then the Ok button.



Index: 1
Text: Add
ToolTipText: To Add
Key: k1
Minimum Width: 1440.00
Tag:
Actual Width: 1440.00
Alignment: 0 - sbrLeft
Style: 0 - sbrText
Bevel: 1 - sbrInset
AutoSize: 0 - sbrNoAutc
☑ Enabled ☑ Visible

Changing the properties of Status Bar Control.

Design the form as shown below.


Components

- [ ] Kodak Image Thumbnail Control
- [ ] Microsoft ActiveMovie Control
- [ ] Microsoft ActiveX Plugin
- [x] Microsoft ADO Data Control 6.0 (OLEDB)
- [ ] Microsoft Agent Control 2.0
- [ ] Microsoft Calendar Control 8.0
- [ ] Microsoft Chart Control 6.0 (OLEDB)
- [ ] Microsoft Comm Control 6.0
- [ ] Microsoft Common Dialog Control 6.0
- [ ] Microsoft Data Bound Grid Control 5.0 (SP3)
- [ ] Microsoft Data Bound List Controls 6.0
- [ ] Microsoft DataGrid Control 6.0 (OLEDB)
- [ ] Microsoft DataList Controls 6.0 (OLEDB)

Microsoft ADO Data Control 6.0 (OLEDB)
Location: C:\WINDOWS\SYSTEM\MSADODC.OCX

Demo for StatusbarControl.

Type the code in the respective controls andevents.

```
Private Sub Form_Load()

Text1.Text = " "

Text2.Text = " "

Text3.Text = " "

Text3.Locked = True

Text3.Enabled = False

End Sub

Private Sub StatusBar1_PanelClick(ByValPanel As MSComctlLib.Panel)

Select Case Panel

Case StatusBar1.Panels.Item(1)

Text3.Text = Val(Text1.Text) +Val(Text2.Text)

MsgBox "sum : "&Text3.Text

Text3.Text = ""

Case StatusBar1.Panels.Item(2)

If Val(Text1.Text)>Val(Text2.Text) Then

Text3.Text = Val(Text1.Text) -Val(Text2.Text)
```

```
 MsgBox "difference : "&Text3.Text

 Text3.Text = ""

ElseIf Val(Text2.Text)>Val(Text1.Text)Then

 Text3.Text = Val(Text2.Text) -Val(Text1.Text)

 MsgBox "difference : "&Text3.Text

 Text3.Text = ""

End If

 Case StatusBar1.Panels.Item(3)

 Text3.Text = Val(Text1.Text) *Val(Text2.Text)

MsgBox "product : "&Text3.Text

 Text3.Text = ""

End Select

End Sub
```

Variables and Data Types.

Programming involves data and ways of manipulating these data. Thus, one should learn how to represent data in the computer.

Variables, Types, Declaration and Scope.

One often needs to store values temporarily when performing calculations with Visual Basic. In Visual Basic, the values can be stored in containers called **variables**. Generally, variables have a name and a data type (which determines the range of values the variable can store and the operations defined for that type).

Data types control the internal storage of data in Visual Basic. Visual Basic automatically associates a type to a variable. By default, Visual Basic uses the Variant data type. But programmers can always associate variables to a type explicitly

The following table enumerates the common VB data types:

Data Type	Description and Range.
Boolean	Data that is either True or False.
Byte	Positive numeric values without decimals that range from 0 to 255.
Double	Numeric values that range from -1.79769313486232E+308 to 1.79769313486232E+308*.
Integer	Numeric values with no decimal point or fraction that range from –32,768 to 32,767.
Long	Integer values with a range beyond that of Integer data values.It ranges from –2,147,483,648 to 2,147,483,647.
String	Data that consists of 0 to 65,400 characters of alphanumeric (letters and numbers) and special characters (punctuations, etc.).
Variant	Data of any data type and used for control and other values for which the data type is unknown.

***The value of 1.2E+6 is computed by multiplying 1.2 by 6 raise to 10.**

This is equal to 1.2 times 1,000,000 = 1,200,000.

Variable Declaration.

Data have different sizes, thus a variable that would contain a *String* value should have enough space to accommodate strings. Variable declaration specifies how much space VB should allocate for a variable.

A variable is declared using the **Dim** statement through the following syntax:

Dim *<var_name>* **As** *<data_type>*

A variable name must begin with an alphabet letter and should not exceed 255 characters. It should not contain special characters such as %, &, !, #, @ and $.

Most of the programming language variables must be declared or defined in advance for the compiler. The reason for doing this has been to help the compiler to process the application faster.

Explicit Declaration.

Generally speaking, when declaring a variable, information is given to reserve some space in memory. Automatically, whenever Visual Basic finds a new variable, it assigns default variable type and value.

```
    Syntax : Dim Variable [as type]
Example: Dim a as string

    Dim count as integer
```

When Visual Basic finds a **Dim** statement, it creates one or more new variables as specified in the statement. Then it creates a placeholder by reserving some space in the memory and assigning a name to it. When the following statement:

`A= "This is sample program"` is used.

Visual Basic places the value "This is sample program" in the placeholder for the "A" variable. When the program asks for the value of this variable, Visual Basic reads it from the same area of memory.

Implicit Declaration.

It is not necessary to declare the variable before using it. Whenever the Visual Basic finds a new variable, it assigns a default variable type and value to it. Using a new variable in the code is equivalent to declaring it without type. Visual basic adjusts its type according to the value assigned to it. For example, consider a declaration as follows:

```
Dim A,Dim B
```

And then assign the text value to one and numeric value to the other.

```
A="This is sample program"
B=50.45
```

The variable 'A' is a string variable and 'B' is a numeric variable.

Using Option Explicit Statement.

The main usage of declaring the variables by using **Option Explicit** is, it checks in the module for usage of any undeclared variables and reports an error to the user. The **Option Explicit** statement must be included in every module in which the variable declaration is required.

Variables Scope.

The scope of the variables is with respect to the location where they are declared.

Local variables.

These are declared inside a procedure. These variables are available to the code inside the procedure and can be declared using the DIM statement. When the variable's scope is limited to a procedure, it is called ***local variable***.

```
Dim int c as Integer
```

The local variable's life span is short. The value of the variable is lost while exiting from the procedure and the memory used by these variables are released and can be reclaimed.

Static Variables.

Static variables are not reinitialized each time. Visual Basic invokes procedure and retains or preserves the value even after executing the procedure. For Example, with the use of static variables, one can keep track of the number of times a command button is clicked.

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

Module Level Variables.

These variables are available to all the procedures in the module. They are declared using the ***Public*** or the ***Private*** keyword.

```
     Public cnt as integer
Private cnt as integer
```

Declaring the variable by using **Public** keyword makes it available throughout the application even for other modules. **Private** keyword makes it available only to that module. At the module level there is no difference between Dim and Private, but private is preferred because it gives the code more readability.

Public vs Local Variables.

The variables can have the same name and different scope. For instance, it is possible to have a public variable named '**G'** and local variable named **'G'.** But referencing to the value of **G** within the procedure would access the local variable and references to **G** outside the procedure would access the public variable.

Operators.

Assignment Operator.

The assignment operator **=** is used to assign data to a variable. Assigning data to a variable is simply storing a value to the variable. The syntax is as follows:

<var_name> = <expression>

<var_name> is a variable name (which is declared using the *Dim* statement). *<expression>* can be a literal, another variable, a mathematical expression, or a function call. Take the following examples:

```
'Assigning a literal to a variable

age = 18
stude_name = "Jay Fernandez"

'Assigning the value of a variable to another variable
grade =score

'Assigning the result of a mathematical expression
grade =score1 + score2
```

Always bear in mind that the left hand and right hand side values of the assignment operator should be of the same data type or at least of compatible type. One cannot assign a *String* value to a *Double* variable.

Math Operators.

Math operators are used to manipulate data. To get the average of 2 numbers, probably one can use the formula *(x + y)/2*. The following table describes VB's primary math operators.

Operator.	**Example.**	**Description.**
+	Score1 + Score2	Adds two values
-	Price – Discount	Subtracts the second value from the first
*	Price * Discount	Multiplies two values
/	Total / 2	Divides the first value by the second value
^	2^2	Raises the first value to the second value
-	-2	Negates a value. This operator is called *unary minus*.
Mod	10 Mod 2	Divides two numbers and returns only the remainder.
\	Total\2	\ denotes Integer Division.

\ Operator is used to divide two numbers and return an integer result. Before integer division is performed, the operands are rounded to Integer or Long expressions. Any fractional portion is truncated. If any operand in Null result is also Null. If it is Empty then it is treated as 0. Numerator should not be 0.

Several mathematical expressions can be combined in one expression as in the example:

grade =(score1 + score2 + score3) / 3.

Conditional Operators.

To control the VB program flow, we can use various conditional operators. Basically, they resemble mathematical operators. Conditional operators are very powerful tools, they let the VB program compare data values and then decide what action to take, whether to execute a program or terminate the program and etc. These operators are shown in Table.

Operator	Meaning
=	Equal to
>	More than
<	Less Than
>=	More than and equal
<=	Less than and equal
<>	Not Equal to

You can also compare strings with the above operators. However, there are certain rules to follows: Upper case letters are less than lowercase letters, "A"<"B"<"C"<"D".......<"Z" and number are less than letters.

Logical Operators.

In addition to conditional operators, there are a few logical operators, which offer added power to the VB programs. They are shown in Table.

Operator	Meaning
And	Both sides must be true
or	One side or other must be true
Xor	One side or other must be true but not both
Not	Negates truth

Concatenation Operator &.

& is used to force string concatenation of two operands.

Example.

Result = operand1 & operand2

Operator Precedence.

Another matter that one should take note of is operator precedence. Precedence is the order of computation. ^ is evaluated first, then * and /, then + and -. If there are * and / in an expression, execution is done from left to right. This holds true to + and -. Let us demonstrate this in the following examples:

The expression 2 + 3 * 4 is equal to 14. This was computed by doing the multiplication first before addition (since multiplication has higher precedence over addition).

The expression 4 * 3 / 2 is equal to 6. We evaluate this expression from left to right, which is by doing the multiplication first before division.

You can override the operator precedence by using parentheses. VB always evaluates expressions enclosed in parentheses before anything else in the expression. Consider the following expression:

4 * (2 +3)

This is equal to 20, computed by adding 2 and 3 and multiplying the sum by 4. Without the parentheses, the value of the entire expression is 11 (multiplying 4 with 2 and adding 3 to the product).

Operators are evaluated in the order of precedence as shown below:

Arithmetic Operators.

Exponentiation. ^

Negation.	-
Multiplication and Division.	*, /
Integer Division.	\
Modulo Arithmetic.	Mod
Addition & Subtraction.	+, -

Comparison Operator.

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

Logical operators.

Not.	Not
And.	And
Or.	Or
Xor.	Xor

Converting Data Types .

Conversion Function.	**Converts an expression to.**
Cbool	Boolean
Cbyte	Byte
Ccur	Currency
Cdate	Date
CDbl	Double
Cint	Integer
CLng	Long
CSng	Long
CStr	String
Cvar	Variant

Conversion Table.

Cbyte.

Converts an expression to a Byte. The syntax is:

CByte(expression).

The expression argument is any valid numeric or string expression.

In general, we can document our code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, we can use CByte to force byte arithmetic in cases where currency, single precision, double-precision, or integer arithmetic normally would occur.

This example below uses the CByte fimction to convert an expression to a Byte.

```
MyDouble = 125.5678 ' MyDouble is a Double.
MyByte = CByte(MyDouble) ' MyByte contains 126.
```

Cbool.

The Cbool function converts an expression to a Boolean. The syntax of this function is

CBool(expression).

The expression argument is any valid numeric or string expression. If expression is zero. False is returned; otherwise, True is returned. If expression can not be interpreted as a numeric value, a run-time error occurs.

This example uses the CBool function to convert an expression to a Boolean. If the expression evaluates to a nonzero value, CBool returns True; otherwise, it returns False.

```
A = 5: B == 5            'Initialise variables.
Check = CBool(A = B) ' Check contains True. A
= 0                         ' Define variable.
Check = CBool(A)         ' Check contains False.
```

Ccur.

Ccur function converts an expression to a Currency. The Syntax is as follows

CCur(expression).

The expression argument is any valid numeric or string expression.

In general, we can document our code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, we can use CCur to force currency arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

An example below uses the CCur function to convert an expression to a Currency.

```
    MyDouble = 543.214588            ' MyDouble is a Double.
    MyCurr = CCur(MyDouble * 2)        ' Convert result of    MyDouble *
2
                                     ' (1086.429176) to a
                                     ' Currency (1086.4292).
```

Cdate .

Cdate function converts an expression to a Date. The syntax is:

CDate(date) The date argument is any valid date expression.

We use the **IsDate** function to determine if date can be converted to a date or time. CDate recognizes date and time literals as well as some numbers that fall within the range of acceptable dates. When converting a number to a date, the whole number portion is converted to a date. Any fractional part of the number is converted to a time of day, starting at midnight.

CDate recognizes date formats according to the locale setting of our system. The correct order of day, month, and year may not be determined if it is provided in a format other than one of the recognized date settings. In addition, a long date format is not recognized if it also contains the day-of-the-week string.

An example below uses the CDate function to convert a string to a Date. In general, hard coding dates and times as strings (as shown in this example) is not recommended. Use date and time literals (such as #2/12/1969#, #4:45:23 PM#) instead.

```
MyDate = "February 12,1969"       ‘Define date.
MyShortDate = CDate(MyDate)         ' Convert to Date data type.
My Time = "4:35:47 PM"              ' Define time.
MyShortTime = CDate(MyTime)     ' Convert to Date data type.
```

CDbl.

CDbl function converts an expression to a Double. The syntax is :

CDbl(expression).

The expression argument is any valid Numeric or string expression.

In general, we can document our code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, we can use CDbl or CSng to force double- or single-precision arithmetic in cases where currency or integer arithmetic normally would occur.

We should use the CDbl function instead of Val to provide internationally aware conversions from any other data type to a Double. For example, different decimal separators and thousands separators are properly recognized depending on the locale setting of our system.

An example uses the CDbl function to convert an expression to a Double.

```
MyCurr = C Cur (234.456784)                          ' MyCurr is a Currency.
MyDouble = CDbl(MyCurr * 8.2 / 0.01)          ' Convert result to a Double.
```

Cint.

Cint function converts an expression to an Integer. The syntax is :

CInt(expression).

The expression argument is any valid numeric or string expression.

In general, we can document our code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use Cint or CLng to force integer arithmetic in cases where currency, single-precision, or double-precision arithmetic normally would occur.

We should use the Cint function instead of Val to provide internationally-aware conversions from any other data type to an Integer. For example, different decimal separators are properly recognised depending on the locale setting of our system, as are different thousand separators. If expression lies outside the acceptable range for the Integer data type, an error occurs.

This example uses the Cint function to convert a value to an Integer.

```
MyDouble = 2345.5678    ' MyDouble is a Double. Mylnt
= Clnt(MyDouble) ' Mylnt contains 2346.
```

CLng.

0

CLng converts an expression to a Long. The syntax is given below :

CLng(expression)

The expression argument is any valid numeric or string expression.

In general, we can document our code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use CInt or CLng to force integer arithmetic in cases where currency, single-precision, or double-precision arithmetic normally would occur.

This example uses the CLng function to convert a value to a Long.

```
My Val1 = 25427.45:    MyVal2 = 25427.55 ' My Vail, MyVal2 are Doubles.
MyLongI = CLng(MyVall)                    ' MyLongI contains 25427.
MyLong2 = CLng(MyVaI2)                    ' MyLong2 contains 25428.
```

CSng.

CSng converts an expression to a Single. The syntax is as given below :

CSng(expression)

The expression argument is any valid numeric or string expression.

In general, we can document our code using the data type conversion functions to show that the result o f some operation should be expressed as a particular data type rather than the default data type . For example, we should use CDbl or CSng to force double- or single-precision arithmetic in cases where currency or integer arithmetic normally would occur.

The example below uses the CSng function to convert a value to a Single.

```
' MyDoublel, MyDouble2 are Doubles.

MyDoublel == 75.3421115:
MyDouble2 = 75.3421555
MySinglel = CSng(MyDoublel)          ' MySinglel contains 75.34211.
MySingle2 = CSng(MyDouble2)          ' MySingle2 contains 75.34216.
```

CStr.

CStr function converts an expression to a String. The syntax is :

CStr(expression).

The expression argument is any valid numeric or string expression.

In ge neral, we can document our code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, we can use CStr to force the result to be expressed as a String.

We should use the CStr function instead of Str to provide internationally-aware conversions from any other data type to a String.

The example given below uses the CStr function to convert a numeric value to a String.

```
MyDouble = 437.324           ' MyDouble is a Double.
MyString = CStr(MyDouble)     ' MyString contains "437.324".
```

Cvar.

Cvar function converts an expression to a Variant. The syntax is

CVar (expression).

The expression argument is any valid numeric or string expression.

In general, we can document our code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, we can use CVar to force the result to be expressed as a Variant.

The example given below uses the CVar function to convert an expression to a Variant.

```
MyInt = 4534                  ‛ MyInt is an Integer.
MyVar = CVar(MyInt & "OOO") ' MyVar contains the string 4534000.
```

Constants.

Constant refers to the value that will not be changed during the execution of a program. For instance, if the program does math calculation, the value of ***pi*** may appear many times in the code. Such values are best represented by constants, instead of typing the values over and over again. A constant ***pi*** can defined and used in the code.

Once a constant has been declared, its value cannot be changed in subsequent statements. Usually the constants are processed faster than variables, because the compiler automatically substitutes constant names with their values.

Declaring the constant variable is similar to the declaring the variables, except that in addition to the constant variable name, a value must also be mentioned.

Const myarray[as type]=value
Public const pi as double=3.141592.

Visual basic uses constants extensively to define the various arguments of its methods and the settings of the various control properties.

Built-in Functions.

Visual Basic provides many useful built-in functions. By using the built-in functions you will not have to spend a lot of time writing your own code to perform a particular task.

For example, the user need not write code for performing square root of a number because Visual Basic supplies a built-in Square root function for this.

Some of these functions are explained below:

Abs	Absolute value of a number.
Cos	Cosine of an angle.
Sin	Sine of an angle.
Sqr	Square root of a number.
Str	Number converted to string.
Val	Numeric value of a string.
InputBox	Text entered in a dialog box by
MsgBox	Text displayed in a dialog box.

Abs() function.

This function returns absolute value. The absolute value is the positive equivalent of the number. This function is generally used distance calculation and weight because such values must always be positive.

For example,

Abs(-25) = 25

Cos() function.

This function returns a cosine of an angle expressed in radians in double data type.

Syntax:

Cos(number)

The required number argument is a Double or any valid numeric expression that expresses an angle in radians.

To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

Sin() function.

This function returns a sine of an angle expressed in radians in double data type.

Syntax:

Sin(number)

The required number argument is a Double or any valid numeric

expression that expresses an angle in radians.

To convert degrees to radians, multiply degrees by pi/ISO. To convert radians to degrees, multiply radians by 180/pi.

Sqr() function.

This function returns the square root of a positive number. If the number is negative, the Sqr () function causes an error because by definition the square root of a negative number is undefined.

Syntax:
    Sqr(positive number)

Example:
    Sqr(9) = 3

Str() function.

Converts a number to a string with the numeric digits in the string.
Syntax:
    Str(number)

When numbers are converted to strings, a leading space is always reserved for the sign of number. If number is positive, the returned string contains a leading space and the plus sign is implied.

For example, when a number is converted to a string, a leading space is always reserved for its sign.

```
Dim MyString
MyString=Str(1259)              'Returns I259".
MyString = Str(-1259.65)   ' Returns "-     1259.65".
MyString = Str(1259.001)  'Returns " 1259.001".
```

Val() function.

This function returns the number of a string made up of digits. The Val() function starts reading the string from the left and stops when it reaches a character that is not part of a number**.**

Syntax:
    Val("string")

For example,
    x = "12-5-98"

    val(x) = 12

InputBox() function.

Visual Basic provides this function to accept text from the user.

This function displays a dialog box with a prompt and a text box control

and waits for the user to enter some text. After entering the text, you can press OK or Cancel button.

Syntax:

```
InputBox(Prompt, [Title], (Default])
```

Prompt	The prompt, which you want to have in the dialog box.
Title	This argument is optional. This is the title of the dialog box. If you omit this argument, the project name is displayed as the title.
Default	The default input. If you anticipate the user's response, use this argument t display it when the dialog box is first opened.

The simplest format of the InputBoxQ function is as follows:

```
X == InputBox(Prompt)
```

Here X is a variable. The value entered by the user is assigned to the variable X. The return value of this function is always a string, even if the user enters number information.

For example, the prompt in the InputBox is "Enter a number", title of the InputBox is 'Use Response" and default value is '0'. The code should be written in the following manner:

```
X = InputBox("Enter a umber", "User Response", "O")
```

As a result of the above code, the InputBox will be displayed as shown in Figure.

1. **MsgBox() function.**

The function displays a dialog box with a message and waits for the user to close it by clicking on a button.

Syntax:

```
MsgBox Prompt, [buttons], [Title]
```

The message can be anything that user wants to have in this dialog box. The simplest form of the MsgBoxQ function is as follows:

2. **MsgBox Prompt.**

For example, if you want only the message "Do you want to continue ?" to be displayed in the dialog box, write the code given below:

```
MsgBox "Do you want to continue ?"
```

As a result of the above code, the MsgBox will display as shown in Figure.

The function displays the message in the dialog box that has OK button. Since you have not specified any title to your dialog box, the project name is taken as the title of your dialog box.

Suppose you want to have two buttons in the dialog box, one button for 'Yes' and another button for 'No' and the title is "Continue". Write the following code:

MsgBox "Do you want to continue ?", VbYesNo, "Continue"

You will see the 'Yes', 'No' buttons because of the value of the Buttons 'VbYesNo'. The MsgBox() function can display other buttons in the dialog box based on the following values.

Button Values	**Description**
VbOkOnly	Displays OK button only
VbOKCancel	Displays OK and Cancel buttons
VbAbortRetryIgnore	Displays Abort, Retry and Ignore buttons
VbYesNoCancel	Displays Yes, No and Cancel buttons
VbYesNo	Displays Yes and No buttons
VbRetryCancel	Displays Retry and Cancel buttons

Button values and Description.

Variable Declaration.

Data have different sizes, thus a variable that would contain a *String* value should have enough space to accommodate strings. Variable declaration specifies how much space VB should allocate for a variable.

A variable is declared using the **Dim** statement through the following syntax:

Dim *<var_name>* **As** *<data_type>*

A variable name must begin with an alphabet letter and should not exceed 255 characters. It should not contain special characters such as %, &, !, #, @ and $.

Most of the programming language variables must be declared or defined in advance for the compiler. The reason for doing this has been to help the compiler to process the application faster.

Explicit Declaration.

Generally speaking, when declaring a variable, information is given to reserve some space in memory. Automatically, whenever Visual Basic finds a new variable, it assigns default variable type and value.

```
       Syntax : Dim Variable [as type]

Example: Dim a as string
       Dim count as integer
```

When Visual Basic finds a **Dim** statement, it creates one or more new variables as specified in the statement. Then it creates a placeholder by reserving some space in the memory and assigning a name to it. When the following statement:

```
        A= "This is sample program"
```
is used.

Visual Basic places the value "This is sample program" in the placeholder for the "A" variable. When the program asks for the value of this variable, Visual Basic reads it from the same area of memory.

Implicit Declaration.

It is not necessary to declare the variable before using it. Whenever the Visual Basic find a new variable, it assigns default variable type and value. Using a new variable in the code is equivalent to declaring it without type. Visual basic adjusts its type according to the value assigned to it. For example, consider a declaration as follows:

```
Dim A,Dim B
```

And then assign the text value to one and numeric value to the other.

```
A="This is sample program"
B=50.45
```

The variable 'A' is a string variable and 'B' is a numeric variable.

Using Option Explicit Statement.

The main usage of declaring the variables by using **Option Explicit** is, it checks in the module for usage of any undeclared variables and reports an error to the user. The **Option Explicit** statement must be included in every module in which the variable declaration is required.

Variables Scope.

The scope of the variables is with respect to the location where they are declared.

Local variables.

These are declared inside a procedure. These variables are available to the code inside the procedure and can be declared using the DIM statement. When the variable's scope is limited to a procedure, it is called ***local variable***.

```
Dim int c as Integer
```

The local variable's life span is short. The value of the variable is lost when the procedure is being exited and the memory used by these variables is freed and can be reclaimed.

Static Variables.

Static variables are not reinitialized each time. Visual Basic invokes procedure and retains or preserves the value even after executing the procedure. For Example, with the use of static variables, one can keep track of the number of times a command button is clicked.

```
Private Sub Command1_Click()

Static cnt As Integer
cnt = cnt + 1
Print cnt

End Sub
```

Module Level Variables.

These variables are available to all the procedures in the module. They are declared using the ***Public*** or the ***Private*** keyword.

```
      Public cnt as integer.
Private cnt as integer.
```

Declaring the variable by using **Public** keyword makes it available throughout the application even for other modules. **Private** keyword makes it available only to that module. At the module level there is no difference between Dim and Private, but private is preferred because it gives the code more readability.

Public vs Local Variables.

The variables can have the same name and different scope. For instance, it is possible to have a public variable named **'G'** and local variable named **'G'.** But referencing to the value of **G** within the procedure would access the local variable and references to **G** outside the procedure would access the public variable.

1. Controlling Program Flow.

1. If...Then...Else

To effectively control the VB program flow **If...Then...Else** statement is used together with the conditional operators and logical operators.

The general format for the if...then...else statement is

```
If  conditions Then
      VB expressions
Else
      VB expressions
End If
```

any If..Then..Else statement must end with End If. Sometime it is not necessary to use Else.

Example:

```
Private Sub OK_Click()
      firstnum = Val(usernum1.Text)
      secondnum = Val(usernum2.Text)
      total = Val(sum.Text)
      If total = firstnum + secondnum And Val(sum.Text) <> 0 Then
      correct.Visible = True
      wrong.Visible = False
```

```
            Else
            correct.Visible = False
            wrong.Visible = True
            End If
    End Sub
```

Control Flow

All programming languages must provide one or more ways to execute some statements out of the sequence in which they appear in the program listing. Apart from calls to Sub and Function procedures, you can gather all the basic control flow statements in two groups: *branch* statements and *loop* statements.

Branch Statements

The main branch statement is the *If...Else...Else If...End If* block. Visual Basic supports several flavors of this statement, including single-line and multiline versions:

```
' Single line version, without Else clause
If x > 0 Then y = x
' Single line version, with Else clause
If x > 0 Then y = x Else y = 0
' Single line, but with multiple statements separated by colons
If x > 0 Then y = x: x = 0 Else y = 0

' Multiline version of the above code (more readable)
If x > 0 Then
    y = x
    x = 0
Else
    y = 0
End If

' An example of If..ElseIf..Else block
If x > 0 Then
    y = x
ElseIf x < 0 Then
    y = x * x
Else                ' X is surely 0, no need to actually test it.
    x = -1
End If
```

You should be aware that any nonzero value after the *If* keyword is considered to be True and therefore fires the execution of the *Then* block:

```
' The following lines are equivalent.
If value <> 0 Then Print "Non Zero"
If value Then Print "Non Zero"
```

Even if this latter notation lets you save some typing, you shouldn't believe that it also makes your program faster, at least not necessarily. Benchmarks show that if the variable being tested is of type Boolean, Integer, or Long, this shortened notation doesn't make your program run faster. With other numeric types, however, you can expect some modest speed increment, about 20 percent or less. If you feel comfortable with this technique, go ahead and use it, but be aware that in many cases the speed improvement isn't worth the decreased readability.

Many advanced optimization techniques become possible when you combine multiple conditions using AND and OR operators. The following examples show how you can often write more concise and efficient code by rewriting a Boolean expression:

```
' If two numbers are both zero, you can apply the OR operator
' to their bits and you still have zero.
If x = 0 And y = 0 Then ...
If (x Or y) = 0 Then ...

' If either value is <>0, you can apply the OR operator
' to their bits and you surely have a nonzero value.
If x <> 0 Or y <> 0 Then ...
If (x Or y) Then ...

' If two integer numbers have opposite signs, applying the XOR
' operator to them yields a result that has the sign
'  bit set. (In other words, it is a negative value.)
If (x < 0 And y >= 0) Or (x >= 0 And y < 0) Then ...
If (x Xor y) < 0 Then ...
```

It's easy to get carried away when you're working with Boolean operators and inadvertently introduce subtle bugs into your code. For example, you might believe that the following two lines of code are equivalent, but they aren't. (To understand why, just think how numbers are represented in binary.)

```
' Not equivalent: just try with x=3 and y=4, whose binary
' representations are 0011 and 0100 respectively.
If x <> 0 And y <> 0 Then ...
If (x And y) Then ...
' Anyway, you can partially optimize the first line as follows:
If (x <> 0) And y Then ...
```

Another frequent source of ambiguity is the NOT operator, which toggles all the bits in a number. In Visual Basic, this operator returns False only if its argument is True (-1), so you should never use it with anything except the Boolean result of a comparison or with a Boolean variable:

```
If Not (x = y) Then ...   ' The same as x<>y
If Not x Then ...         ' The same as x<>-1, don't use instead of x=0
```

One detail that surprises many programmers coming to Visual Basic from other languages is that the *If* statement doesn't support the so-called *short-circuit evaluation*. In other words, Visual Basic always evaluates the whole expression in the *If* clause, even if it has enough information to determine that it is False or True, as in the following code:

```
' If x<=0, it makes no sense to evaluate Sqr(y)>x
' because the entire expression is guaranteed to be False.
If x > 0 And Sqr(y) < z Then z = 0

' If x=0, it makes no sense to evaluate x*y>100.
' because the entire expression is guaranteed to be True.
If x = 0 Or x * y > 100 Then z = 0
```

Even though Visual Basic isn't smart enough to optimize the expression automatically, it doesn't mean that you can't do it manually. You can rewrite the first *If* statement above as follows:

```
If x > 0 Then If Sqr(y) < z Then z = 0
```

You can rewrite the second *If* statement above as follows:

```
If x = 0 Then
    z = 0
ElseIf x * y > 100 Then
    z = 0
End If
```

The *Select Case* statement is less versatile than the *If* block in that it can test only one expression against a list of values:

```
Select Case Mid$(Text, i, 1)
    Case "0" To "9"
        ' It's a digit.
    Case "A" To "Z", "a" To "z"
        ' It's a letter.
    Case ".", ",", " ", ";", ":", "?"
        ' It's a punctuation symbol or a space.
    Case Else
        ' It's something else.
End Select
```

The most effective optimization technique with the *Select Case* block is to move the most frequent cases toward the top of the block. For instance, in the previous example you might decide to test whether the character is a letter before testing whether it's a digit. This change will slightly speed up your code if you're scanning regular text that's expected to contain more words than numbers.

Surprisingly, the *Select Case* block has an interesting feature that's missing in the more flexible *If* statement—namely, the ability to perform short circuit evaluation, sort of. In fact, *Case* subexpressions are evaluated only until they return True, after which all the remaining expressions on the same line are skipped. For example, in the *Case* clause that tests for punctuation symbols in the preceding code snippet, if the character is a dot all the other tests on that line are never executed. You can exploit this interesting feature to rewrite (and optimize) some complex *If* statements composed of multiple Boolean subexpressions:

```
' This series of subexpressions connected by the AND operator:
If x > 0 And Sqr(y) > x And Log(x) < z Then z = 0
' can be rewritten as:
Select Case False
    Case x > 0, Sqr(y) > x, Log(x) < z
        ' Do nothing if any of the above meets the condition,
        ' that is, is False.
    Case Else
        ' This is executed only if all the above are True.
        z = 0
End Select

' This series of subexpressions connected by the OR operator:
If x = 0 Or y < x ^ 2 Or x * y = 100 Then z = 0
' can be rewritten as:
Select Case True
    Case x = 0, y < x ^ 2, x * y = 100
        ' This is executed as soon as one of the above is found
        ' to be True.
        z = 0
End Select
```

As it is for similarly unorthodox optimization techniques, my suggestion is to thoroughly comment your code, explaining what you're doing and always including the original *If* statement as a remark. This technique is highly effective for speeding up portions of your code, but you should never forget that optimization isn't all that important if you're going to forget what you did or if your code looks obscure to colleagues who have to maintain it.

Then comes the *GoTo* statement, deemed to be the main cause of tons of spaghetti code that plagues many applications. I must admit, however, that my attitude toward this four-letter keyword isn't so negative. In fact, I still prefer one single *GoTo* statement to a chain of *Exit Do* or *Exit For* statements for getting out of a series of nested loops. I suggest this: Use the *GoTo* statement as an exception to the regular flow of execution, and always use significant label names and meaningful remarks all over the code to explain what you're doing.

The *GoSub…Return* keyword pair is a little bit better than *GoTo* because it's more structured. In some cases, using *GoSub* to call a piece of code inside the current procedure is better than calling an external *Sub* or *Function.* You can neither pass arguments nor receive return values; but, on the other hand, the called code shares all the parameters and local variables with your current procedure, so in most cases you don't need to pass anything. You should be aware, however, that when you compile to native code, the *GoSub* keyword is about 6 to 7 times *slower* than a call to an external procedure in the same module, so always benchmark the two approaches if you're writing time-critical code.

Working with MDI and Menus

Introduction to MDI.

This chapter concentrates on menus and MDI application that includes a parent form and a child form. When an application is designed, it might contain a number of forms and each of these forms are displayed separately on the screen and moved, minimized or minimized separately from any other form. These forms can be organized as a group using MDI.

The *multiple-document interface (MDI* **)** allows the developer to create an application that maintains multiple forms within a single container form. Applications such as Microsoft Excel and Microsoft Word are examples for multiple-document interfaces.

An MDI application allows the user to display multiple documents at the same time, with each document displayed in its own window. Documents or child windows are contained in a parent window, which provides a workspace for all the child windows in the application.

For example, Microsoft Excel allows the users to create and display multiple-document windows of different types. Each individual window is confined to the area of the Excel parent window. When the users minimize Excel, all of the document windows are minimized as well; only the parent window's icon appears in the task bar.

A child form is an ordinary form that has its MDIChild property set to true. An application can include many MDI child forms of similar or different types.

Creating MDI Form.

Select **Project** menu, choose **Add MDI Form.**

An application can have only one MDI form. If a project already has an MDI form, the Add MDI Form command on the Project menu is disabled.

Creating application's child forms.

To create an MDI child form, create a new form (or open an existing one) and set its MDIChild property to True.

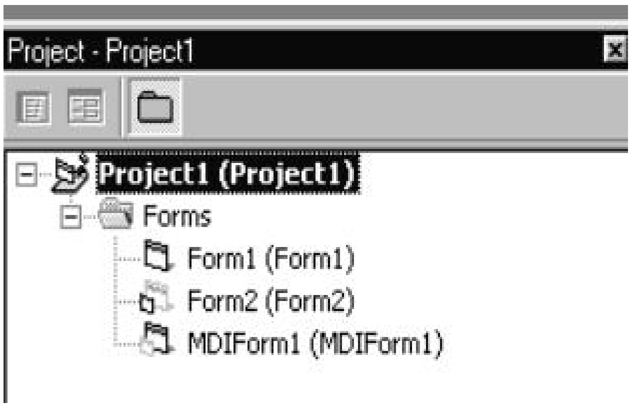
Working with MDI Child Forms at Design Time.

At design time, child forms are not restricted to the area inside the MDI form. The users can add controls, set properties, write code, and design the features of child forms just as the user would do with any other Visual Basic form.

The users can determine whether a form is an MDI child by looking at its MDIChild property or by examining the Project Explorer. If the form's MDIChild property is set to True, it is a child form.

Visual Basic displays special icons in the Project Explorer.

For the MDI and MDI child forms, as shown in the **figure** below.

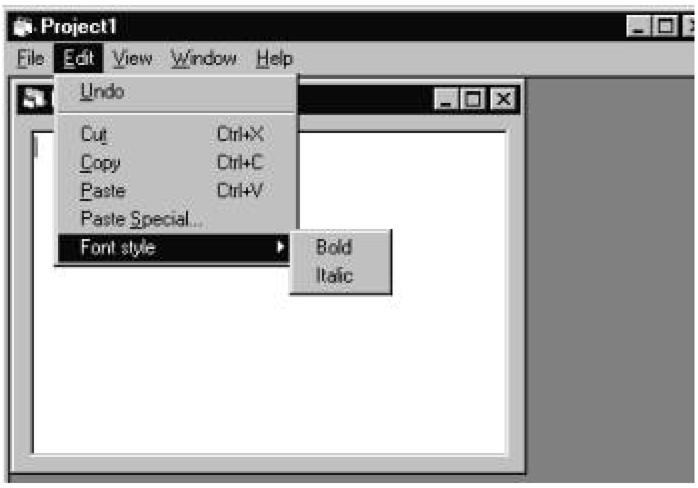


Icons in the Project Explorer with MDI child, standard, and MDI forms .

Introduction to Menu.

- Menu Terminologies.

The figure below shows the various parts of a menu.



Parts of a Menu.

- Menu bar - The menu bar always appears under the application title bar which contains names of menus.

- Menu - The menu contains the list of commands that appear when the users click a menu bar item. The list includes the menu title at the top.

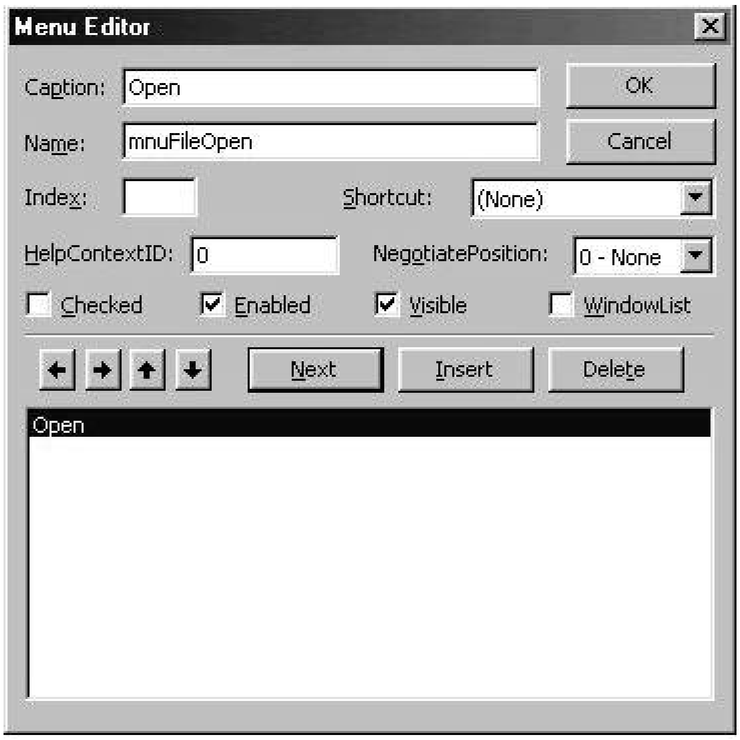
- Menu Item - A menu item, also called a command, refers to one of the choices listed on a menu. According to standard user-interface design guidelines, every menu should contain at least one command.

- Submenu - A submenu, or cascading menu, is a menu that branches off from another menu item. The command from which the cascading menu branches have an arrow next to it to indicate that a new menu will appear when the user clicks that command.

Pop-up menu - A pop-up menu is a context-sensitive menu that typically appears when the user click the right mouse button in the users application-however, this can be controlled through the code.

Menu Editor.

The following is the Menu Editor window.



Menu Editor.

As you can see, the editor has two general sections. In the top half, you set the properties we enumerated on the previous page.

In the bottom half you create the hierarchical structure of the menu (the hierarchy determines how the menu items are organized and displayed on the Form).

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

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

Caption.

Simply use the shortest name you can. Users hate long captions because they take up to much space on the screen and reading them slows down using the menu. Also, try to use a caption that does not have the same first letter as any other menu caption. This will allow you to use the first letter of the control caption as the shortcut - it makes it much easier for the user to remember!.

Name.

While it can be anything, remember that the menu event will bear this name. For easy identification, the name should be prefixed with mnu e.g. mnuFile, mnuEdit.

Checked.

Menu items are either checked or not. You can check it from within the menu editor or by using code. Generally, you'll add checkmarks to menu options that perform on or off actions, such as to display or not to display a Form. For example, this code will cause a menu item to be displayed with a small checkmark to its left.

```
mnuFileOpen.checked = True
```

Visible.

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

```
mnuFileOpen.visible = False
```

Enabled.

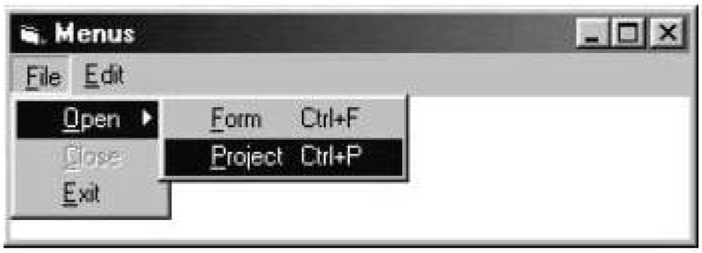
To allow the user to see the menu, but not to select it, set the enabled property to False.

```
mnuFileOpen.enabled = False
```

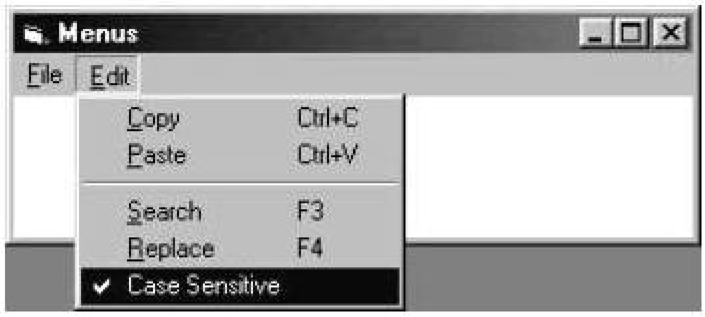
Shortcut.

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

Example.



File Menu.



Edit Menu.

As shown in the figure, the File Menu has 3 menu items: Open, Close, and Exit. When you click on the Open submenu, you can find the items labeled Form and Project. Under the Edit menu, you have items Copy, Paste, Search, and Replace. Follow through the following steps:

- On the menu bar, click Tools.

- Click Menu Editor.
- Let's create the File menu. Enter the following property settings.

Caption &File.

Name mnuFile.

Leave the rest of the properties as they are: Enabled and Visible.

- Click Next.

Since Open is a menu item under File, click the right arrow button. This also causes all the succeeding new menu items to fall under the Open menu.

- For the Open menu item.

Caption	&Open
Name	mnuOpen

- Click Next.
- Click the right arrow button to create a submenu.
- For the Form submenu.

Caption	&Form
Name	mnuForm
Shortcut	Ctrl+F

- Click Next.
- For the Project submenu.

Caption	&Project
Name	mnuProject
Shortcut	Ctrl+P

- Click Next.
- Click the left arrow button.
- For the Close submenu.

Caption	&Close
Name	mnuClose
Enabled	Disable by unchecking the checkbox

- Click Next.
- For the Exit submenu.

Caption &Exit

Name mnuExit

- Click Next.
- Click the left arrow button.
- Create the Edit menu.

Caption &Edit

Name mnuEdit

- Click Next.
- Click the right arrow button to add items for the Edit menu.

You can always edit an existing menu structure.

- Delete an item: just select the menu item and press the Delete button.
- Edit the properties of a menu item: select the item to be edited and edit the properties.
- Modify the order of items listed in the menu editor: select the item and press either the (↑) up or (↓)down button.
- You can still use the left (←) or right (→)buttons to move items from one level to another.
- To insert a menu item: Select the position in the menu hierarchy where you will insert the item and press the Insert button.

Procedures & functions.

Procedures and functions provide a means of producing structured programs. Rather than repeating the same operations at several different places in the program, they can be placed in a procedure or function. This not only makes the program easier to read, but saves a lot of time and effort in maintaining the program.

Procedures.

A procedure is a block of code that performs some operation. The events we have been using so far are a special form of procedure known as an **event procedure**. For example, associating code with a CommandButton to quit an application is a procedure.

The basic Syntax for a procedure is:

[**Private** | **Public**][**Static**] **Sub** procName ([arglist])

Parts of the Procedure	
Part	**Description**
Public	Indicates that the procedure is available to all modules. If **Option Private** is used in the module, the procedure is not available to modules outside of the project.
Private	Indicates that the procedure is only available to other procedures or functions in the current module or form.
Static	Indicates that all variables declared within the procedure are retained, even when the procedure is out of scope.
procName	The name of the procedure. Must be unique to the module if declared Private, otherwise unique to the project. The name of the procedure follows the naming rule for Variables.
Arglist	A list of variables passed to the procedure as arguments, and their data types. Commas

	separate multiple arguments. Arguments may be Optional, and may be Read Only.

Parts of the procedure

The following example is a Private Procedure to print the sum of two numbers on the Form.

```
Private Sub printSum(ByVal x As Integer, ByVal y As Integer)
   Debug.Print Str(x + y)
End Sub
```

Optional Arguments.

If an argument has the **Optional** keyword in front of it, then that argument does not have to be provided to the procedure. MsgBox is an example of a procedure that takes optional arguments. If an argument is omitted, the comma must still be used as a placeholder. The following calls the MsgBox procedure, but omits the second parameter which describes the dialog box.

```
   MsgBox "Hello", , "Greeting"
```

The following example is a procedure that uses optional arguments to specify a recipient and a sender.

```
Private Sub greeting(strMessage As String, Optional strRecipient As String, Optional
strSender As String)

   If strRecipient <> "" Then
      Debug.Print "To " & strRecipient & ", ";
   End If

   If strSender <> "" Then
      Debug.Print strSender & " sends the message ";
   End If

   Debug.Print strMessage
End Sub
```

The **greeting** procedure may be called with any of the following:

```
   greeting "Hello, how are you"
   greeting "Hello, how are you?", "Computer"
   greeting "Hello, how are you?", "Computer", "Gez"
   greeting "Hello, how are you?", , "Gez"
```

Read Only Arguments.

The optional parts ByRef and ByVal are used to determine whether an argument is a copy of the variable, or the actual variable. ByRef indicates the variable is passed by reference, and any changes made within the procedure to the variable will be reflected to where the procedure was called. ByRef is the default in Visual Basic 6. ByVal indicates the variable was passed by value, and any changes made within the procedure to the variable will not be reflected to where the procedure was called, as it's only a copy.

This example is a procedure that alters a variable. The actual variable from where it was called will be changed.

```
Private Sub changeWhereCalled(ByRef num As Integer)
    num = num + 1
    Debug.Print Str(num)
End Sub
```

This example is a procedure that alters a variable, but does not change the value where it was called.

```
Private Sub noChangeWhereCalled(ByVal num As Integer)
    num = num + 1
    Debug.Print Str(num)
End Sub
```

Functions.

A function is similar to a procedure, except it returns a value to the calling code. The basic Syntax for a function is:

```
[Private | Public][Static] Function funcName ([arglist]) As Data Type
    ' Procedure body here
    [funcName = expression]
End Function
```

Modules.

Modules contain procedures or functions that may be called anywhere within the project if they're declared as Public. To add a new module into the current project, either select "Add Module" from the Project menu, or right-click the Project in the Project Explorer and select "Add", then "Module". The module only has one property, Name. The three-letter mnemonic for a Module name is "mod" (eg. modMathRoutines).

Sub Main.

Sub Main is a special procedure that may be used by Visual Basic to launch an application. The procedure should be written in a module. The procedure may not be declared using the keyword Private. The Sub Main procedure should be selected as the "Startup Object" in the "Project Properties" from the "Project" menu.

This example uses Sub Main in a Module to prompt for a name. The name is then added to the caption of the Form. To try this example, add a Module to your project and change the Startup Object in Project Properties to Sub Main. This example uses a form called frmStart. Change it to the name of the Form you want to start.

```
Public Sub Main()
    Dim strName As String

    strName = InputBox("Enter your name", "Name")
    frmStart.Caption = "Hello " & strName
    frmStart.Show
End Sub
```

Next>> Next>>

Introduction to Forms.

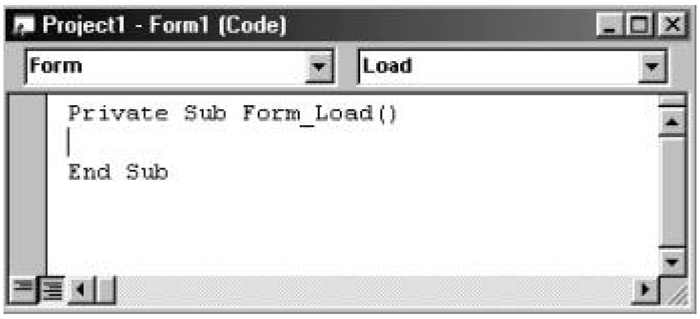
The *form* is the central control in the development of a Visual Basic project.The form appears when the users begin a new project.

The main elements of form designs are:

a.Form Window.

b.Code Window.

Form Window is a Form. **Code Window** is common to all the controls in an application. Code window has two parts, Controls and Events. Double clicking the control opens the corresponding code window with default events. **Load** is a default event for **Form.**



```
Private Sub Form_Load()

End Sub
```

Figure Shows Code Window.

Form Properties.

Like all controls, the form has many (over 40) properties. Some of the properties are:

Property.	**Description.**
Name.	Name used to identify the form.
Caption.	Text that appears in the title bar of the form.
Icon.	Reference to icon that appears in title bar of the form.
Left.	Distance from left side of the computer screen to left side of the form.
Top.	Distance from top side of the computer screen to top side of the form.
Width.	Width of the form.
Height.	Height of form.
BackColor.	Background color of form.
BorderStyle.	Form can either be sizable (can resize using the mouse) or fixed size.
Font.	A font for whole form.
WindowState.	It mentions the visual state of the form window at runtime.

Form Properties.

Form Events.

The Form primarily acts as a '*container'* for other controls. It does support events. By using those events it can respond to some user interactions.

Event.	**Description.**
Click.	Event executed when user clicks on the form with the mouse.
Load.	Event executed when the form first loads into the computer's memory. This is a good place to set initial values for various properties and other project values.
DblClick.	Event executed when user double clicks the mouse on the form.
GotFocus.	Event executed when form is focused.
Initialize.	Event executed when form is initialized.
KeyDown.	Event executed when any key is pressed but not released .
KeyUP.	Event executed when any pressed key is released.
KeyPress.	Event executed when any key is pressed.
MouseDown.	Event executed when mouse button is pressed but not released.
MouseMove.	Event executed when mouse is moved over the form.
MouseUp.	Event executed when pressed mouse button is released.

Form Events

The control names are used in event procedures. This is not true for forms. All form event procedures have the format like.

Form_EventName.
Irrespective of the *Name* property, which the user has assigned to the form, *event procedures* are listed under the word *Form*. So, when looking for form event procedures in the code window, scroll down the *Object List* until *Form* is located.

Designing a Program.

Example: Add two numbers and display the result.

The user has to input two numbers in the text box provided and when he clicks on the Add button the result has to be displayed in the third text box. So first the user has to design the screen and secondly, he has to write code for the Add command button's click event.

Step 1: GUI Design.

Design the form as shown below.

Create three text boxes, three labels and a command button.

Name it as given below.

Controls.	**Name.**	**Caption.**
Label1	LabelNum1	Enter Number 1
Label2	LabelNum2	Enter Number 2
Label3	LabelResult	Result
Command1	CmdAdd	Add

Text1	TextNum1
Text2	TextNum2
Text3	TextResult

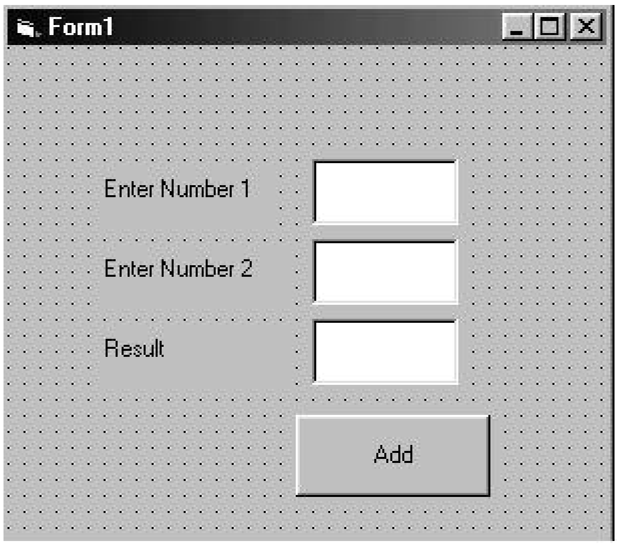
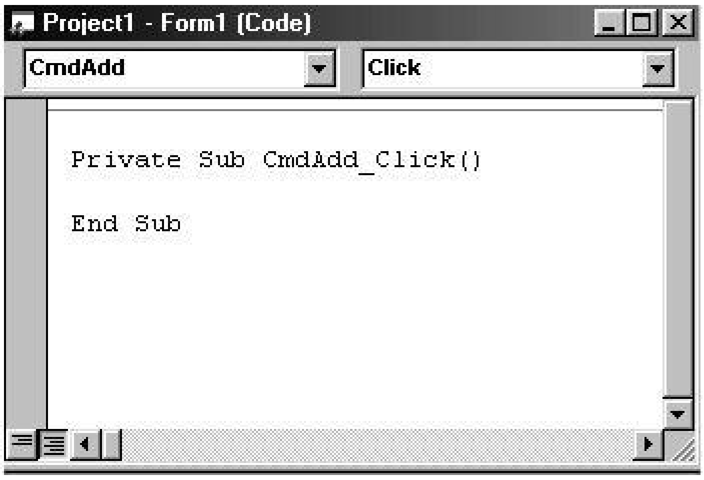


Figure Shows Form Design View.

Step 2: Writing Code.

For a command button the default event is click event. When you double click on the Add command button on the form, the two lines of code is displayed in code window as shown in the figure below, which is common for all the events. This means any code written between these two lines will be executed when the command button is clicked in the run time.



```
Private Sub CmdAdd_Click()

End Sub
```

Figure Shows Code window of CmdAdd button

The code is as following.:

```
Private Sub CmdAdd_Click()

    TextResult.Text = Val(TextNum1.Text) + Val(TextNum2.Text)

End Sub
```

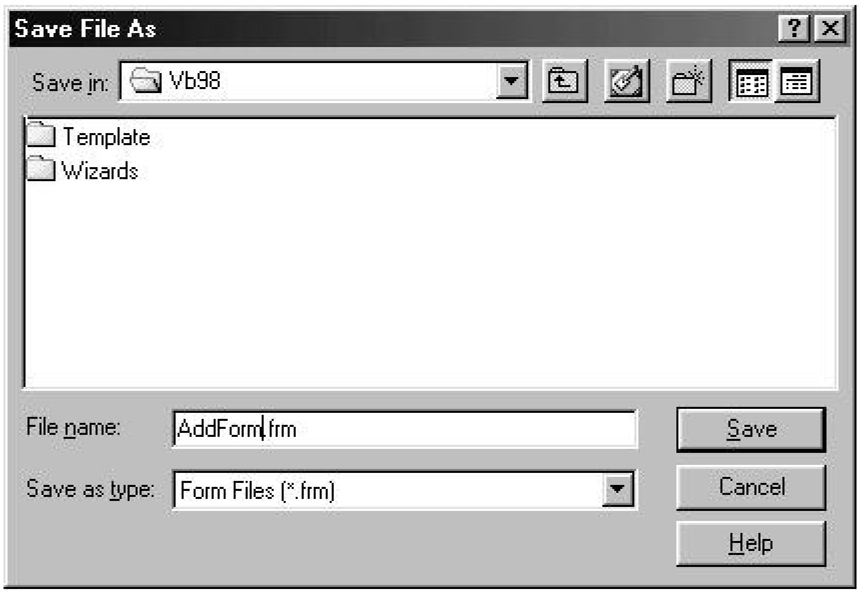
Saving Visual Basic Project.

Steps used to save a Visual Basic Project are :

Click **File** menu.

Select **Save Project.**

Since Forms are stored within the Project, first, enter Form name and its location. It is displayed in the following window.



File name: AddForm.frm

Save as type: Form Files (*.frm)

Figure Shows Saving a Form.

Give Form name and specify the location and click on the **Save** button.

Now it will ask us to give the Project name and location for the Project.

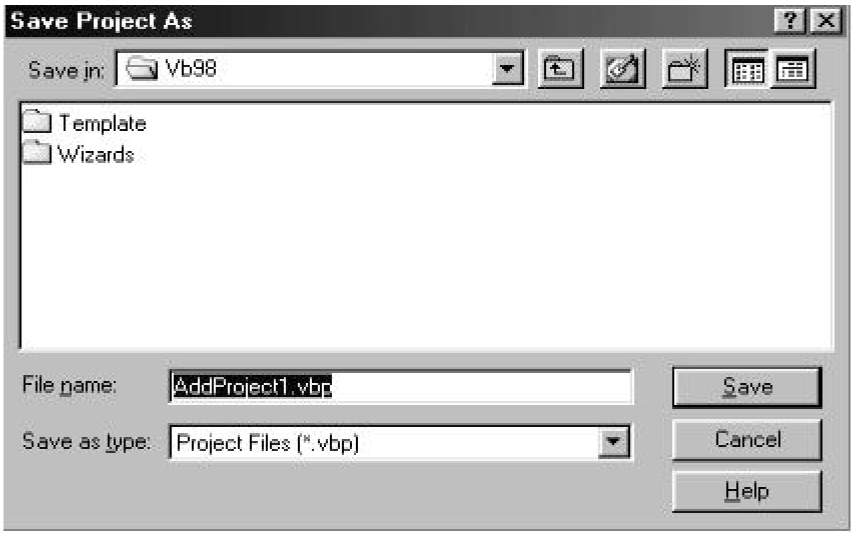


Figure Shows Saving a Project.

Give Project name and location and click on the **Save** button.

Suppose, if more than one form is inserted in the project, it will ask us to save all the Forms. Follow step 3 to save a form.

Running Visual Basic Project

To run a Visual Basic Project :

1. Press F5 **(or)** Go to **Run** Menu and click **Start.** (Run->Start) **(or)**
2. Use ▶ icon, which appears in the tool bar.

Loop Structures.

Loop Structures are statements that execute instructions repeatedly. A common practical application is when you need to compute 3^6. This expression is evaluated by multiplying itself 6 times (3*3*3*3*3*3)

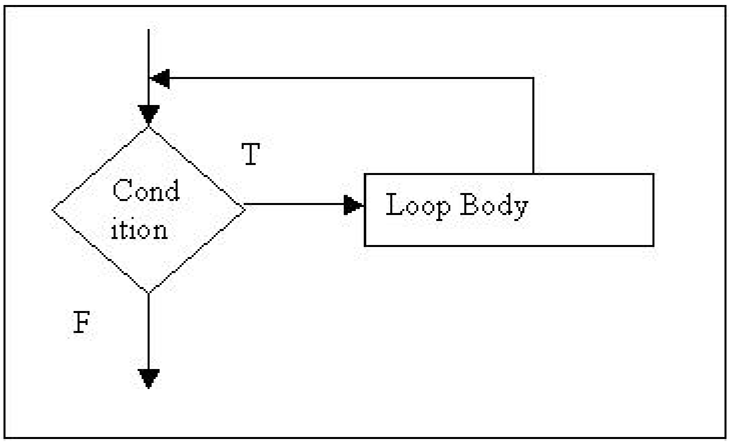
Visual Basic provides several Loop Structures. They are classified as Sentinel-controlled Structures and Counter-controlled Structures. Sentinel-controlled Loop Structures iterate routines until a special value called sentinel value contains a certain value to indicate. For example, we iterate until variable *done* has the value *True*. Variable *done* in this case is our sentinel. Counter-controlled repetition requires a counter variable (or sometimes called a loop counter). The counter variable is incremented (or decremented) every iteration. Loop terminates when the counter value reaches a particular value.

Do-While Loop Structure

Syntax:

Do While (*<condition>*)

    One or more VB Statements
Loop



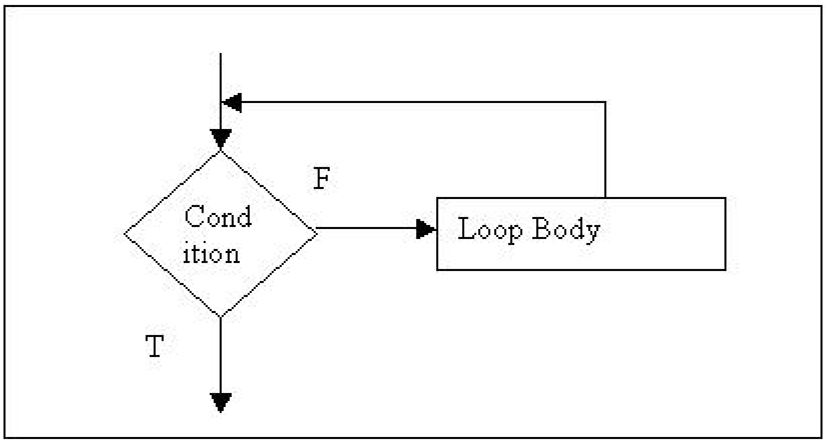
Flowchart of Do-While loop structure

The statements enclosed by *Do-While* and *Loop*, called the *loop body*, are executed repeatedly while *<condition> results a* boolean expression -True. *<condition>* is evaluated the first time the loop begins. Thus, if *<condition>* is initially *False*, the loop body will never execute. One of the statements in the loop body should somehow set the *<condition>* to *False* to terminate the loop. If it remains *True*, VB will perpetually execute the statements. This is called *infinite loop* and often causes the computer to appear to have hung.

Do-Until Loop Structure.

Syntax:

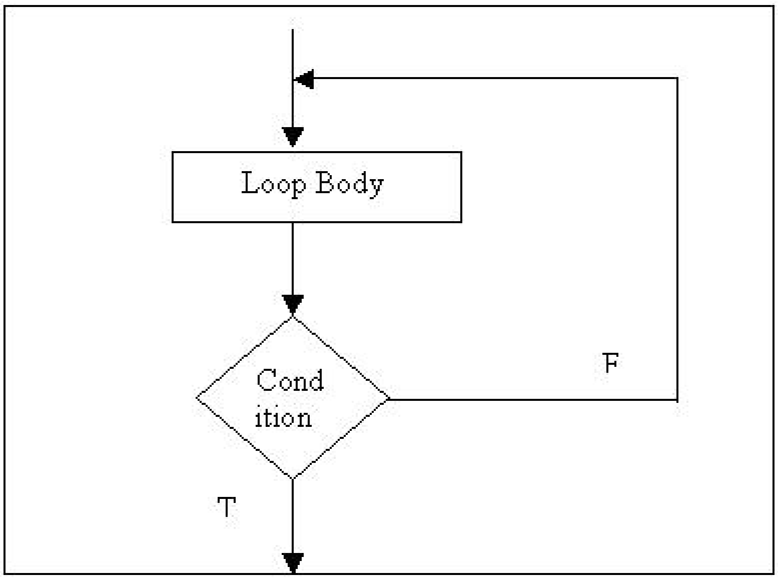
Do Until (*<condition>*)
    One or more VB Statements
Loop



Flowchart of Do-Until Loop structure.

Do-Until loop works exactly like the *Do-While* loop except that the *Do-Until* loop continues executing the loop body until the comparison is *True*. Just like *Do-While*, *Do-Until* evaluates *<condition>* first to determine if the loop body should be executed.

Do-Loop-Until Loop Structure
Syntax:
Do
   One or more VB Statements
Loop **Until** (<comparison_test>)
Condition
    Loop Body
FT
T



Flowchart of Do-Loop Until loop structure

This structure works exactly like the preceding two loop structures. Unlike the first two, the loop body is executed first before evaluating *<condition>*. If *<condition>* is true, VB repeats the loop body. Otherwise, VB exits the loop and executes the statement following the loop code. *Do-Loop-Until* executes the loop body at least once.

For Loop Structure

Syntax:

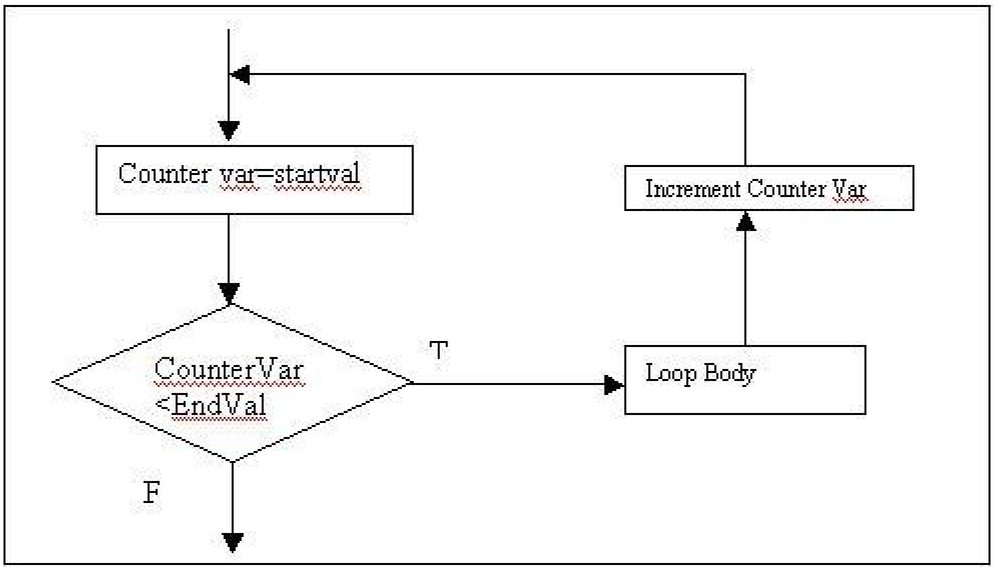
For *<counter_var>* = *<start_val>* **To** *<end_val>* **Step** *<increment_val>*
   One or more VB statements
Next *<counter_var>*

The **For-Loop** also iterates a block of statements. Unlike the other loop structures that have been discussed, *For-Loop* iterates for a specified number of times.

The number of iterations is determined by *<start_val>* and *<end_val>*, both *Integers*. Initially, *<counter_var>*, an *Integer* variable, receives the value of *<start_val>*. *<counter_var>* is incremented by the value of *<increment_val>* every iteration. By default, *<increment_val>* is equal to 1. In this case, the loop terminates when *<counter_var>* is greater than *<end_val>*. Thus, if *<start_val>* is equal to 1 and *<end_val>* is equal to 5, then there would be 5 iterations. *<counter_var>* will have the value 1 in the first iteration, 2 in the second iteration, and 5 in the last iteration.

CounterVar<EndVal
Counter var=startval
T

F
Loop Body
Increment Counter Var



Flowchart for For loop structure

The *Step* clause is optional (*Step 1* by default). If a negative value is assigned to *<increment_val>*, Visual Basic counts down. The *Next* clause tells the *For-Loop* to add *<increment_val>* to *<counter_var>* and checks if it has not gone beyond *<end_val>*. The *<counter_var>*s after *For* and *Next* should be the same variable.

Take the following examples:

For x = 2 **To** 7
  Loop Body
Next x

The loop body is iterated 6 times. Value of *x* is incremented by one every iteration (*x*=2 in the first iteration, *x*=3 in the second iteration, .. , *x*=7 in the last iteration).

For x = 2 **To** 7 **Step** 2
  Loop Body
Next x

The loop body is iterated 3 times. Value of *x* is incremented by two every iteration (*x*=2 in the first iteration, *x*=4 in the second iteration, and *x*=6 in the last iteration). At the end of the third iteration, *Next x* assigns the value 8 to *x*. Since it is greater than 7, the *<end_val>*, the loop terminates.

For x = 9 **To** 1 **Step** -3
  Loop Body
Next x

The loop body is iterated 3 times. Value of *x* is decremented by three every iteration (*x*=9 in the first iteration, *x*=6 in the second iteration, and *x*=3 in the last iteration). At the end of the third iteration, *Next x* assigns the value 0 to *x*. Since it is less than 1, the *<end_val>*, the loop terminates.

The following *Do-While* and *For-Loop* structures are the same:

For x = 1 **To** 8 **Step** 2 *Loop Body* **Next** x	x= 1 **Do While** (x<=9) *Loop Body* x = x + 2 Loop

For-Loop is a perfect structure for programs that requires a specific number of iterations (instead of a condition which is met at no specific time) or for a program that needs to access a sequence of values.

What Is an Array?

Let us first understand the necessity of an array. For storing the roll number of a student, you use a variable as shown below.

Rollno = 101

Suppose if you want to store 100 student's roll number, then what will you do? Then you need to have 100 variables to store them as rollno1, rollno2 ….rollno100. this would be a tedious task and at the same time it is difficult to keep track of all the student's rollnumber. Array helps us to do this in a simple manner.

An array is a collection of similar variables, in which each has the same name and all are of the same type. Remember that a variable can be thought of as a cup that holds an unknown value or an always-changing value.

Think of an array, then, as a collection of cups. Each cup in the collection can hold the same type of data, and every cup in the collection has the same name. Each cup within the collection is an *element* and has a number assigned to it that reflects its position within the collection. The first element of an array usually has a position number of 0 (zero).

Arrays can be different sizes. One array might have three elements, another might have 30 elements, and it's even possible for an array to have no elements at all.

Declaring Arrays.

To declare an array the following syntax is used:

Dim|Public|Private *ArrayName*(*Subscript*) As *DataType*

In this syntax,

Dim , **Public**, and **Private** are Visual Basic keywords that declare the array and its scope. If you use **Dim**, the array is private to the procedure in which it is declared. **Public** makes the array visible from anywhere in the program, and **Private** (within the General section of a form or module) makes the array visible only to the form or module in which it's declared.

ArrayName is the name of the array.

Dim rollno(100) As Integer

To assign a value to each element in the array *rollno* the following syntax is used.

rollno(0) = 90
rollno (1) = 34

```
rollno (2) = 27
rollno (3) = 10
rollno (4) = 89
```

To change the value of the fourth element of the array *rollno* from 10 to 45, you would do as follows:

```
rollno (3) = 45
```

1. **Changing the Number of Elements in an Array.**

Although the number of elements in an array is set while declaring it, it is also possible to alter the size of the array. When you change the number of elements in an existing array, it is said that the array has been *redimensioned*.

To do use the ReDim keyword as shown in the following syntax :

ReDim [Preserve] *ArrayName*(*Subscript*) As *DataType*

In this **syntax** ,

ReDim is the Visual Basic keyword denoting that the array is being redimensioned.

Preserve is an optional Visual Basic keyword that forces all pre-existing elements in the array to hold their values. If you don't use the Preserve keyword when you redimension the array, the value of all elements will be changed to zero for numeric data types, and a zero-length string ("") for variable-length strings. Fixed-length strings will be filled with zeros, and variants will be initialized to EMPTY, which could be either zero or a zero-length string, depending on the expression.

ArrayName is the name of the array.

Subscript is the subscript for the highest element in the array.

As is the Visual Basic keyword that signifies a type declaration. When redimensioning an array, the *As* keyword is optional.

DataType is any valid Visual Basic data type, such as Integer or Double. When redimensioning an array, the *DataType* is optional and cannot be changed with the Redim keyword unless the array is of type Variant.

1. **Using ReDim in your code .**

The actual implementation of the ReDim statement is different from this conceptual illustration. If you create an array that you'll later redimension, you can't hard code the element size of the array when you first declare it.

The following code shows how, conceptually, to redimension the array newArray

```
ReDim Preserve rollno (9)
rollno (9) = 23
```

To create an array that you'll later resize, you must first create the array without any elements.

1. **Setting upper and lower bounds.**

While declaring an array, we must suffix the array name by the upper limit in paranthesis. For example, an array declarations can appear in the Declarations section of a module.

The default lower bound of an array is 0. for example:

Dim counter(14) as integer

Creates an array with 15 elements, with index numbers running from 0 to 14. However we can change the default lower bound to 1 by placing an Option Base statement in the Declaration section of a module as:

Option Base 1

Another way to specify lower bounds is to provide it explicitly using the To keyword.

Dim counter(1 to 15) As integer

Here the index number of counter ranges from 1 to 15

User Defined Data Types.

Visual Basic also allows users to define their own data types.

Any data type that the user defines using the Type reserved word is the User-Defined data type. User-defined data types can contain one or more elements of any fundamental Visual Basic data

The Visual Basic keyword Type signals the beginning of a user defined type declaration. An example of User-defined data-type will clear the point.

Now suppose we want to keep the descriptive data of the employees in our organization, like name, address, city, pincode, and telephone number. Instead of defining variables for each and every detail of the employee, we can use user-defined data-type very efficiently. The syntax is as follows:

```
Type info

        name as stri
        ng

        address as string
        city as strin
g
        pincode as string
        telephone as long
End Type
```

Here we have just described the structure of our user-defined data-type, we have not yet reserved any storage for the data. After defining the new data-type we can create variables using Dim statement in the manner given below :

Dim employee as info.

The above statement creates the variable employee. The defined variable contains all the components of the data-type we defined. When we use the defined variable name, we refer to all the data simultaneously that is, in the above example, when we use the variable employee, we refer to all the data like name, address, city, pincode and the telephone number simultaneously. When we want to refer to a single component, we can use the variable name followed by a '.' and the component name. Thus employee, telephone is a long data component that stores the telephone number of the employee.

Loop Statements

The most frequently used looping structure in Visual Basic is undoubtedly the *For...Next* loop:

```
For counter = startvalue To endvalue [Step increment]
    ' Statements to be executed in the loop...
Next
```

You need to specify the *Step* clause only if *increment* is not equal to 1. You can exit the loop using an *Exit For* statement, but unfortunately Visual Basic doesn't provide any sort of "Repeat" command that lets you skip the remaining part of the current iteration and restart the loop. The best you can do is use (nested) *If* statements or, if you don't want to make the logic too complex, use a plain *GoTo* keyword that points to the end of the loop. In fact, this is one of the few occasions when a single *GoTo* statement can make your code *more* readable and maintainable:

```
For counter = 1 To 100
    ' Do your stuff here ...
    ' if you want to skip over what follows, just GoTo NextLoop.
    If Err Then Goto NextLoop
    ' more code that you don't want to enclose within nested IF blocks
    ' ...
NextLoop:
Next
```

TIP

Always use an Integer or Long variable as the controlling variable of a *For...Next* loop because they're faster than a Single or a Double controlling variable, by a factor of 10 or more. If you need to increment a floating-point quantity, the most efficient technique is explained in the next example.

CAUTION

A compelling reason to stay clear of floating-point variables as controlling variables in *For...Next* loops is that, because of rounding errors, you can't be completely sure that a floating-point variable is incremented correctly when the increment is a fractional quantity, and you might end up with fewer or more iterations than expected:

```
Dim d As Single, count As Long
For d = 0 To 1 Step 0.1
    count = count + 1
Next
```

```
Print count          ' Displays "10" but should be "11"
```

When you want to be absolutely sure that a loop is executed a given number of times, use an integer controlling variable and explicitly increment the floating-point variable within the loop:

```
Dim d As Single, count As Long
' Scale start and end values by a factor of 10
' so that you can use integers to control the loop.
For count = 0 To 10
    ' Do what you want with the D variable, and then increment it
    ' to be ready for the next iteration of the loop.
    d = d + 0.1
Next
```

I covered the *For Each...Next* loop already in Chapter 4, and I won't repeat its description here. I just want to show you a neat trick that's based on this type of loop and the *Array* function. This technique permits you to execute a block of statements with different values for a controlling variable, which don't need to be in sequence:

```
' Test if Number can be divided by any of the first 10 prime numbers.
Dim var As Variant, NotPrime As Boolean
For Each var In Array(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
    If (Number Mod var) = 0 Then NotPrime = True: Exit For
Next
```

The values don't even have to be numeric:

```
' Test if SourceString contains the strings "one", "two", "three", etc.
Dim var2 As Variant, MatchFound As Boolean
For Each var2 In Array("one", "two", "three", "four", "five")
    If InStr(1, SourceString, var2, vbTextCompare) Then
        MatchFound = True: Exit For
    End If
Next
```

The *Do...Loop* structure is more flexible than the *For...Next* loop in that you can place the termination test either at the beginning or the end of the loop. (In the latter case, the loop is always executed at least once.) You can use either the *While* clause (repeat while the test condition is True) or the *Until* clause (repeat while the test condition is False). You can exit a *Do* loop at any moment by executing an *Exit Do* statement, but—as with *For...Next* loops—VBA doesn't offer a keyword that skips over the remaining statements in the loop and immediately restarts the loop.

```
' Example of a Do loop with test condition at the top.
' This loop is never executed if x <= 0.
Do While x > 0
    y = y + 1
    x = x \ 2
Loop

' Example of a Do loop with test condition at the bottom.
' This loop is always executed at least once, even if x <= 0.
Do
    y = y + 1
    x = x \ 2
Loop Until x <= 0

' Endless loop: requires an Exit Do statement to get out.
Do
     ...
Loop
```

The *While...Wend* loop is conceptually similar to the *Do While...Loop*. But you can test the condition only at the beginning of the loop, you don't have an *Until* clause, and you don't even have an *Exit While* command. For these reasons, most programmers prefer the more flexible *Do...Loop* structure, and in fact you won't see a single *While...Wend* loop in this entire book.

Other Functions

A few VBA functions are closely related to control flow, even if by themselves they don't alter the execution flow. The *IIf* function, for example, can often replace an *If...Else...End If* block, as in the following code:

```
' These lines are equivalent.
If x > 0 Then y = 10 Else y = 20
y = IIf(x > 0, 10, 20)
```

The *Choose* function lets you select a value in a group; you can use it to distinguish among three or more cases. So, instead of this code:

```
' The classic three-choices selection
If x > y Then
    Print "X greater than Y"
ElseIf x < y Then
    Print "X less than Y"
Else
    Print "X equals Y"
End If
```

you can use this shorter version:

```
' Shortened form, based on Sgn() and Choose() functions.
' Note how you keep the result of Sgn() in the range 1-3.
Print "X " & Choose(Sgn(x _ y) + 2, "less than", "equals", _
    "greater than") & " Y"
```

The *Switch* function accepts a list of *(condition, value)* pairs and returns the first *value* that corresponds to a *condition* that evaluates as True. See, for example, how you can use this function to replace this *Select Case* block:

```
Select Case x
```

```
    Case Is <= 10: y = 1
    Case 11 To 100: y = 2
    Case 101 To 1000: y = 3
    Case Else: y = 4
End Select
```

Same effect in just one line.

```
' The last "True" expression replaces the "Else" clause.
y = Switch(x <= 10, 1, x <= 100, 2, x <= 1000, 3, True, 4)
```

You should remember two things when you're using this function: First, if none of the expressions returns a True value, the *Switch* function returns Null. Second, all the expressions are always evaluated, even though only one value is returned. For these reasons, you might get unexpected errors or undesired side effects. (For example, if one expression raises an overflow or division-by-zero error.)

CAUTION

While the *IIf*, *Choose*, and *Switch* functions are sometimes useful for reducing the amount of code you have to write, you should be aware that they're always slower than the *If* or *Select Case* structure that they're meant to replace. For this reason, you should never use them in time-critical loops.

File Handling in VB

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

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

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

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

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

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

Here is a little more useful example:

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

In this example, the code reads one line at a time, walking through the file until the end-of-file (EOF) is true. The line of data (minus the carriage return / line feed characters) is put into the string variable temp$. In the applications each line can be saved into an array, or they can be concatenated into a single string variable, or any other processing that is desired can be done.

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

```
OPEN "filename" FOR INPUT AS #1
WHILE NOT EOF(1)
LINE INPUT #1, temp$
Alltext$ = alltext$ &  temp$s & vbcrlf
WEND
Textbox1.text = alltext$
CLOSE #1
```

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

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

```
OPEN "filename" FOR OUTPUT AS #1
For I=1 to 10
PRINT #1, I
NEXT I
CLOSE #1
```

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

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

```
OPEN "filename" For INPUT as #1
WHILE NOT EOF(1)
LINE INPUT #1, TEMP$
Number1 = VAL (mid$(temp$, 1,10))
Number2 = VAL (mid$(temp$, 11,10))
Number3 = VAL (mid$(temp$, 21,10))
WEND
CLOSE #1
```

Binary (Data) Files

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

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

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

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

```
OPEN "filename" for BINARY  as #1
PUT #1, Var1, Var2, Var3
CLOSE #1
```

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

```
OPEN "filename" for BINARY as #1
GET #1, var1, var2, var3
CLOSE #1
```

File Operations in Visual Basic

Let us five basic file type operations using built in Visual Basic functions. These techniques are considered old style by programmers familiar with Object Oriented programming who prefer to use the File System Object (FSO). We can categorize these operations in two ways: operations on files and operations on directries, or the newer term, folders.

a. Copying Files

When you copy a file you keep the existing files, and make a copy of the file with a new name. Visual Basic provides a statement for this called FileCopy.

Here is the syntax for the FileCopy statement.

FileCopy source, destination

Where source is the name of the file to copy, and destination is the name of the copied file.

There are several choices when it comes to specifying the file names here ---you can use the full path names for the files, or you can just specify the name of the files.

By way of background, Windows keeps track of something called the current drive and the current directory for us --- thses are basically pointers in the File System, and in the old days of DOS allowed us to perform mundane file operations without having to specify the name of the Drive and the Directory. These pointers still carry on in VB and Windows, so if we use this syntax in the Click Event Procedure of a Command Button

```
Private Sub Command1_Click()
FileCopy "a.txt", "b.txt"
End Sub
```

Windows looks for a file called "a.txt" in the current drive and current directory of out PC, and if the operating system finds it, copies the file as "b.txt", again in the default drive and directory of the PC.

The problem here is that if the file is not found, the program bombs, just like this

- **The Current Drive and Current Directory**

As it turns out, Visual Basic has a function that can be used to determine the current directory called the CurDir function…

```
Private Sub Command2_Click()
MsgBox "The current directory is " & CurDir
End Sub
```

Once you know the current directory, you can then use the ChDir and ChDrive functions to change either the current drive or the current directory, like thi…

```
Private Sub Command3_Click()
ChDrive ("d")
```

```
ChDir "\vbfiles"
MsgBox "The current directory is " & CurDir
End Sub
```

Now you may not want to leave anything to chance, in which case, you can use the full path name with the FileCopy Statement, like this…..

```
Private Sub Command1_Click()
FileCopy "c:\vbfiles\a.txt", "c:vbfiles\b.txt"
End Sub
```

If you attempt to copy a file that is opened, you'll receive this error message…

- Does a file exist?

There is no confirmation that the copy was successful, but you can determine if a file exists by using the Visual Basic Dir$ function. The Dir$ function requires just a single argument representing the file name (as was the case with the Copy File statement, you can specify just the file name or the full path name). If the file is found, then Dir$ returns the file name (not the full path). If the file is not found, then Dir$ returns an empty string. Let's see how we can use the Dir$ function to determine if a file exists before we copy it.

```
Private Sub Command1_Click()
Dim retval As String
Retval = Dir$("c:\vbfiles\b.txt")
If retval = "b.txt" then
Msgbox "b.txt exists—no need to copy it…"
Else
FileCopy "C:\vbfiles\a.txt", "c:\vbfiles\b.txt"
End if
End sub
```

If we now run the program, and click on the command button…

We receive a message saying that the file already exists---Dir$ has done its job.

By the way, you will discover that there is a Dir function as well---Dir returns a variant return value and Dir$ returns a string.

b. Renaming Files

Renaming files is similar to copying them—this time we use the Visual Basic Name statement. Here is the syntax.

Syntax

```
Name oldpathname As newpathname
```

As was the case when we copied files, we can choose either to specify a file name or to include the full path name.

```
Private Sub Command1_Click()
Name "c:\vbfiles\b.txt" as "c:\vbfiles\newb.txt"
End Sub
```

This code will result in the file 'b.txt' begin renamed to 'newb.txt'. Once again, do not expect a confirmation message telling you that the rename was successful—the only message you will receive is if the file does not exist.

c. Deleting Files

The final file operation is that of deleting a file. Visual Basic provides us with the kill statement which will delete a file (and dangerously, a wildcard selection of files) of our choosing. Here's the syntax…

Syntax

```
Kill pathname
```

Let us say that we wish to delete the file "newb.txt' file that we created just a few minues ago. This code will do the trick…

```
Private Sub Command1…Click()
Kill "c:\vbfiles\newb.txt"
End Sub
```

Angain, there will be no confirmation message, only an error message if the file we are attempting to delete does not exist.

As mentioned, you can use wildcards as an argument to the Kill statement (WARNING: Don't attempt this at home!!!). For instance, this code will delete EVERY file in the \VBFILES directory that has a file extension of *.txt…

```
Private Sub Command1_Click()
Kill "C:\vbfiles\*.txt"
End Sub
```

Most dangerously, this code will delete EVERY A FILE IN THE \vbfiles directory…

```
Private Sub Command1_Click()
Kill "C:\vbfiles\*.*"
End Sub
```

Be careful when using the Kill statement---when issued through Visual Basic, there's no going back. There is no Undo statement, and files deleted in this way are NOT moved to the Windows Recycle Bin.

d. Moving Files

There is no explicit Visual Basic statement to move a file. To simulate a move of a file, all we need to do is combine the FileCopy and Kill statements that we've already seen. For instance, to move the file a.txt from C:\VBFILES to c:\VBILES\CHINA, we can execut this code…

```
Private Sub Command1_Click()
FileCopy "c:\vbfiles\a.txt", "c:\vbfiles\china\a.txt"
Kill "c:\vbfiles\a.txt"
End Sub
```

Again, do not expect any confirmation messages---only errors if the files you reference do not exist.

That's it for Visual Basic (Primitive) action that we can take against files.

Introduction to DAO.

The Data Access Object (DAO) model is the Jet Database Engine's Object Oriented interface. It is a hierarchy of classes that correspond to a logical view of a relational system, such as the database itself, the tables defined in it and their fields, indexes and so on. These classes are used to create data access objects that refer to the particular database.

In order to work with data access objects, a reference has to be set to the appropriate DAO library.

To add this,

Select References from the Project menu.

Select the DAO 3.51 Object library

Click ok.

DAO hierarchy is shown below.

Data Access Object Hierarchy.

The element in the DAO hierarchy is actually classes, not objects. A class is similar to a data type, in that it describes what "kind" of object user is referring to. For example in the following declaration,

```
Dim MyWs as workspace.
```

MyWs is a variable that stands for an object of the workspace class. Most of the Data Access Objects are represented as both "Objects and Collection".

The table below lists some of the data access objects and the collections they represent.

AO.	Represents.	Contains Collection.
DBEngine.	Jet Database Engine.	Workspace.
Workspace.	As user session.	Databases, Users, Groups.
Database.	An open Database.	TableDefs, QueryDefs, Recordsets.
Tabledef.	Stored definition of a base table.	Fields, Indexes.
Field.	A column in a table.	-
Index.	An index defined on a table.	Fields.
QueryDef.	Stored information of a query.	Fields, Parameters.
Recordset.	Records in a base table or records resulting from a query.	Fields.

DAO and Collections.

The table below lists the data access objects and collections they can be appended to,

Object.	Collection.
Workspace.	Workspaces.
Database.	Databases.
TableDef.	TableDefs.
Field.	Fields.
Index.	Indexes.
QueryDef.	QueryDefs.
Recordset.	Recordsets.

DAO and More collections.

The member objects of a collection can be accessed through a Zero-based index. For example, the first TableDef in a Database called MyDatabase is referred to as MyDatabase.TableDefs(0). The second TableDef is referred to as MyDatabase.TableDefs(1) and so on.

Objects in the hierarchy are identified by the full "path" through the nested collections to which they belong, using the .(dot) navigation operator. Thus,

DBEngine.Workspaces(0).Databases(0).TableDef(0).Fields("Dept_no").

refers to the filed names "Dept_No" in the first TableDef in the TableDefs collection of the first Database in the Database collection of the first workspace in the workspaces collection of the DBEngine.

Opening an existing Database.

The OpenDatabase method is used to open a specified database. This method returns a reference to the Database object that represents it. The open database is automatically added to the Databases collection.

Syntax:

```
Dim db as Database.

Set db = OpenDatabase("Employee.mdb");
```

Db is a variable that represents the Database Object. To open the database for exclusive use, the following syntax is used:

Set db = OpenDatabase("Employee.mdb", True).

The value TRUE indicates that no other user can open the database. The default is False.

To open the database in the read only mode, the following syntax is used:

Set db = OpenDatabase("Employee", False, True).

That is the third argument True, will provide only a read access on the database.

Creating a Recordset Object.

A RecordSet Object represents the records in the base table or the records that result form a query. A RecordSet Object is created on a database object. The Recordsets collection of the database object contains all open RecordSet Objects created. The RecordSet object also has Fields that contains all the Field Objects of a record in the RecordSet.

Type.

If type is not specified, OpenRecordset creates a table-type Recordset. If an attached table or query is specified, OpenRecordset creates a dynaset-type Recordset.

A Recordset object represents the records in a base table or the records that results from a running query. The different types of Recordset Objects and their description are dealt in the following table.

Record Type.	Description.
Table-Type Recordset.	(vbTableType) A set of records that represents a single table that can be used to add, change, or delete records.
Dynaset-Recordset.	(vbDynasetType) A dynamic set of records that represent a database table or the results of a query containing fields from an one or more tables. Records can be add, change, or delete from a dynaset-type Recordset, and the changes will be reflected in the underlying table(s).
Snapshot-type Recordset.	(vbSnapshotType) A static copy of a set of records that can be used to find data or generate reports. A snapshot-Type Recordset can contain fields from one or more tables in database but cannot be updated.

Types of RecordSet Objects and their description.

Opening a RecordSet.

The OpenRecordSet method is used to create a new RecordSet Object and append it to the RecordSet collection of the Database.

Syntax:

Dim rs as Recordset

Set rs = db.OpenRecordset("Employee", dbOpenTable, dbReadOnly)

Here db is the variable that represents the database object. The dbOpendTable specifies the type of Recordset to be created, the dbReadOnly opens the recordset in a read only mode.

Navigating through a recordset.

The Move method moves to a record in a database specified by a row number. An optional parameter may be specified to move relative to a bookmark.

The following example moves to the fourth record in the Recordset.

MoveFirst:

The MoveFirst method moves to the first record in the Recordset.

Syntax:

rs.MoveFirst.

MoveLast:

The MoveLast method moves to the last record in the Recordset.

Syntax:

rs.MoveLast.

MoveNext:

The MoveNext method moves to the next record in the Recordset. The EOF property should be checked to prevent an error occurring. EOF is set to True if the current record is after the last record in the Recordset.

Syntax:

```
rs.MoveNext
If rs.EOF Then
rs.MovePrevious
MsgBox "At last record", vbInformation, Database"
End If.
```

MovePrevious:

The MovePrevious method moves to the previous record in the Recordset. The BOF property should be checked to prevent an error occurring. BOF is set to True if the current record is before the first record in the Recordset.

```
rs.MovePrevious
If rs.BOF Then
rs.Recordset.MoveNext
MsgBox "At first record", vbInformation, "Database"
End If.
```

Locating a Record in a Recordset:

The Recordset object has a set of Find methods that enable you to locate a specific record according to some search criteria. If a record matching the criteria isn't located, the NoMatch property is set to True. You should check NoMatch after a Find and if it is True, reposition the current record to a valid record in the Recordset. In the following examples, the Recordset's Bookmark property is used to return to the position in the Recordset before the Find method was invoked should the search criteria fail. The Bookmark property's data type is a String.

FindFirst:

Locates the first record in the Recordset with the given criteria. The search begins at the beginning of the Recordset.

Syntax:

rs.FindFirst.

FindLast:

Locates the last record in the Recordset with the given criteria. The search begins at the end of the Recordset.

rs.FindLast.

FindNext:

Locates the next record in the Recordset with the given criteria. The search begins at the current record.

Syntax:

rs.FindNext.

FindPrevious:

Locates the previous record in the Recordset with the given criteria. The search begins at the current record.

Syntax:

rs.FindPrevious.

Adding Records:

The AddNew method adds a new record to the end of the Recordset.

Syntax:

rs.AddNew.

Editing Records:

The Edit method Copies the current record from a Recordset object to the copy buffer for subsequent editing. The Copy Buffer is an area created by the Jet database engine to place records. Before using the Edit method, you should check the Updatable property of the Recordset.

Syntax:

rs.Edit.

Updating Records:

The Update method saves the contents of the copy buffer to a specified Recordset object. Use after changing data in the current record.

Syntax:

rs.Update.

Example for D A O.

Follow the steps given below to create a VB application to Accept, Edit, Delete, Save Employee details and to navigate through the records using Data Access Objects.

Create the following screen with four labels, three text boxes and eleven command buttons.

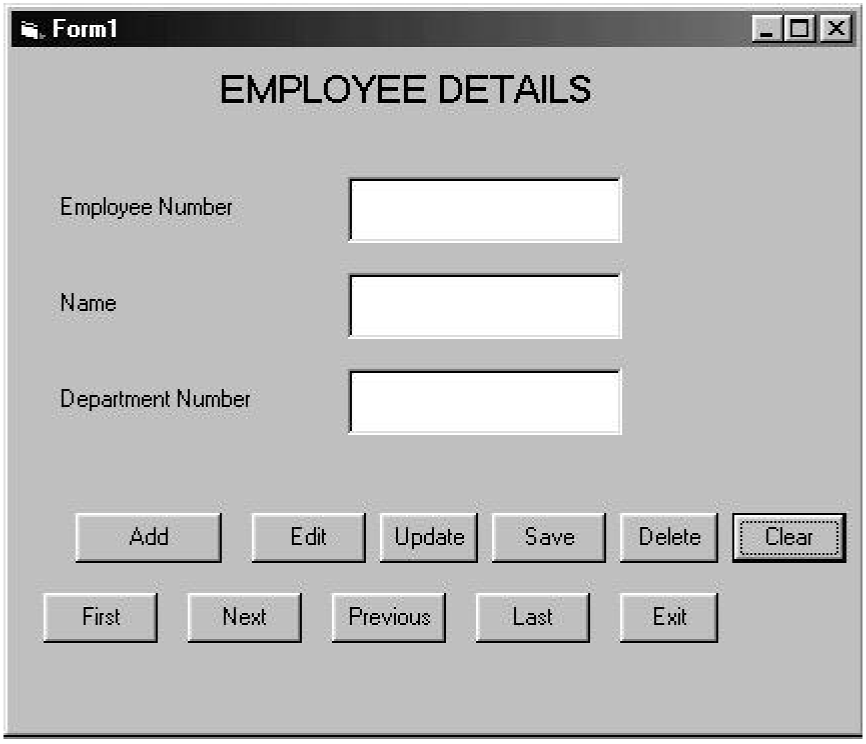
Name the labels as shown in the picture.

Name the Text boxes as EmpnoTxt, NameTxt and DnoTxt.

Name the command buttons as AddCmd, ClearCmd, DelCmd, EditCmd, ExitCmd, FirstCmd, LastCmd, NextCmd, PrevCmd, SaveCmd, UpdateCmd.

Write the code given below.

Save the form and the project and execute the file to see the following figure as shown below.



Design Screen for Employee Details .

The code below declares two objects db and rs. Declare it in the General Declaration section for opening the database and the recordset respectively. Db is a variable that represents database object.

```
Dim db As Database.

Dim rs As Recordset.
```

In the following code, the database is opened in the read only mode and also the table is opened.

```
Set db = OpenDatabase("c:\Employee.mdb", False)

Set rs = db.OpenRecordset("Emp", dbOpenTable)

End Sub
```

Type the following in the code window of AddCmd and click event. Here the text boxes are cleared and the AddNew method of the Recordset object is used to add a new row to the recordset. Once the AddNew is applied to the recordset, the edit buffer is created with the empty record in it.

Calling the update method saves the changes made by the user. The user has to click on the save button after making changes or inserting new record.

```
Private Sub AddCmd_Click()

EmpnoTxt.Text = " "

NameTxt.Text = " "

DnoTxt.Text = " "

rs.AddNew

EmpnoTxt.SetFocus

End Sub

Private Sub ClearCmd_Click()

EmpnoTxt.Text = " "

NameTxt.Text = " "

DnoTxt.Text = " "

End Sub
```

The rs.delete method deletes the current record and the recordset pointer is moved to the next record.

```
Private Sub DelCmd_Click()
```

```
rs.Delete

rs.MoveNext

If rs.EOF Then

rs.MoveLast

End If

End Sub
```

The rs.edit method sets the record to the edit mode and the changes are made.

```
Private Sub EditCmd_Click()

rs.Edit

End Sub

Private Sub ExitCmd_Click()

End

End Sub
```

The rs.MoveFirst, rs.MoveNext, rs.MoveLast, rs.MovePrevious moves the recrodset the first, next, last and previous row respectively

```
Private Sub FirstCmd_Click()

rs.MoveFirst

EmpnoTxt.Text = rs("Empno")

NameTxt.Text = rs("Name")

DnoTxt.Text = rs("Deptno")

End Sub

Private Sub LastCmd_Click()

rs.MoveLast

EmpnoTxt.Text = rs("Empno")

NameTxt.Text = rs("Name")

DnoTxt.Text = rs("Deptno")

End Sub
```

```
Private Sub NextCmd_Click()

rs.MoveNext

If rs.EOF Then

rs.MoveFirst

End If

EmpnoTxt.Text = rs("Empno")

NameTxt.Text = rs("Name")

DnoTxt.Text = rs("Deptno")

End Sub

Private Sub PrevCmd_Click()

rs.MovePrevious

If rs.BOF Then

rs.MoveFirst

End If

EmpnoTxt.Text = rs("Empno")

NameTxt.Text = rs("Name")

DnoTxt.Text = rs("Deptno")

End Sub

Private Sub SaveCmd_Click()

If rs.EditMode = dbEditAdd Then

rs("Empno") = EmpnoTxt.Text

rs("Name") = NameTxt.Text

rs("Deptno") = DnoTxt.Text

End If

rs.Update

End Sub
```

The Update method of the Recordset object saves the contents of the edit buffer to the Recordset object.

```
Private Sub UpdateCmd_Click()

rs("Empno") = EmpnoTxt.Text

rs("Name") = NameTxt.Text

rs("Deptno") = DnoTxt.Text

rs.Update

End Sub
```

An introduction to OLE DB.

OLE DB is a low level object based programming interface designed to access a variety of data sources. That is, OLE DB is capable of dealing with any type of data information regardless of its format or storage method.

It can support both relational and non-relational data sources. OLE DB is not designed to be accessed directly from VB due to its complex interfaces but it can be done through ADO as all OLE DB’s functionality is exposed by ADO.

ActiveX Data Objects.

There are two ways for accessing Database Information:

- ActiveX Data Controls (ADC).
- ActiveX Data Objects (ADO).

ADO uses a new database connection framework called OLE DB, which allows faster access to multiple data providers.

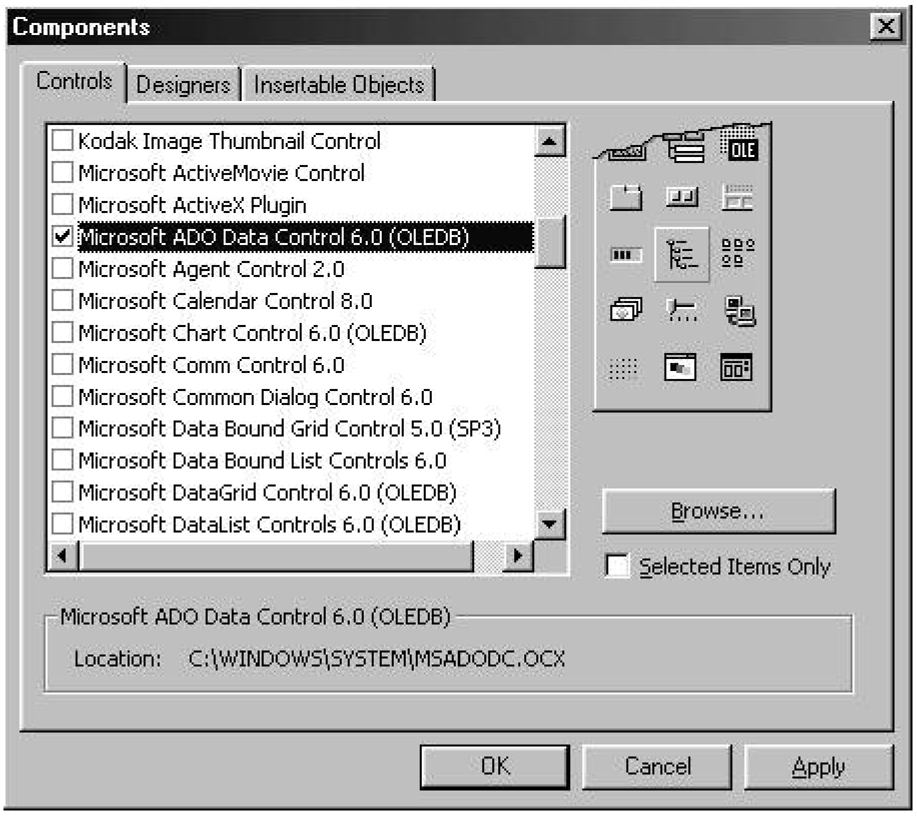
Example - Working with ADO Data Control.

Let us discuss an example to retrieve records from the Emp table of Employee database using the ADO contro1.

Start Visual Basics 6.0 and select a new standard Exe project.

Select **Project** Menu and select **Components**, you will see a figure as shown below.

Select or Check the **Microsoft ADO Data Control 6.0 (OLEDB).**


Components

Controls | Designers | Insertable Objects

- [ ] Kodak Image Thumbnail Control
- [ ] Microsoft ActiveMovie Control
- [ ] Microsoft ActiveX Plugin
- [x] Microsoft ADO Data Control 6.0 (OLEDB)
- [ ] Microsoft Agent Control 2.0
- [ ] Microsoft Calendar Control 8.0
- [ ] Microsoft Chart Control 6.0 (OLEDB)
- [ ] Microsoft Comm Control 6.0
- [ ] Microsoft Common Dialog Control 6.0
- [ ] Microsoft Data Bound Grid Control 5.0 (SP3)
- [ ] Microsoft Data Bound List Controls 6.0
- [ ] Microsoft DataGrid Control 6.0 (OLEDB)
- [ ] Microsoft DataList Controls 6.0 (OLEDB)

Browse...

- [ ] Selected Items Only

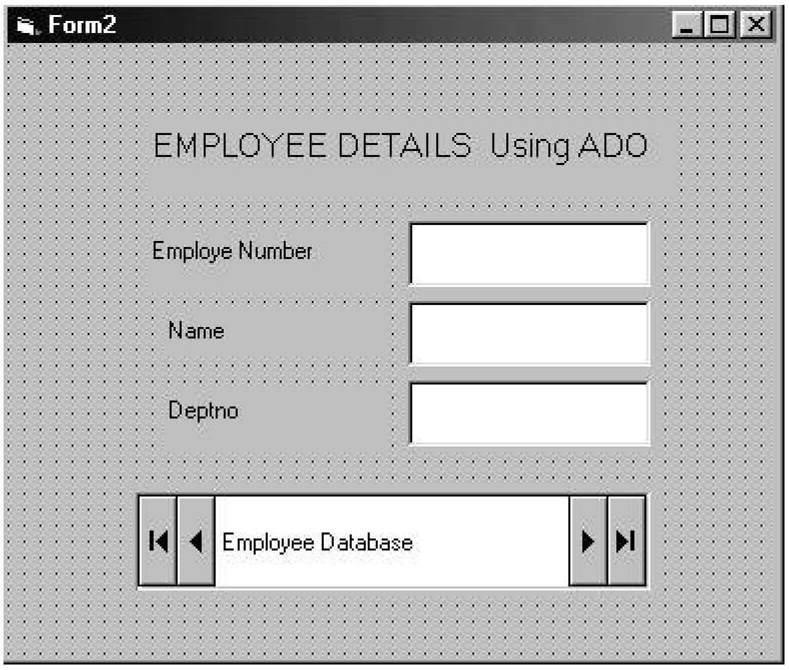
Microsoft ADO Data Control 6.0 (OLEDB)

Location: C:\WINDOWS\SYSTEM\MSADODC.OCX

OK | Cancel | Apply

Components Dialog to add OLEDB.

Design a screen as shown below with required controls along with ADO control.


Form2

EMPLOYEE DETAILS Using ADO

Employe Number

Name

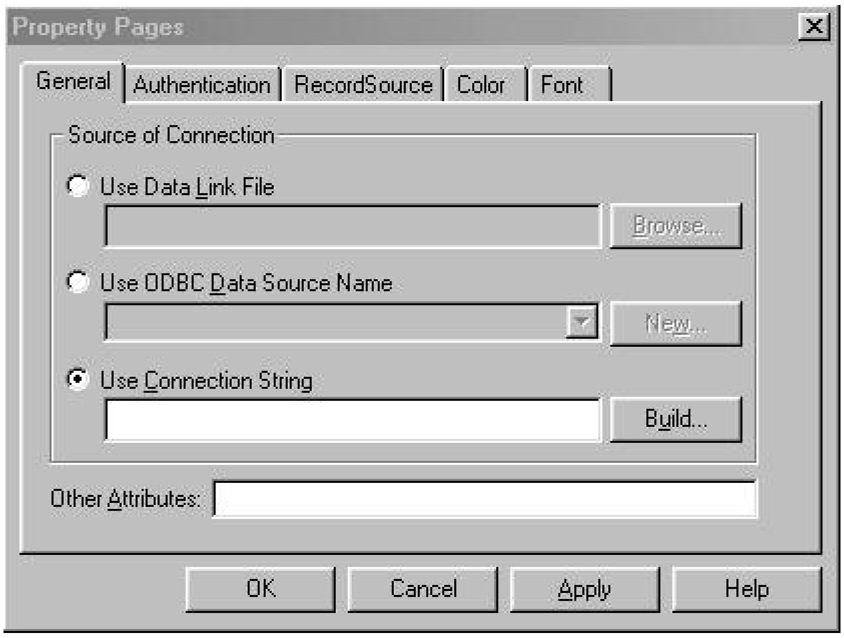
Deptno

Employee Database

Design screen to accept Employee Details using ADO control.

Set the ADO Data control's caption as **Employee Database,** and name it as Adodc1.

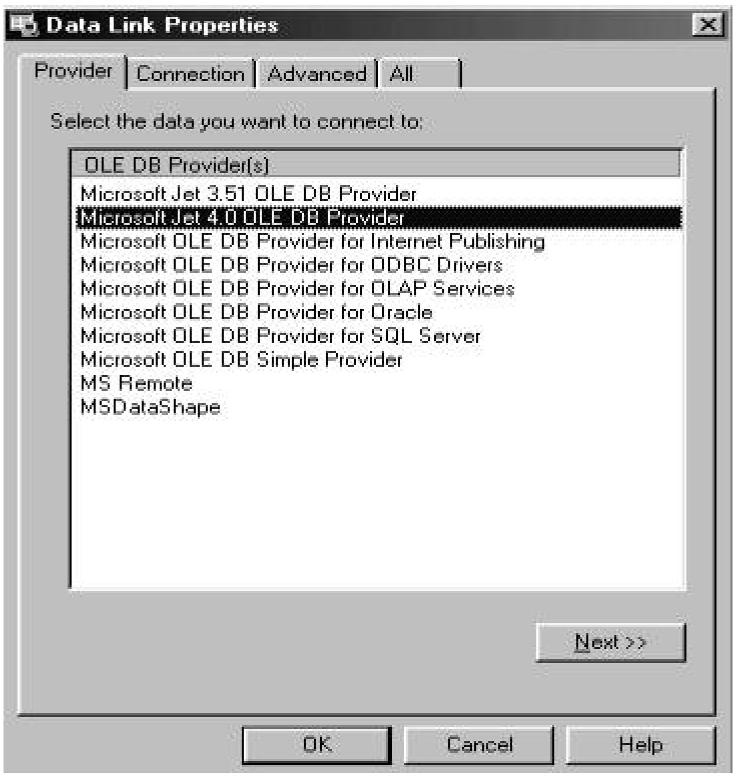
Right click the ADO DC control and select ADO DC properties, the following figure appears.



Property pages of ADO DC.

Select the Use Connection String option, click on the Build button, the following figure appears.

Select **Mircrosoft Jet 4.0 OLE DB Provider**, as we want to use jet engine to access Ms-Access database from the Data Link Properties window that is displayed as shown below. Click on the Next button.


Data Link Properties
Provider
Connection
Advanced
All
Select the data you want to connect to:
OLE DB Provider(s)
Microsoft Jet 3.51 OLE DB Provider
Microsoft Jet 4.0 OLE DB Provider
Microsoft OLE DB Provider for Internet Publishing
Microsoft OLE DB Provider for ODBC Drivers
Microsoft OLE DB Provider for OLAP Services
Microsoft OLE DB Provider for Oracle
Microsoft OLE DB Provider for SQL Server
Microsoft OLE DB Simple Provider
MS Remote
MSDataShape
Next >>
OK
Cancel
Help

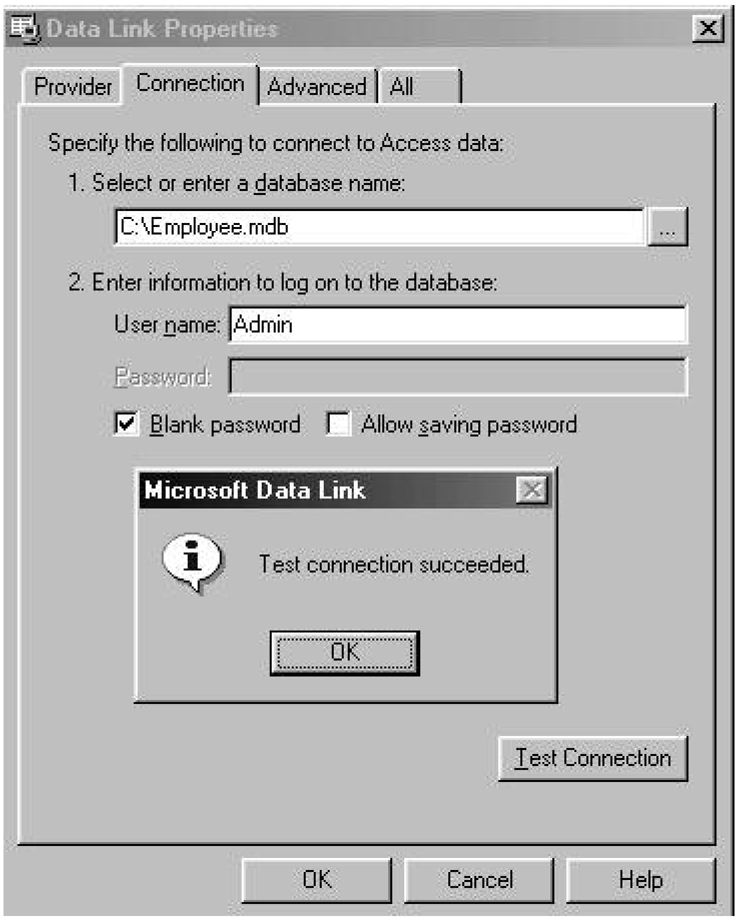
Selecting OLEDB provider.

Connection tab is seen selected. Select the Database required.

Click on the Test button, to ensure that the connection is successful.

If it is a successful connection you will see a small window indicating that the test is success as shown in figure.

Click on the OK button.



Specify the following to connect to Access data:
1. Select or enter a database name:
C:\Employee.mdb
2. Enter information to log on to the database:
User name: Admin
Password:
[✓] Blank password [ ] Allow saving password

Test connection succeeded.

Data Link Properties window on successful.

database connection.

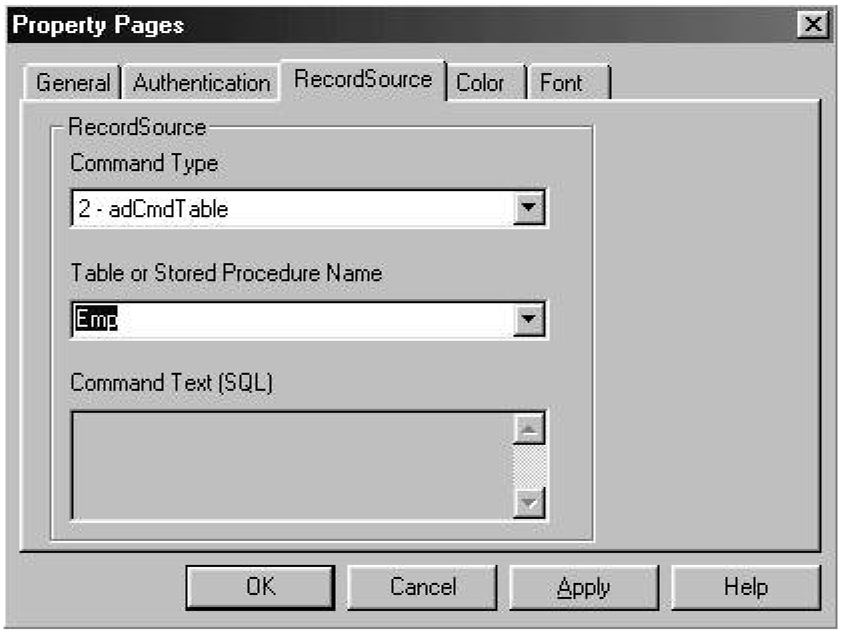
Once again click **OK** button to close the Data Link Properties window.

Select RecordSource tab.

From the **command type** list as **1- adCmdTable** as shown in the figure.

Select the table name as **Emp** or any other required table from the list given.

Finally click ok.



RecordSource

Command Type: 2 - adCmdTable

Table or Stored Procedure Name: Emp

Command Text (SQL)

Property pages window to select command type and the table.

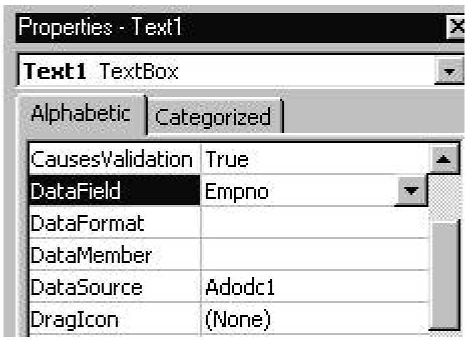
Note:

Selecting the respective OLE Db provider in the Data Link Properties can connect any other Database.

Associating the controls with respective properties.

Select the Text1 textbox and set the DataSource property as Adodc1.

Select the DataField property to the field name Empno as shown in the property sheet below.



Text1 TextBox

CausesValidation	True
DataField	Empno
DataFormat	
DataMember	
DataSource	Adodc1
DragIcon	(None)

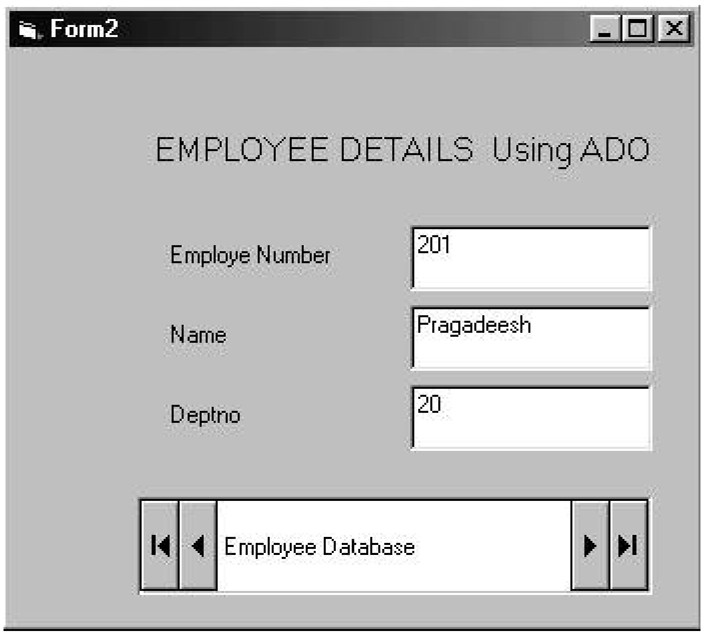
Property sheet of Textbox1.

Similarly attach the remaining textboxes Text2 and Text3 to the Name and Deptno Fields respectively.

Save the Project and Form and run the application.

You will see the output as shown below.

By clicking the navigating keys, you can browse through the records.



Form2

EMPLOYEE DETAILS Using ADO

Employe Number 201

Name Pragadeesh

Deptno 20

Output window – ADO DC.

Working with ActiveX Data Objects (ADO).

ADO is the object-based interface that provides a logical set of objects that can be accessed from code.These objects are:

Object.	Functionality.
Connection.	Establishes the connection with the data base.
Command.	Defines the commands that will be executed against the database.
Recordset.	Contains the data that is retrieved from the database.

These objects present an interface in the form of properties and methods that can be queried and manipulated. ADO was specifically developed to be small, lightweight, fast and feature complete – for the database applications or for the Internet.

An important thing in ADO is that the objects in this model are not dependant on one another. This means that one can create instances of objects independent of one another, for example, one can create a Recordset object without creating a connection object.

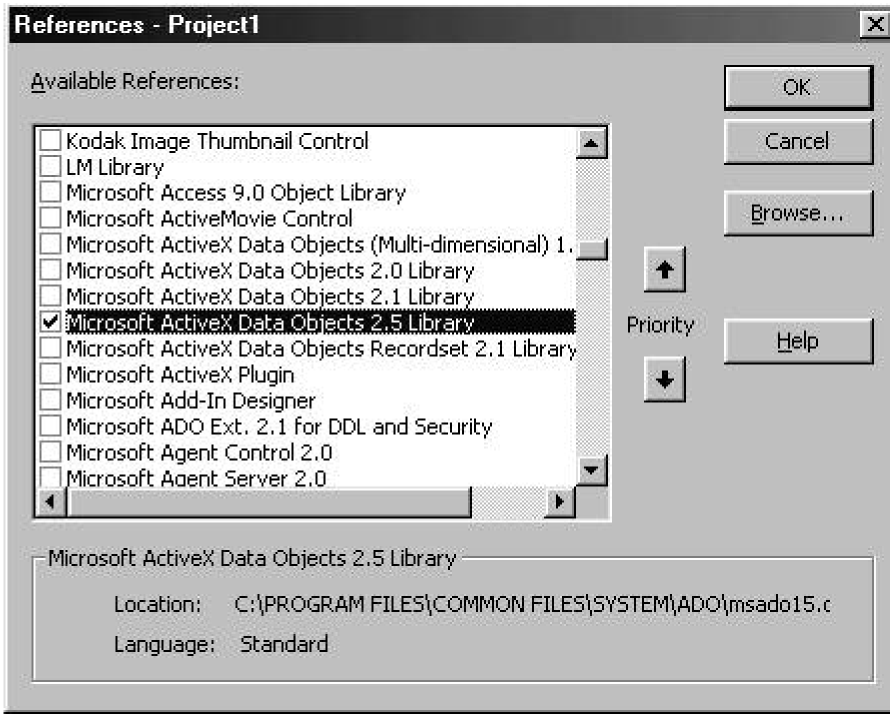
Unlike the older technologies, ADO is more flexible, ADO code is easier to write, read and maintain. ADO is built on top of OLE DB and is capable of accessing any sort of data that is wrapped and exposed by an appropriate OLE DB provider.

Connectionless recordset is not same as the disconnected recordset. Making the recordset structure externally creatable means that user can create a new recordset object anytime and anywhere in the code and one can use it without a connection to a database. A disconnected recordset supports a static, client-side cursor that automates downloading the records on the client side. One can have disconnected recordsets with RDO (Remote Data Objects) but cannot have connectionless recordsets.

A connectionless recordset is a recordset whose fields have been defined by the application to match the structure of the information user wants to manage. Previously this capability was reserved for the data object model such as ADO 1.x, RDO, or DAO.

The ADO Connection and ADO Recordset Objects are ActiveX DLLs. (Dynamic Link Libraries). They can be accessed only through the reference to the type library.

Select Project References from the Visual Basics menu and check or select Microsoft ActiveX Data Objects 2.5 library.



- [ ] Kodak Image Thumbnail Control
- [ ] LM Library
- [ ] Microsoft Access 9.0 Object Library
- [ ] Microsoft ActiveMovie Control
- [ ] Microsoft ActiveX Data Objects (Multi-dimensional) 1.
- [ ] Microsoft ActiveX Data Objects 2.0 Library
- [ ] Microsoft ActiveX Data Objects 2.1 Library
- [✓] Microsoft ActiveX Data Objects 2.5 Library
- [ ] Microsoft ActiveX Data Objects Recordset 2.1 Library
- [ ] Microsoft ActiveX Plugin
- [ ] Microsoft Add-In Designer
- [ ] Microsoft ADO Ext. 2.1 for DDL and Security
- [ ] Microsoft Agent Control 2.0
- [ ] Microsoft Agent Server 2.0

Microsoft ActiveX Data Objects 2.5 Library
Location: C:\PROGRAM FILES\COMMON FILES\SYSTEM\ADO\msado15.c
Language: Standard

Reference window - Adding library to the application.

Now we can access all the ADO objects.

The ADO object model comprises the following seven objects.

ADO Objects:	
Object.	Description.
Command.	The Command object defines the command to be given to a data source. This may be either a SQL statement or an invokation of a stored procedure.
Connection.	The Connection object may be used by the Command and Recordset objects to provide a session with the data source.
Error.	The Error object may be used if the OLE DB provider has implemented it. In its absence, errors are raised in the Visual Basic Err object.
Field.	The Field object represents a column in a Recordset object.
Parameter.	The Parameter object is implemented as a collection and stores the parameters to be used by a Command object.
Property.	The Property object may be used by the Connection, Command, Recordset, and Field objects. Each of the objects has a set of unique properties that either describe or

	control the behaviour of the object. The properties may either be built-in, or added to the Properties collection by the data provider.
Recordset.	The Recordset object contains a set of records retrieved from the Connection object.

ADO Objects.

Cursors.

The Recordset object fetches data using Cursors, which are used to identify the current position in the result set, and what row will be returned next. The Recordset object supports four cursor types.

Recordset Cursors .		
Cursor.	**Constant.**	**Description.**
Command.	adOpenDynamic.	The result set reflects subsequent changes, additions and deletions by other users. There are no limitations on how the result set may be navigated.
Forward-only.	adOpenForwardOnly.	The result set reflects subsequent changes, additions and deletions by other users. The result set may only be navigated in a forward direction, and may improve performance if you make a single pass through the result set. If no cursor type is specified, this cursor is used by default.
Keyset-Driven.	adOpenKeyset.	The result set reflects subsequent modifications to data, but not records added by other users. This cursor prevents access to records deleted by other users. There are no limitations on how the result set may be navigated.
Static.	AdOpenStatic.	Returns a static set of records that cannot be modified. Modifications, additions, and deletions to data by other users will not be seen with this type of cursor. There are no limitations on how the result set may be naviagted.

Recordset Cursors:

The cursor type is set with the CursorType property of the Recordset object.

```
        adoRS.CursorType = adOpenDynamic.
```

Opening a Recordset:

The Open method is used to retrieve a recordset from the database. The method takes four optional parameters, the recordset source, the active connection, the cursor type, the lock type, and options. The following table describes the parameters.

Open Method Parameters.	
Parameter.	**Description.**
Source.	The source may be a Command object variable, a SQL statement, a table, or a stored procedure. In the absence of this parameter, the recordset will be opened

	with the value of the **Source** property of the Recordset object.
ActiveConnection.	The ActiveConnection property may be either a valid Connection object variable, or a Connection string. In the absence of this parameter, the recordset will use the **ConnectionString** property of the Recordset object.
CursorType.	Determines the type of cursor to use with the Recordset, where the cursor type constants are adOpenDynamic, dOpenForwardOnly, adOpenKeyset, or adOpenStatic. In the absence of this parameter, the recordset will use the value of the **CursorType** property of the Recordset object. If no value is specified as either a parameter, or in the CursorType property, adOpenForwardOnly is used by default.

LockType.	Determines how to implement concurrent sessions. The following constants may be used: LockType Constants. **LockType.** — **Description.** adLockBatchOptimistic. — Uses an optimisic lock for bacth updates, as opposed to immediate updates. adLockOptimistic. — Only locks a record when an Update method is called. adLockPessimistic. — Usually locks a record immediately upon editing. adLockReadOnly. — No locking is used as the records are read only. If no value exists, adLockReadOnly is used by default. In the absence of this parameter, the recordset will be locked using the value in the **LockType** property of the Recordset object.
Options.	A constant that indicates how the provider should evaluate the Source argument if it represents something other than a Command object. LockType Constants. **CommandType.** — **Description.** adCmdText. — Specifies that the command should be interpreted as text. adCmdTable. — Specifies that the command should use a SQL statement to retrieve all records from the table specified in Source. adCmdTableDirect. — Specifies that the command should retrieve all records from the table specified in Source directly. adCmdStoredProc. — Specifies that the value of Source is a stored procedure. adCmdUnknown. — Specifies the type of command in Source is unknown. adCommandFile. — Specifies that the Recordset should be restored from the file named in Source.

	<table><tr><td>adFetchAsync.</td><td>After the initial quantity specified in the CacheSize property is</td></tr></table>

Open Method Parameters:

The following may be used to open a recordset without using any of the properties of the Recordset object.

```
adoRS.Open strSQL, strConn, adOpenForwardOnly, adLockReadOnly, adCmdText
```

Example–ADO.

The following example uses the Properties to determine how the recordset should be opened and calls the Open method with no parameters.

Type the following code in the Form_Load code window.

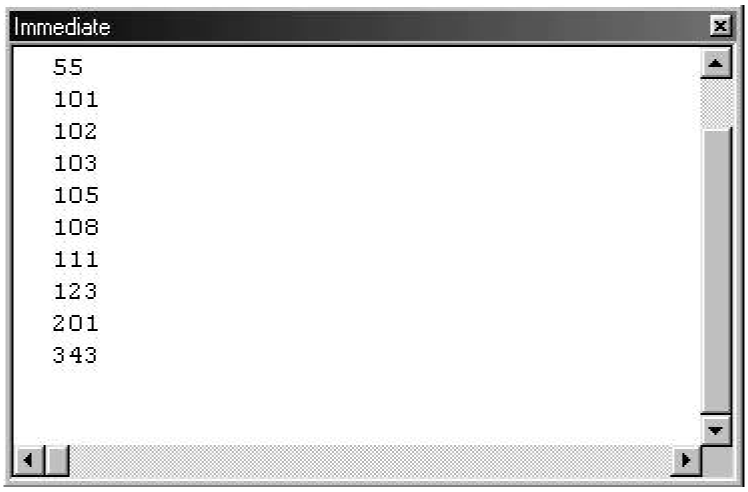
The application when executed lists all the Employee numbers in the Immediate window.

Open a stand exe project,

```
Dim adoConn As New ADODB.Connection.

Dim adoRS As New ADODB.Recordset.

Dim strConn As String.

strConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=".

strConn = strConn & "C:\employee.mdb".

adoConn.ConnectionString = strConn.

adoConn.Open.

adoRS.Source = "SELECT * FROM Emp".

adoRS.CursorType = adOpenKeyset.

adoRS.ActiveConnection = adoConn.

adoRS.Open.

Do While Not adoRS.EOF.

Debug.Print adoRS.Fields("Empno").Value.

adoRS.MoveNext.

Loop.

adoRS.Close.

Set adoRS = Nothing.

adoConn.Close.
```

Set adoConn = Nothing.

Output is printed in the Immediate window as shown in the figure.



```
55
101
102
103
105
108
111
123
201
343
```

output displayed in the immediate window.

Navigating Through a Recordset.

The Move method moves to a record in a database specified by a row number. An optional parameter may be specified to move relative to a bookmark. The following example moves to the fourth record in the Recordset.

```
adoRS.Move 4
```

MoveFirst:

The MoveFirst method moves to the first record in the Recordset.

```
adoRS.MoveFirst
```

MoveLast:

The MoveLast method moves to the last record in the Recordset.

```
adoRS.MoveLast
```

MoveNext:

The MoveNext method moves to the next record in the Recordset. The EOF property should be checked to prevent an error occurring. EOF is set to True if the current record is after the last record in the Recordset.

```
adoRS.MoveNext

If adoRS.EOF Then

adoRS.MovePrevious
```

```
MsgBox "At last record", vbInformation, "Database"

End If
```

MovePrevious:

The MovePrevious method moves to the previous record in the Recordset. The BOF property should be checked to prevent an error occurring. BOF is set to True if the current record is before the first record in the Recordset.

```
adoRS.MovePrevious

If adoRS.BOF Then

adoRS.Recordset.MoveNext

MsgBox "At first record", vbInformation, "Database"

End If
```

Adding Records:

The AddNew method adds a new record to the end of the Recordset.

```
adoRS.AddNew
```

Updating Records

The Update method saves the contents of the edit buffer to a specified Recordset object. Use after changing data in the current record.

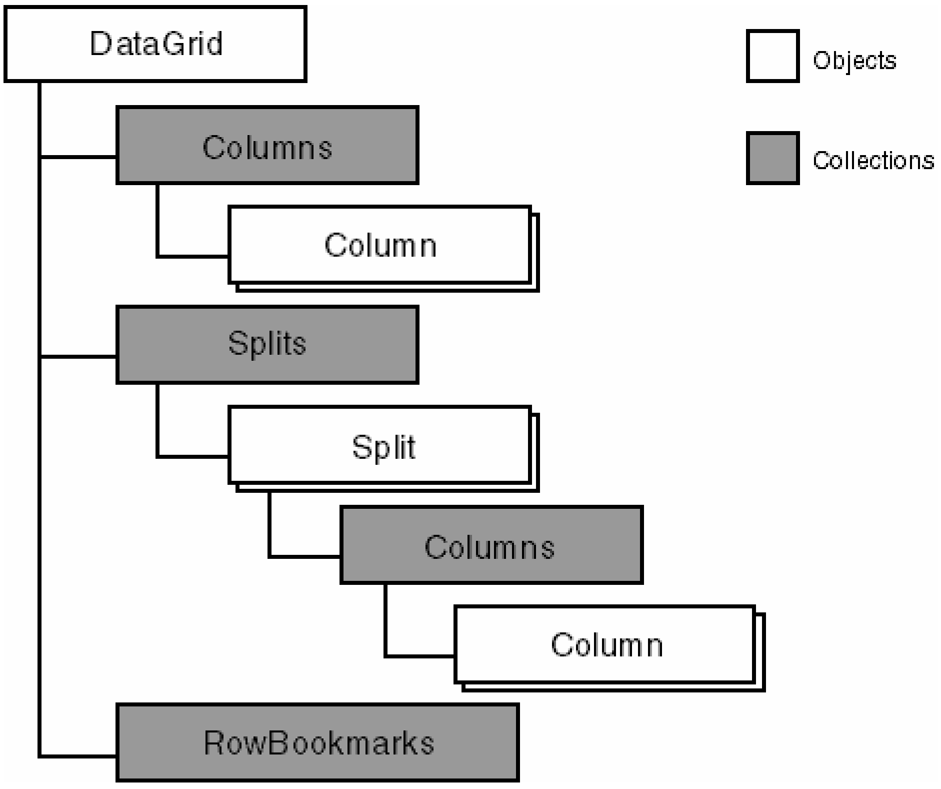
```
response = MsgBox("Update this record", vbYesNo + vbQuestion, "Update")

If response = vbYes Then

   adoRS.Update

Else

   adoRS.CancelUpdate

End If
```

The DataGrid Control

Probably the most usual way to display data in a database table is with a grid control. Visual Basic 6 comes with several grid controls, but only two of them can work with the newer ADO Data control and other ADO data sources: the DataGrid control and the Hierarchical

FlexGrid control. I'll describe the DataGrid control in this section and the Hierarchical FlexGrid control in the next section.

Before looking at the individual properties, methods, and events supported by the DataGrid control, you should be familiar with its object model. As you can see in Figure 15-2, this is a simple object model, with the DataGrid control at the top of the hierarchy and the Columns and Splits collections under it. You can split a DataGrid control into two or more sections and navigate through them independently or in a synchronized manner. The DataGrid control is included in the MSDATGRD.OCX file, which must therefore be distributed with any application that uses this control.



Setting Design-Time Properties

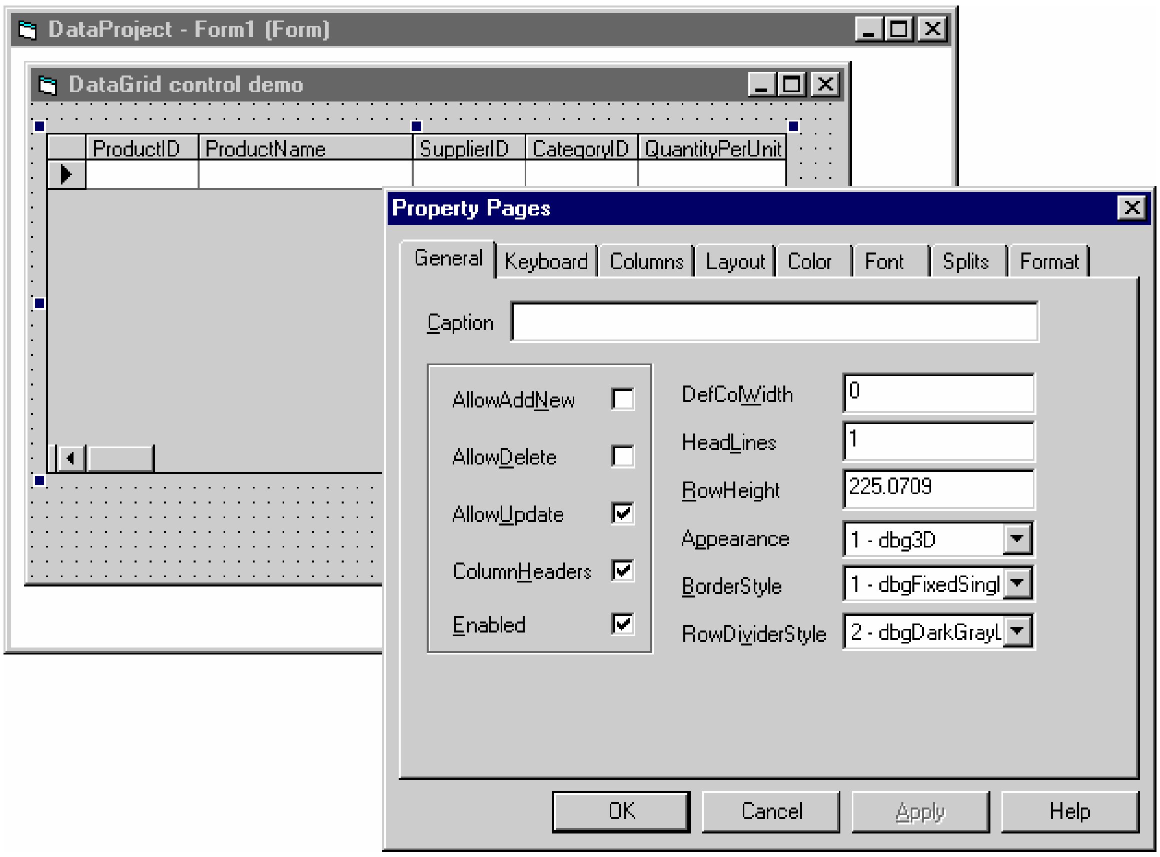
Since the DataGrid control can work only as a bound control to an ADO data source, the first thing to do is prepare such a source. This can be a design-time source such as an ADO Data control or a DataEnvironment object, or it can be a run-time source such as an ADO Recordset or an instance of a custom class that qualifies as a data source. Working with design-time sources is definitely preferable, because you can retrieve the field structure at design time and adjust column width and other attributes in a visual manner, without writing code.

Editing the Column layout

After you have bound the DataGrid control to an ADO Data control or to a DataEnvironment's Command object through the DataGrid's *DataSource* property, you can right-click on the control and select the Retrieve Fields menu command. This prepares a column layout at design time, with each column taking its caption and width directly from the database field to which it maps. You can then right-click again on the control and select the

Edit menu command, which puts the grid in edit mode. In this mode, you can adjust the column width, scroll the grid horizontally by using the scroll bar at the bottom, and right-click on the control to display a menu of commands. These commands allow you to add and remove columns, split the grid into two or more sections, cut and paste columns to rearrange their order, and so on. To modify other properties, however, you must right-click once again on the control and select the Properties command, which brings up a Property Pages dialog box with as many as eight tabs, as shown in Figure 15-3.

Contrary to what the documentation states, it seems impossible in practice to have distinct column layouts for different split sections. In fact, if you delete an existing column or add a new column to a split, all the other splits are affected as well. A possible workaround for this problem is to set a column's *Visible* property to False. Because this attribute can be set on a split-by-split basis, you can effectively hide a column in all the splits where it shouldn't appear.



Caption	
AllowAddNew	☐
AllowDelete	☐
AllowUpdate	✓
ColumnHeaders	✓
Enabled	✓
DefColWidth	0
HeadLines	1
RowHeight	225.0709
Appearance	1 - dbg3D
BorderStyle	1 - dbgFixedSingl
RowDividerStyle	2 - dbgDarkGrayL

The General and Keyboard tabs

By default, the grid has no caption, but you can enter a custom string in the General tab of the Property Pages dialog box; if a nonempty string is specified for the *Caption* property, it will appear in a gray section above the column headers. The *AllowAddNew*, *AllowDelete*, and *AllowUpdate* Boolean properties determine which operations are allowed on the grid. The *ColumnHeaders* property can be set to False to hide the gray row containing the column headers. Note that in the Font tab you can set the *HeadFont* property, which determines the character font used for column headers.

The *DefColWidth* property is the default width of the grid's columns: If set to 0 (its default value), the width of each column is the largest value between the underlying field's size and the column header's width. The *HeadLines* property is an integer between 0 and 10, and corresponds to the number of rows used for column headings; you can use 0 to remove column headers, but it's preferable to set the *ColumnHeaders* property to False to achieve the same result. The *RowHeight* property is the height of each row in twips. The DataGrid control doesn't support rows of different heights.

You can set the *BorderStyle* property to 0-dbgNoBorder to suppress the fixed border around the grid. The *RowDividerLine* property determines the style used to draw the dividing lines between the rows and can be one of the following enumerated values: 0-dbgNoDividers, 1-dbgBlackLine, 2-dbgDarkGrayLine (the default), 3dbgRaised, 4-dbgInset, or 5-dbgUseForeColor. If 3-dbgRaised or 4-dbgInset is used, the color of the dividing line depends on the Microsoft Windows settings.

The Keyboard tab allows you to set some properties that affect how keys behave when the DataGrid control has the focus. If the *AllowArrows* property is True, the user can visit all the cells in the grid using the arrow keys; if *WrapCellPointer* is also True, pressing the right arrow key at the end of a row moves the focus rectangle to the first cell in the next row and pressing the left arrow key at the beginning of a row moves the focus rectangle to the last cell in the previous row.

The *TabAction* property decides what happens when the Tab key or Shift+Tab key combination is pressed and the DataGrid control is the active control. The default action is 0dbgControlNavigation, in which case the next control (or the previous control, if Shift+Tab is pressed) on the form receives the focus. If you set this property to 1dbgColumnNavigation, pressing the Tab key moves the focus rectangle to the next column unless the current cell is the last (or the first, if Shift+Tab is pressed) of its row. In this case, pressing this key causes the focus to move to the next (or previous) control in the TabIndex order. Finally, the setting 2-dbgGridNavigation is similar to the previous one, but the Tab key never moves the focus rectangle to another control and the behavior at the beginning or end of the row depends on the *WrapCellPointer* property.

By default, tab and arrow keys never move the focus rectangle to another split in the same grid. You can, however, set the *TabAcrossSplit* property to True to let the user navigate through splits by using the Tab key. In this case, the value of the *WrapCellPointer* and *TabAction* properties are ignored, unless the user presses the Tab key when the current cell is in the last column of the rightmost split or presses the Shift+Tab key combination when the current cell is in the first column of the leftmost split.

The Columns and Format tabs

The Columns tab allows you to set the *Caption* property of each individual Column object, as well as its *DataField* property, which contains the name of the field in the data source to which the column is bound.

The Format tab allows you to set the *DataFormat* property of each Column object, using the same dialog box used for individual bound controls. Typically, you use this tab to format numbers, currency values, dates, and times. You can also use a custom format, if needed. The settings on this tab are reflected in the *DataFormat* property of individual Column objects at run time. A few other properties of Column objects, which will be described later, are set on the Layout tab.

The Splits tab

If the grid is subdivided into two or more split areas, you can set the attributes for these areas in the Splits tab. You can't create new splits in this property page, but you act on the fields in this page to set each split's appearance and behavior.

To modify the attributes of a split, you have to select it in the upper drop-down list. If the grid isn't split, there will be only one item in the drop-down list, the *Split 0* item, and your setting will affect the entire grid control. You can set the *Locked* property to True to turn the DataGrid into a read-only control. The *AllowFocus* property determines whether the split can receive the focus (it's similar to the *TabStop* property of individual Visual Basic controls). The *AllowSizing* property determines whether the split can be interactively resized with the mouse at run time. If *AllowRowResizing* is True, the user can resize rows in this split by using the mouse. (Resize operations affect all the rows in all the splits because the DataGrid control doesn't support rows with different heights.) The *RecordSelectors* property determines whether there is a gray column for displaying record selectors on the left side of the split (or the whole grid).

You can control whether multiple splits vertically scroll together or independently of one another by using the *ScrollGroup* property of the Split object, which is an integer greater than or equal to 1. All the splits with the same value scroll together, so you can create splits that scroll independently by assigning different values to this property. The *ScrollBars* property affects the presence or absence of scroll bars in a particular split and takes one of the following values: 0-dbgNone, 1dbgHorizontal, 2-dbgVertical, 3-dbgBoth, and 4-dbgAutomatic. (The default is 4dbgAutomatic—show a scroll bar only if necessary.) If you have a group of Split objects that scroll together and the *ScrollBars* property of each is set to 4dbgAutomatic, only the rightmost split of the group will show a vertical scroll bar.

The *MarqueeStyle* property determines how the DataGrid control highlights the currently selected cell. This property can have one of the following values: 0dbgDottedCellBorder (a dotted border around the cell, also known as a focus rectangle, is used), 1-dbgSolidCellBorder (a solid border is used, which is usually more visible than a dotted border), 2-dbgHighlightCell (text and background color are inverted), 3-dbgHighlightRow (the entire row is highlighted—this is useful only when the grid or the split isn't editable), 4-dbgHighlightRowRaiseCell (similar to the previous one, but the current cell appears to be raised), 5-dbgNoMarquee (the current cell isn't highlighted in any way), or 6-dbgFloatingEditor (the default—the current cell is highlighted using a floating editor window with a blinking cursor, as in Microsoft Access).

The *AllowRowSizing*, *MarqueeStyle*, and *RecordSelectors* properties are exposed by the DataGrid control as well as its Split objects. Setting one of these properties for the DataGrid control has the same effect as setting the same property for all its Split objects.

The last two properties shown in the Splits tab work together to determine how many columns are visible in the split and whether they are resized to fit in the visible area. More precisely, the *Size* property can be assigned a numeric value whose meaning depends on the *SizeMode* property. If *SizeMode* is 0-dbgScalable, *Size* contains an integer that corresponds to the width of that split with respect to other scalable splits; for example, if you have two splits with *Size* = 1 and *Size* = 2, respectively, the first split will take one third of the grid's width and the second split will take the remaining two thirds. If *SizeMode* is 1-dbgExact, then *Size* is a floating-point number that corresponds to the split's exact width in twips; this setting ensures that the split always has the same width, whether other splits are added or removed.

The Layout tab

In the Layout tab, you can set column attributes on a split-by-split basis. The DataGrid control, in fact, allows you to display the same column with different attributes in different splits. For example, a column can be read-write in one split and read-only in another; or it can be invisible in some of the splits and visible in others. You set the read-only attribute with the *Locked* property and the visibility attribute with the *Visible* property. The *AllowSizing* Boolean property determines if the right border of the column can be dragged to resize the column's width. The *WrapText* Boolean property causes the text in the cell to wrap to the next row if necessary: You can use this property with the *RowHeight* property to produce multiline displays. The *Button* property, if set to True, causes a button for a drop-down menu to appear in the cell when it gets the focus. When the user clicks on this button, the DataGrid control receives a *ButtonClick* event, to which you typically react by dropping down a list of values using a standard ComboBox, a bound ListBox, or even another DataGrid control.

The *DividerStyle* property affects the style of the vertical line on the right border of a column and can be one of the following values: 0-dbgNoDividers, 1dbgBlackLine, 2-dbgDarkGrayLine (the default), 3-dbgRaised, 4-dbgInset, 5dbgUseForeColor, or 6-dbgLightGrayLine. The *Alignment* property sets the alignment of the contents of the column and can be 0-dbgLeft, 1-dbgRight, 2-dbgCenter, or 3-dbgGeneral. (By default, text is left-aligned and numbers are right-aligned.) The *Width* property specifies the width of each Column object, expressed in the units of the DataGrid's container.

Run-Time Operations

The DataGrid control is complex and is likely to demand some time from you before you're familiar with it. I'll outline the most common operations that you might want to perform on it, together with a few tricks to get the most out of this object.

Working with the current cell

The most important run-time properties of the DataGrid control are *Row* and *Col*, which set or return the position of the cell in the focus rectangle. The first row and the leftmost column return zero values. Once you make a given cell the current cell, you can retrieve and modify its contents using the DataGrid's *Text* property:

```
' Convert the current cell's contents to uppercase.
Private Sub cmdUppercase_Click()
    DataGrid1.Text = UCase$(DataGrid1.Text)
End Sub
```

The *EditActive* property returns True if the current cell is being edited and False otherwise; you can also assign a value to this property to enter or exit edit mode programmatically. When the edit mode is entered, a *ColEdit* event is triggered:

```
' Save the current cell value before editing.
Private Sub DataGrid1_ColEdit(ByVal ColIndex As Integer)
    ' SaveText is a module-level variable.
    SaveText = DataGrid1.Text
End Sub
```

You can determine whether the current cell has been modified by querying the *CurrentCellModified* property, and you can also set this property to False and then set *EditActive* to False to completely cancel the edit operation. The *CurrentCellVisible* property is exposed by both the DataGrid and Split objects; it returns True if the current cell is visible in the object. If you set a Split's *CurrentCellVisible* property to True, the Split scrolls until the cell becomes visible; if you set the DataGrid control's *CurrentCellVisible* property to True, all the splits scroll to make the cell visible. While the current cell is being edited, you can also read and modify the grid's *SelStart, SelLength,* and *SelText* properties, as you would do with a regular TextBox control.

Because the DataGrid control is always bound to an ADO data source, the *Bookmark* property, which sets or returns the bookmark to the current record, is often more useful than the *Row* property. Even more interesting, whenever the user moves to another row, the current record in the underlying Recordset object automatically changes to reflect the new current cell. Thus, you can retrieve additional fields from the Recordset by simply querying the Recordset's Fields collection. The following code assumes that the DataGrid control is bound to an ADO Data control:

```
' Display the current product's unit price in Euro currency.
' The RowColChange event fires when a new cell becomes current.
Private Sub DataGrid1_RowColChange(LastRow As Variant, _
    ByVal LastCol As Integer)
    ' The DOLLAR_TO_EURO_RATIO variable is defined elsewhere in the
module.
    lblEuroPrice = Adodc1.Recordset("UnitPrice") * DOLLAR_TO_EURO_RATIO
End Sub
```

The DataGrid control's *Split* property returns an integer in the range 0 through *Splits.Count-1*, which points to the split section that contains the current cell. You can also assign a new value to this property to move the focus to another split. When a grid is split into more sections, a few properties of the DataGrid control—such as *RecordSelectors* and *FirstRow*—are equivalent to the same properties exposed by the current split. In other words:

```
' The following statements are equivalent.
DataGrid1.RecordSelectors = True
DataGrid1.Splits(DataGrid1.Split).RecordSelectors = True
```

Accessing other cells

There are a few properties that let you retrieve and set the properties of any cell in the grid, but you have to use them in a way that isn't always intuitive. Each column object exposes the *Text* and *Value* properties: The former sets or returns the text displayed in the column for the current row, while the latter is the actual value in the column for the current row before it's formatted for display to the user. The Column object also exposes the *CellText* and *CellValue* methods, which return the contents of a cell in that column for any row, given its bookmark.

There are several ways to retrieve the bookmark relative to a row, as I'll show you in a moment.

VisibleRows and *VisibleCols* are read-only properties that return the number of visible rows and columns, respectively. There are no properties that directly return the total number of rows and columns. You can use the *ApproxCount* property, which returns the approximate number of rows; this number might differ from the actual value. To retrieve the number of columns, you must query the *Count* property of the Columns collection.

The DataGrid object exposes two methods that let you access the bookmark of any row in the control. *GetBookmark* returns a bookmark of a row relative to the current row: *GetBookmark(0)* is the same as the *Bookmark* property, *GetBookmark(-1)* is the bookmark of the row preceding the current row, *GetBookmark(1)* is the bookmark of the row following the current row, and so on. The other available method, *RowBookmark*, returns the bookmark of any visible row: *RowBookmark(0)* is the bookmark of the first visible row, and *RowBookmark(VisibleRows-1)* is the bookmark of the last visible row.

The bookmark of the first row is also returned by the *FirstRow* property. According to the documentation, you can assign a new bookmark to this property to programmatically scroll the grid's contents, but I found that I always get an "Invalid bookmark" error when I try to assign a value to it. The *LeftCol* property holds the index of the first visible column, so you can programmatically display the upper left corner of the grid using the code shown below.

```
DataGrid1.LeftCol = 0
Adodc1.Recordset.MoveFirst
DataGrid1.CurrentCellVisible = True
```

The *FirstRow*, *LeftCol*, and *CurrentCellVisible* properties are also exposed by the Split object; here, also, assigning a value to the *FirstRow* property without raising an error appears impossible.

You can use the value returned by any of the preceding bookmark methods as an argument of the Column object's *CellText* and *CellValue* methods, described previously. For example, this code displays the difference in the Total field between the current row and the row that precedes the current row:

```
Private Sub DataGrid1_RowColChange(LastRow As Variant, _
    ByVal LastCol As Integer)
    Dim gcol As MSDataGridLib.Column
    If DataGrid1.Row > 0 Then
        ' Get a reference to the current column.
        Set gcol = DataGrid1.Columns("Total")
        ' Display the difference between the values in the "Total"
column
        ' of the current row and the cell immediately above it.
        Label1 = gcol.CellValue(DataGrid1.GetBookmark(-1)) - gcol.Value
    Else
        Label1 = "(First Row)"
    End If
End Sub
```

Managing cell selections

Users can select any number of adjacent columns by clicking on the column headers while keeping the Shift key pressed; they can also select any number of rows—even nonadjacent ones—by clicking on the leftmost gray column while keeping the Ctrl key pressed. (Multiple row selection, therefore, requires that the grid's or the split's *RecordSelectors* property is set to True.) The *SelStartCol* and *SelEndCol* properties set and return the indices for the first and last selected columns, respectively. You can clear the column selection by setting these properties to -1, or by invoking the *ClearSelCols* method. These properties and this method are also exposed by the Split object.

Because the user can select nonadjacent rows, the system to determine which rows are currently highlighted is based on the DataGrid control's *SelBookmarks* collection, which contains the bookmarks of all the selected rows. For example, to select the current row, execute the following statement:

```
DataGrid1.SelBookmarks.Add DataGrid1.Bookmark
```

You can iterate on all the selected rows using a *For Each* loop. For example, the following code takes advantage of the *SelChange* event—which fires any time a column or a row is selected or deselected—to update a Label control with the sum of all the cells in the Total column for the rows that are currently selected:

```
Private Sub DataGrid1_SelChange(Cancel As Integer)
    Dim total As Single, bmark As Variant
    For Each bmark In DataGrid1.SelBookmarks
        total = total + DataGrid.Columns("Total").CellValue(bmark)
    Next
    lblGrandTotal = total
End Sub
```

There's no method that programmatically clears selected rows; you can do this only by removing all the items in the *SelBookmark* collection, as in the following code:

```
Do While DataGrid1.SelBookmarks.Count
    DataGrid1.SelBookmarks.Remove 0
Loop
```

Monitoring edit operations

The DataGrid control has a rich collection of events that let you trap nearly every user action. Almost all these events are in the form *Before*xxxx and *After*xxxx, where *Before*xxxx events receive a *Cancel* parameter that you can set to True to cancel the operation. We've already seen the *ColEdit* event, which fires whenever a value in a cell is edited by pressing a key. This event is actually preceded by the related *BeforeColEdit* event, which lets you selectively make a cell read-only:

```
' Refuse to edit a cell in the first column if it already contains a
value.
```

```
Private Sub DataGrid1_BeforeColEdit(ByVal ColIndex As Integer, _
    ByVal KeyAscii As Integer, Cancel As Integer)
    ' Note how you can test Null values and empty strings at the same
time.
    If ColIndex = 0 And DataGrid1.Columns(ColIndex).CellValue _
        (DataGrid1.Bookmark) & "" <> "" Then
        Cancel = True
    End If
End Sub
```

If you cancel the edit operation in the *BeforeColEdit* event, the control doesn't receive any other event for this operation, which might be disorienting if you're accustomed to the ADO way of raising events, where a postnotification event fires even if the code in the prenotification event cancels the operation. The *KeyAscii* parameter contains the code of the key pressed to enter edit mode, or 0 if the user entered edit mode with a click of the mouse. Because this parameter is passed by value, you can't alter it. This isn't a problem, however, because the grid also receives all the usual *KeyDown*, *KeyPress*, and *KeyUp* events, which let you modify the value of the parameter that contains the code for the key the user pressed.

Any time you modify a value in a cell, the DataGrid control receives a *Change* event; if the edit operation actually modifies the value in a cell—that is, if you don't cancel it with the Esc key—the control also receives the *BeforeColUpdate* and *AfterColUpdate* events:

```
Private Sub DataGrid1_BeforeColUpdate(ByVal ColIndex As Integer, _
    OldValue As Variant, Cancel As Integer)
    ' Trap invalid values here.
End Sub
```

But watch out for a quirk in the procedure. You can't access the value that is about to be entered in the grid by using the *Text* or *Value* properties of the DataGrid or the Column, because within this event procedure these properties return the value that was originally in the grid cell—that is, the same value returned by the *OldValue* parameter. It turns out that the DataGrid's *Text* property returns the string entered by the user only when the *EditActive* property is True, but this property has already been reset to False when processing the *BeforeColUpdate* event. The solution is to declare a form-level variable and assign it a value from within the *Change* event. For example, this code correctly checks that the value being entered isn't duplicated in any other record of the Recordset:

```
Dim newCellText As String

' Remember the most recent value entered by the user.
Private Sub DataGrid1_Change()
    newCellText = DataGrid1.Text
End Sub

' Check that the user isn't entering a duplicate value for that column.
Private Sub DataGrid1_BeforeColUpdate(ByVal ColIndex As Integer, _
    OldValue As Variant, Cancel As Integer)
    Dim rs As ADODB.Recordset, fldName As String
    ' Retrieve the field name for the current column.
    fldName = DataGrid1.Columns(ColIndex).DataField
    ' Search for the new value in the Recordset. Use a clone Recordset
    ' so that the current bookmark doesn't change.
    Set rs = Adodc1.Recordset.Clone
    rs.MoveFirst
    rs.Find fldName & "='" & newCellValue & "'"
    ' Cancel the operation if a match has been found.
    If Not rs.EOF Then Cancel = True
End Sub
```

When the user moves to another row, a pair of *BeforeUpdate* and *AfterUpdate* events fire, and you have an opportunity to perform record-level validation and optionally reject the update. Here's the complete sequence of events that fire when the user edits a value in a column and then moves to the next or previous grid row:

KeyDown	The user presses a key.
KeyPress	
BeforeColEdit	The grid enters edit mode.
ColEdit	
Change	Now you can read the new value using the Text property. Here the *ActiveEdit* property becomes True.
KeyUp	The first key is released.
KeyDown	Another key is pressed.
KeyPress	
Change	
KeyUp	
	Other keys are pressed.
BeforeColUpdate	The user moves to another column.
AfterColUpdate	
AfterColEdit	
RowColChange	This event fires only when the move is complete.
BeforeUpdate	The user moves to another row.
AfterUpdate	
RowColChange	This event fires only when the move is complete.

Performing insert and delete operations

The user can delete one or more rows by selecting them and then pressing the Delete key. This operation fires the *BeforeDelete* event (where you can cancel the command) and *AfterDelete* event, and then a *BeforeUpdate* and *AfterUpdate* pair of events. For example, you can write code in the *BeforeDelete* event procedure that checks whether the current record is the master record in a master-detail relationship, and either cancels the operation (as the following code illustrates) or automatically deletes all the related detail records.

```
Private Sub DataGrid1_BeforeDelete(Cancel As Integer)
    Dim rs As ADODB.Recordset, rsOrderDetails As ADODB.Recordset
    ' Get a reference to the underlying Recordset
    Set rs = Adodc1.Recordset
    ' Use the connection to perform a SELECT command that checks
whether
    ' there is at least one record in the Order Details table that has
    ' a foreign key that points to the ProductID value of current
record.
    Set rsOrderDetails = rs.ActiveConnection.Execute _
        ("Select * FROM [Order Details] WHERE [Order
Details].ProductID=" _
        & rs("ProductID"))
    ' If EOF = False, there is a match, so cancel the delete command.
    If Not rsOrderDetails.EOF Then Cancel = True
End Sub
```

If you cancel the delete command, the DataGrid control displays an error message. You can suppress this and other error messages from the control by trapping its *Error* event:

```
Private Sub DataGrid1_Error(ByVal DataError As Integer, _
    Response As Integer)
    ' DataError = 7011 means "Action canceled"
    If DataError = 7011 Then
        MsgBox "Unable to delete this record because there are " _
            & "records in the Order Details table that point to it."
        ' Cancel the standard error processing by setting Response = 0.
        Response = 0
    End If
End Sub
```

Upon entry into this event, the *DataError* parameter contains the error code, whereas the *Response* parameter contains 1; you can prevent the grid from displaying the standard error message by setting the *Response* parameter to 0, as the previous example demonstrates. You can also test the standard error message by means of the DataGrid's *ErrorText* property.

If the *AllowAddNew* property is True, the DataGrid control displays a blank row at its bottom, marked with an asterisk, and the user can enter a new row—and therefore a new record in the underlying recordset—simply by typing a character in one of the cells in this row. When this happens, the control fires a *BeforeInsert* event, immediately followed by an *AfterInsert* event (unless you cancel the command), and then an *OnAddNew* event. The exact event sequence is as follows:

BeforeInsert	The user clicks on the last row.
AfterInsert	
OnAddNew	
RowColChange	This event fires only when the move is complete.

BeforeColEdit	The user types a key.
ColEdit	
Change	
Other Change and Keyxxx events	
BeforeColUpdate	The user moves to another column on the same row.
AfterColUpdate	
AfterColEdit	
RowColChange	This event fires only when the move is complete. The user enters values in other cells on the same row.
BeforeUpdate	The user moves to another row.
AfterUpdate	
RowColChange	This event fires only when the move is complete.

You can monitor the current status using the *AddNewMode* property, which can be assigned one of the following values: 0-dbgNoAddNew (no AddNew command is in progress), 1-dbgAddNewCurrent (the current cell is on the last row, but no AddNew command is pending), 2-dbgAddNewPending (the current row is in the next-to-last row as a result of a pending AddNew command). An AddNew command can be initiated either by the user or by code, as the result of assignment to the *Text* or *Value* properties.

Trapping mouse events

The DataGrid control exposes all the usual mouse events, which are passed the mouse coordinates and the state of the shift keys. Unfortunately, the DataGrid control doesn't support OLE drag-and-drop operations, so you won't find the usual *OLE*xxxx properties, methods, and events. When working with the mouse, you're likely to use three methods exposed by the control: the *RowContaining* method, which returns the visible row over which the mouse cursor is located; the *ColContaining* method, which returns the corresponding column number; and finally the *SplitContaining* method, which returns the split number. If the mouse is outside the grid area—for example, when the mouse is over the record selectors area—these methods return -1. Here is an example that uses the *ToolTipText* property to display a ToolTip with the underlying value of the cell under the mouse, which can be especially useful if the column is too narrow to display longer strings:

```
Private Sub DataGrid1_MouseMove(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    Dim row As Long, col As Long
    On Error Resume Next
    row = DataGrid1.RowContaining(Y)
    col = DataGrid1.ColContaining(X)
    If row >= 0 And col >= 0 Then
        DataGrid1.ToolTipText = DataGrid1.Columns(col).CellValue _
            (DataGrid1.RowBookmark(row))
    Else
        DataGrid1.ToolTipText = ""
    End If
End Sub
```

Changing the grid layout

You can programmatically change the layout of a DataGrid control by using one of the many properties and methods of the Splits and Columns collections. For example, you can add a new column using the *Columns.Add* method, as follows:

```
' Add a Product Name column. (It will become the 4th column.)
With DataGrid1.Columns.Add(3)
    .Caption = "Product Name"
    .DataField = "ProductName"
End With
' You need to rebind the grid after adding a bound column.
DataGrid1.ReBind
```

You can also remove a column from the layout, using the *Columns.Remove* method:

```
' Remove the column added by the previous code snippet.
DataGrid1.Columns.Remove 3
```

Adding a split requires that you use the *Splits.Add* method. The argument you pass to this method is the position of the new split (0 for the leftmost split in the grid):

```
' Add a new split to the left of all existing splits.
DataGrid1.Splits.Add 0
```

After you create a split, you have to decide which columns are visible in it. Because each new split inherits all the columns from the grid, removing a column from one split would remove it from all the other splits. Rather than deleting unwanted columns, make them invisible, as illustrated by the following code:

```
' Add a new split to the right of the existing split.
With DataGrid1.Splits.Add(1)
    ' Ensure that the two splits divide the grid's width in half.
    ' Assumes that the existing split's SizeMode property is 0-
dbgScalable.
    ' (Always set SizeMode before Size!)
    .SizeMode = dbgScalable
    .Size = DataGrid1.Splits(0).Size
    ' This new split can be scrolled independently.
    .ScrollGroup = DataGrid1.Splits(0).ScrollGroup + 1
    ' Hide all the columns except the one labeled "ProductName".
    For Each gcol In .Columns
        gcol.Visible = (gcol.Caption = "ProductName")
    Next
End With
```

Dealing with lookup values

Often a value retrieved from a database table isn't meaningful in itself and is only useful because it's a foreign key to another table where the real information is. For example, the Products table in NWind.mdb includes a SupplierID field, which contains the value of a key in the Suppliers table, where you can find the name and the address of the supplier for that particular product. When you're displaying the Products table in a DataGrid control, you might use a suitable JOIN statement for the ADO Data control's *RecordSource* property so that the grid automatically displays the correct supplier name instead of its key.

The ADO binding mechanism, however, provides you with a better alternative. The trick is to declare a custom StdDataFormat object, assign it to the *DataFormat* property of a Column object, and then use the *Format* event to transform the numeric key values coming from the data source into more descriptive strings of text. The following routine loads all the values from the secondary table (also known as the *lookup table*) into a hidden ComboBox control. The routine then uses the contents of that control in the *Format* event of the custom StdDataFormat object to translate the SupplierID key into the supplier's CompanyName field:

```
Dim WithEvents SupplierFormat As StdDataFormat

Private Sub Form_Load()
    ' Load all the values from the Supplier lookup table into the
    ' hidden cboSuppliers ComboBox control.
    Dim rs As New ADODB.Recordset
    rs.Open "Suppliers", Adodc1.Recordset.ActiveConnection
    Do Until rs.EOF
        cboSuppliers.AddItem rs("CompanyName")
        ' The SupplierID value goes into the ItemData property.
        cboSuppliers.ItemData(cboSuppliers.NewIndex) = rs("SupplierID")
        rs.MoveNext
    Loop
    rs.Close

    ' Assign the custom format object to the SupplierID column.
    Set SupplierFormat = New StdDataFormat
    Set DataGrid1.Columns("SupplierID").DataFormat = SupplierFormat
    ' Make the row height equal to the ComboBox's height.
    DataGrid1.RowHeight = cboSuppliers.Height
End Sub

Private Sub SupplierFormat_Format(ByVal DataValue As _
    StdFormat.StdDataValue)
    Dim i As Long
    ' Search the key value in the cboSuppliers ComboBox.
    For i = 0 To cboSuppliers.ListCount - 1
        If cboSuppliers.ItemData(i) = DataValue Then
            DataValue = cboSuppliers.List(i)
            Exit For
        End If
    Next
End Sub
```

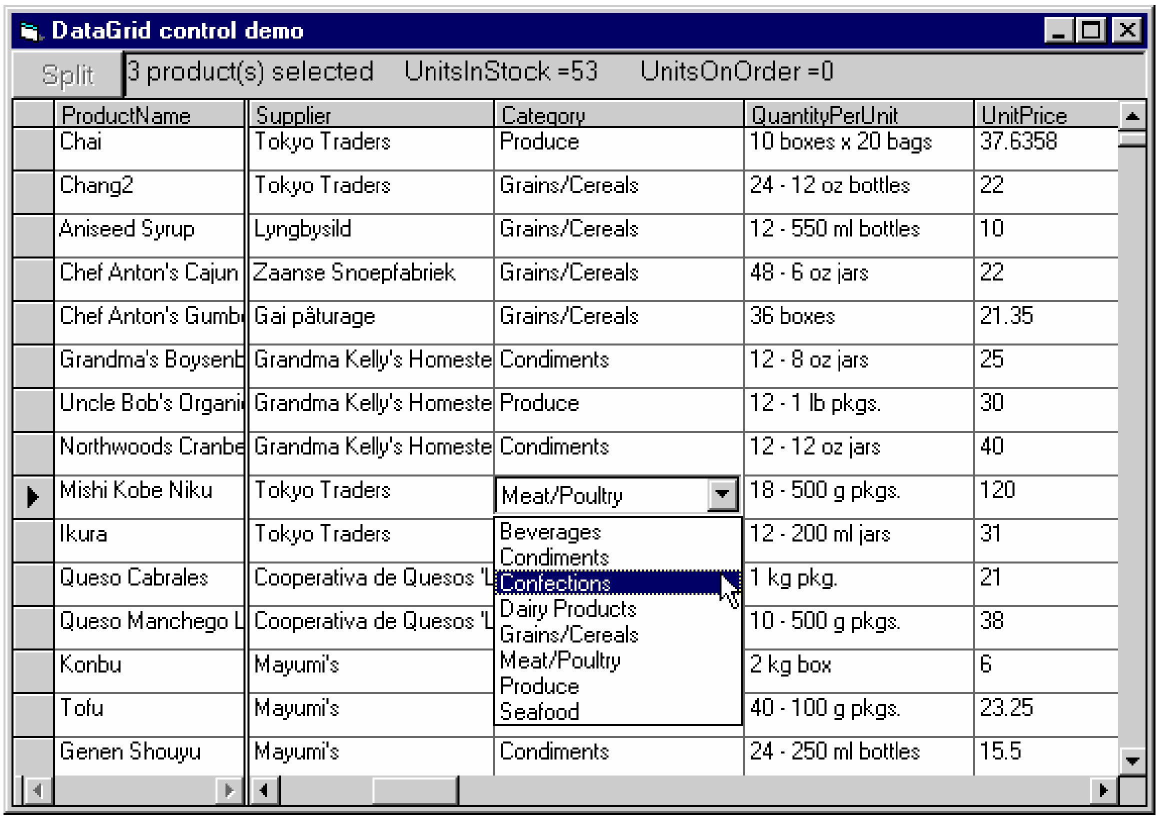
Using the ComboBox control as a repository for the contents of the lookup table isn't a casual decision. In fact, with some wizardry we can even use the ComboBox to let the user select a new value for the SupplierID field. All we have to do is make the ComboBox control appear in front of the DataGrid control, exactly over the cell edited by the user, and then update the underlying SupplierID field when the user selects a new value from the list. For the best visual effect, you also need to trap a few events so that the ComboBox is always in the correct position, as in Figure 15-4. Here's the code that does the trick:

```
Private Sub MoveCombo()
    ' In case of error, hide the ComboBox.
    On Error GoTo Error_Handler
    Dim gcol As MSDataGridLib.Column
    Set gcol = DataGrid1.Columns(DataGrid1.col)

    If gcol.Caption = "SupplierID" And DataGrid1.CurrentCellVisible
Then
        ' Move the ComboBox inside the SupplierID column
        ' if it is the current column and it is visible.
        cboSuppliers.Move DataGrid1.Left + gcol.Left, _
            DataGrid1.Top + DataGrid1.RowTop(DataGrid1.row), gcol.Width
        cboSuppliers.ZOrder
        cboSuppliers.SetFocus
        cboSuppliers.Text = gcol.Text
        Exit Sub
    End If
Error_Handler:
    ' In all other cases, hide the ComboBox.
    cboSuppliers.Move _10000
    If DataGrid1.Visible Then DataGrid1.SetFocus
End Sub

Private Sub cboSuppliers_Click()
    ' Change the value of the underlying grid cell.
    DataGrid1.Columns("SupplierID").Value = _
        cboSuppliers.ItemData(cboSuppliers.ListIndex)
End Sub

Private Sub DataGrid1_RowColChange(LastRow As Variant, _
    ByVal LastCol As Integer)
    MoveCombo
End Sub

Private Sub DataGrid1_RowResize(Cancel As Integer)
    MoveCombo
End Sub

Private Sub DataGrid1_ColResize(ByVal ColIndex As Integer, _
    Cancel As Integer)
    MoveCombo
End Sub

Private Sub DataGrid1_Scroll(Cancel As Integer)
    MoveCombo
End Sub

Private Sub DataGrid1_SplitChange()
    MoveCombo
End Sub
```

This code requires that the DataGrid control's *RowHeight* property match the ComboBox's *Height* property. Because the latter is read-only at run time, execute the following statement in the *Form_Load* event procedure:

```
' Have the row height match the ComboBox's height.
DataGrid1.RowHeight = cboSuppliers.Height
```

Another approach to lookup tables is based on the *Button* property of the Column object and the *ButtonClick* event. In this case, however, you get a better visual result if you display a ListBox (or DataList) control just under the current cell, rather than displaying a ComboBox or DataCombo control over the cell. Since the implementation of this latter method is similar to what I've shown previously, I leave it to you as an exercise.



3 product(s) selected   UnitsInStock =53   UnitsOnOrder =0

ProductName	Supplier	Category	QuantityPerUnit	UnitPrice
Chai	Tokyo Traders	Produce	10 boxes x 20 bags	37.6358
Chang2	Tokyo Traders	Grains/Cereals	24 - 12 oz bottles	22
Aniseed Syrup	Lyngbysild	Grains/Cereals	12 - 550 ml bottles	10
Chef Anton's Cajun	Zaanse Snoepfabriek	Grains/Cereals	48 - 6 oz jars	22
Chef Anton's Gumb	Gai pâturage	Grains/Cereals	36 boxes	21.35
Grandma's Boysenb	Grandma Kelly's Homeste	Condiments	12 - 8 oz jars	25
Uncle Bob's Organi	Grandma Kelly's Homeste	Produce	12 - 1 lb pkgs.	30
Northwoods Cranbe	Grandma Kelly's Homeste	Condiments	12 - 12 oz jars	40
Mishi Kobe Niku	Tokyo Traders	Meat/Poultry	18 - 500 g pkgs.	120
Ikura	Tokyo Traders		12 - 200 ml jars	31
Queso Cabrales	Cooperativa de Quesos 'L		1 kg pkg.	21
Queso Manchego L	Cooperativa de Quesos 'L		10 - 500 g pkgs.	38
Konbu	Mayumi's		2 kg box	6
Tofu	Mayumi's		40 - 100 g pkgs.	23.25
Genen Shouyu	Mayumi's	Condiments	24 - 250 ml bottles	15.5

Beverages
Condiments
Confections
Dairy Products
Grains/Cereals
Meat/Poultry
Produce
Seafood

Sorting data

The DataGrid control doesn't offer any built-in functionality for sorting data. However, thanks to its *HeadClick* event and the ADO Recordset's *Sort* property, sorting data is an easy task that requires only a handful of statements:

```
Private Sub DataGrid1_HeadClick(ByVal ColIndex As Integer)
    ' Sort on the clicked column.
    Dim rs As ADODB.Recordset
    Set rs = Adodc1.Recordset

    If rs.Sort <> DataGrid1.Columns(ColIndex).DataField & " ASC" Then
        ' Sort in ascending order; this block is executed if the
        ' data isn't sorted, is sorted on a different field,
        ' or is sorted in descending order.
        rs.Sort = DataGrid1.Columns(ColIndex).DataField & " ASC"
    Else
        ' Sort in descending order.
        rs.Sort = DataGrid1.Columns(ColIndex).DataField & " DESC"
    End If
    ' No need to refresh the contents of the DataGrid.
End Sub
```

The only limitation of this approach is that it doesn't work well if the column contains lookup values.