



Vardhaman Mahaveer Open University, Kota

Programming in Visual Basic

Course Development Committee

Chairman

Prof. (Dr.) Naresh Dadhich

Vice-Chancellor

Vardhaman Mahaveer Open University, Kota

Co-ordinator/Convener and Members

Convener:

Dr. Anuradha Sharma

Assistant Professor,

Department of Botany, Vardhaman Mahaveer Open University, Kota

Members:

- | | |
|---|--|
| 1. Prof. (Dr.) D.S.Chauhan
Department of Mathematics
University of Rajasthan, Jaipur | 3. Prof. (Dr.) A.K.Nagawat
University of Rajasthan,
Jaipur |
| 2. Prof. (Dr.) M.C.Govil
Govt. Engineering College,
Ajmer | 4. Dr. (Mrs.) Madhavi Sinha
Birla Institute of Technology,
Jaipur |
| 5. Dr. Rajeev Srivatava
LBS College,
Jaipur | |
-

Editing and Course Writing

Editor:

Dr. Rajeev Srivastava

Department of Computer Science

Lal Bahadur Shastri PG College , Jaipur

Unit Writers	Unit No.	Unit Writers	Unit No.
1. Dr. D.K. Gupta Department of Computer Science Modi Institute of Technology, Kota	(1,2,3,4)	4. Dr. Ajay Singh Department of Computer Science Thakur Jai Singh College, Kota	(10, 11)
2. Dr. Ritu Bhargava Department of Computer Science Govt. Women Polytechnic College, Ajmer	(5, 6)	5. Dr. Neeraj Bhargav Department of Computer Science M.D.S. University, Ajmer	(12, 13)
3. Dr. Vijay Singh Rathore Department of Computer Science Shri Karni College, Jaipur	(7, 8, 9)	6. Dr. Leena Bhatia Department of Computer Science Subodh P.G. College, Jaipur	(14,15)

Academic and Administrative Management

Prof. (Dr.) Naresh Dadhich

Vice-Chancellor

Vardhaman Mahaveer Open University,
Kota

Prof. B.K. Sharma

Director (Academic)

Vardhaman Mahaveer Open University,
Kota

Mr. Yogendra Goyal

Incharge

Material Production and
Distribution Department

Course Material Production

Mr. Yogendra Goyal

Assistant Production Officer

Vardhaman Mahaveer Open University, Kota



Vardhaman Mahaveer Open University, Kota

Programming in Visual Basic

Unit No.	Units	Page No.
1.	Introduction to Visual Basic	1 - 17
2.	Introduction to VB Controls	18 - 37
3.	VB Programming Fundamentals	38 - 52
4.	Arrays and Built in Functions	53 - 75
5.	Using Additional Controls and Control Array	76 - 89
6.	Menus	90 - 100
7.	Dialog Boxes and Mouse Events	101 - 125
8.	MDI Forms and Flexgrid Control	126 - 151
9.	Graphics and Error Handling in VB	152 - 169
10.	Connectivity with Datasbase using ADODC Control	170 - 177
11.	Connectivity with Database and Database Operations	178 - 188
12.	Object Linking and Embedding	189 - 203
13.	Object Oriented Programming in VB	204 - 216
14.	Crystal Reports	217 - 231
15.	Data Files	232 - 243
16.	References	244

Preface

We feel great in bringing out this material **“Programming in Visual Basic”** which meets the requirement of the students of BCA Part - II. This material is written entirely according to the syllabus of Vardhaman Mahaveer Open University, Kota

It covers the features of Visual Basic and provides the introduction to various controls of Visual Basic 6.0 and also provided the programming approach in the development. Useful tools like Dialog boxes. Combo Box, List box, arrays, and built in functions based on Date and time are also covered in details in easy way to provide better understanding of the students.

It also covers some advanced concepts of database connectivity, Error handling, MDI forms, databases and text files. Crystal Reports also gives the idea to students for presenting the reports for hard copy using databases and text files.

We have worked hard to make this course reader friendly. We shall be grateful for any suggestions for the improvement.

Unit - 1 : Introduction to Visual Basic

Structure of Unit:

- 1.0 Objectives
- 1.1 Introduction
- 1.2 Introduction to Graphical User Interface (GUI)
- 1.3 Oriented, Event Driven
- 1.4 Visual Basic Integrated Development Environment (IDE)
- 1.5 Editions of Visual Basic
- 1.6 Features and Advantages of Visual Basic
- 1.7 VB Compiler to Compile and Debug and Run the Programs
- 1.8 Summary
- 1.9 Self Assessment Questions

1.0 Objectives

After completing this unit you will be able to understand:

- Version of visual basic, Various parts of Visual basic IDE
- What is Graphical User Interface
- Features of Visual Basic.
- How to compile and debug visual basic program

1.1 Introduction

Visual Basic is easy to learn Programming language. With Visual Basic you can develop Windows based applications and games. Visual Basic is much more easier to learn than other language (like Visual C++), and yet it's powerful programming language.

Visual Basic suit more for application development than for games developing. You can create sophisticated games using Visual Basic, but If you want to make a really advanced professional game like Quake 2, You may choose other language (like C++), that would be much more harder to program with. However, Visual Basic will be probably powerful enough to suit all your application and games programming needs.

Microsoft Visual Basic development system version 6.0 is the most productive tool for creating high-performance components and applications. Visual Basic 6.0 offers developers the ability to create robust applications that reside on the client or server, or operate in a distributed n-tier environment. Visual Basic 6.0 is the Rapid Application Development (RAD) tool available either as a stand-alone product or as a part of the Visual Studio 6.0 suite of tools.

Hardware & software requirements

The following configuration is required to run VB:

1. Windows NT 4.0, Windows 95/98 operating system.
2. PC with an Intel Pentium processor
3. At least 32 MB of RAM (16 MB for Windows 95 or later)

4. Microsoft Internet Explorer version 4.01 or later.
5. Atleast 50MB of disk space

1.2 Introduction to Graphical User Interface (GUI)

VB is a Graphical User Interface (GUI) language. This means that a VB program will always show something on the screen that the user can interact with (usually via mouse and keyboard) to get a job done. The first step in building the VB program is to get the GUI items on the screen. This is done via pull-down menus that list the available graphical objects. Every system is slightly different but, generally speaking, left-clicking on an object allows you to describe attributes like size and position. Right clicking allows you to write code. For example, if the GUI item is a switch, left-clicking would allow the programmer to say how big the switch was, how it was labeled and where on the screen it is positioned. Right-clicking on the switch would bring up a window that allows the programmer to write the code that describes what happens when the user clicks the switch.

1.3 Oriented, Event Driven

It is considered good programming practice to modularize your programs. Instead of thinking of a computer program as a single large collection of code, the good programmer writes code so that you never need to look at more code than fits on the screen (or page) at one time. If we program in modules like this, the program is easier to understand and easy to update. Updating will likely be done by someone else so it is important, important that the program be easy to understand. Small (page size) modules where it is clearly indicated what comes into the module and what goes out makes a program easy to understand. VB forces to program in a modular fashion because each GUI item contains part of the code-the part that applies to that GUI item.

Object Orientation

Object Oriented Programming (OOP) is a concept where the programmer thinks of the program in “objects” (however abstract the objects may be) that interact with each other. In OOP, all the code associated with that object is in one place. Once again, VB forces this good programming practice. The GUI items are the objects and all the code associated with the object are just a click away. This natural way of enforcing good programming practices—plus the ease of programming in BASIC—is exactly why VB has found so many devoted fans.

Event driven

Event driven programming is a particular type of paradigm that functions as a result of some form of input. This input can be from somebody operating the human computer interface or it can also be influenced by messages and orders that are received from another computer program.

When you are using a computer, you are constantly provoking events that will lead to actions being performed by the computer. For example, when you are moving the computer mouse, the sensory input on the bottom of the device is telling the cursor on the screen to move. Likewise, if you were to click an icon on the screen, the clicking event is initiated as a result of that action.

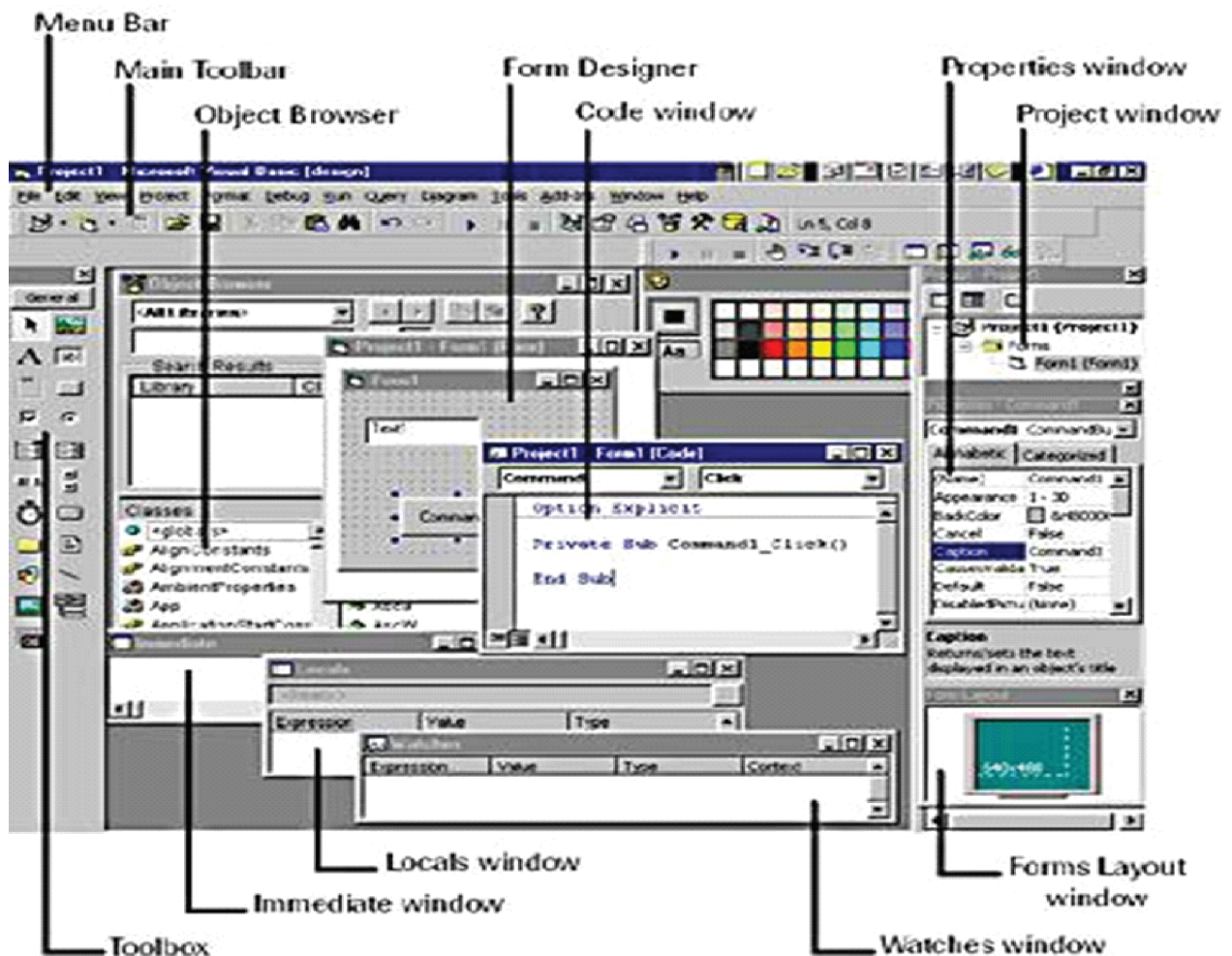
This sort of programming is very common in computer operating systems and graphical user interfaces. This means you are likely to come across event driven programming when you are playing a computer game or navigating your computer’s user interface. It makes use of input devices such as a mouse or a joystick, as well as giving users the ability to move around and interact with a system with relative ease.

In the past, computers could only be operated solely by entering the precise code required to initiate an action. This meant that a considerable amount of training was needed if you were to operate a computer correctly. However, thanks to the technological developments over time, event driven programming has made computers far easier to operate.

Event driven programming is greatly beneficial because of how user friendly it makes computer applications. It means that almost everyone can operate these systems without the need for expert knowledge on computer programming code such as SQL and Visual Basic.

1.4 Visual Basic Integrated Development Environment (IDE3)

One of the most significant changes in Visual Basic 6.0 is the Integrated Development Environment (IDE). IDE is a term commonly used in the programming World to describe the interface and environment that we use to create our applications. It is called integrated because we can access virtually all of the development tools that we need from one screen called an interface. The IDE is also commonly referred to as the design environment, or the program.



The above diagram shows the development environment with all the important points labeled. Many of Visual basic functions work similar to Microsoft word e.g.the Tool Bar and the tool box is similar to other products on the market which work off a single click then drag the width of the object required. The Tool Box contains the control you placed on the form window. All of the controls that appear on the Tool Box controls on the above picture never runs out of controls as soon as you place one on the form another awaits you on the tool box ready to be placed as needed.

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

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

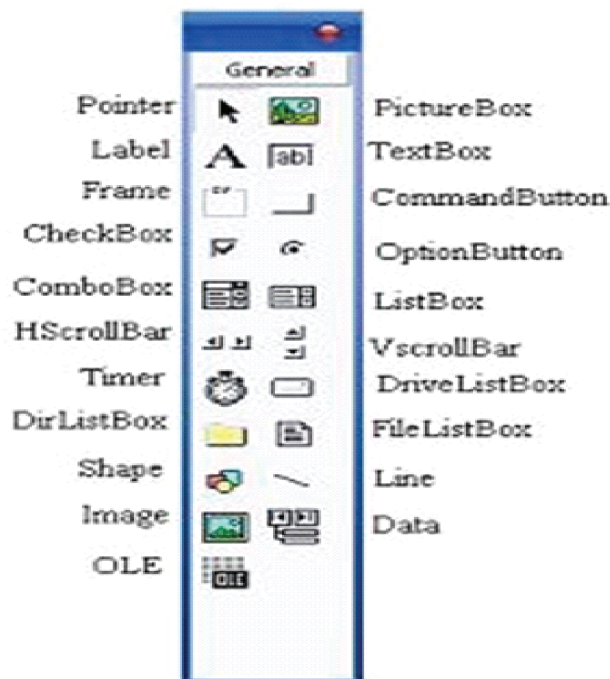
In previous versions of Visual Basic, the IDE was designed as a Single Document Interface (SDI). In a Single Document Interface, each window is a free-floating window that is contained within a main window and can move anywhere on the screen as long as Visual Basic is the current application. But, in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

Menu Bar

This Menu Bar displays the commands that are required to build an application. The main menu items have sub menu items that can be chosen when needed. The toolbars in the menu bar provide quick access to the commonly used commands and a button in the toolbar is clicked once to carry out the action represented by it.

Toolbox

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



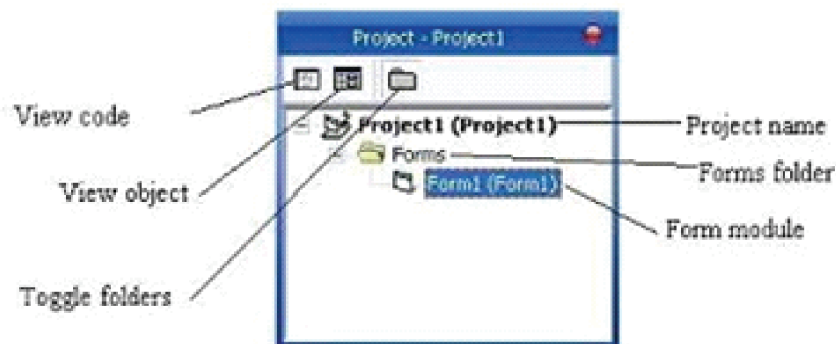
Control	Description
Pointer	Provides a way to move and resize the controls form
PictureBox	Displays icons/bitmaps and metafiles. It displays text or acts as a visual container for other controls.
TextBox	Used to display message and enter text.
Frame	Serves as a visual and functional container for controls
CommandButton	Used to carry out the specified action when the user chooses it.
CheckBox	Displays a True/False or Yes/No option.
OptionButton	OptionButton control which is a part of an option group allows the user to select only one option even it displays multiple choices.
ListBox	Displays a list of items from which a user can select one.
ComboBox	Contains a TextBox and a ListBox. This allows the user to select an item from the dropdown ListBox, or to type in a selection in the Text Box.
HScroll Bar VScrollBar	These controls allow the user to select a value within the specified in the range of values
Timer	Executes the timer events at specified intervals of time
DriveListBox	Displays the valid disk drives and allows the user to select one of them.
DirListBox	Allows the user to select the directories and paths, which are displayed.
FileListBox	Displays a set of files from which a user can select the desired one.
Shape	Used to add shape (rectangle, square or circle) to a Form
Line	Used to draw straight line to the Form
Image	used to display images such as icons, bitmaps and metafiles. But less capability than the PictureBox
Data	Enables the use to connect to an existing database and display information from it.
OLE	Used to link or embed an object, display and manipulate data from other windows based applications.

Project Explorer

Docked on the right side of the screen, just under the toolbar, is the Project Explorer window. The Project Explorer as shown in figure serves as a quick reference to the various elements of a project namely form, classes and modules. All of the object that make up the application are packed in a project. A simple project will typically contain one form, which is a window that is designed as part of a program's interface. It is

possible to develop any number of forms for use in a program, although a program may consist of a single form. In addition to forms, the Project Explorer window also lists code modules and classes.

Figure Project Explorer



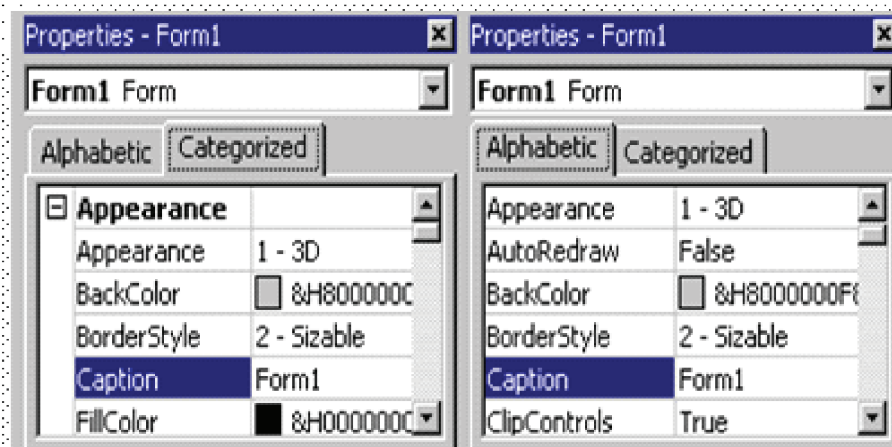
Properties Window

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

Object Browser

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

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



Properties Window

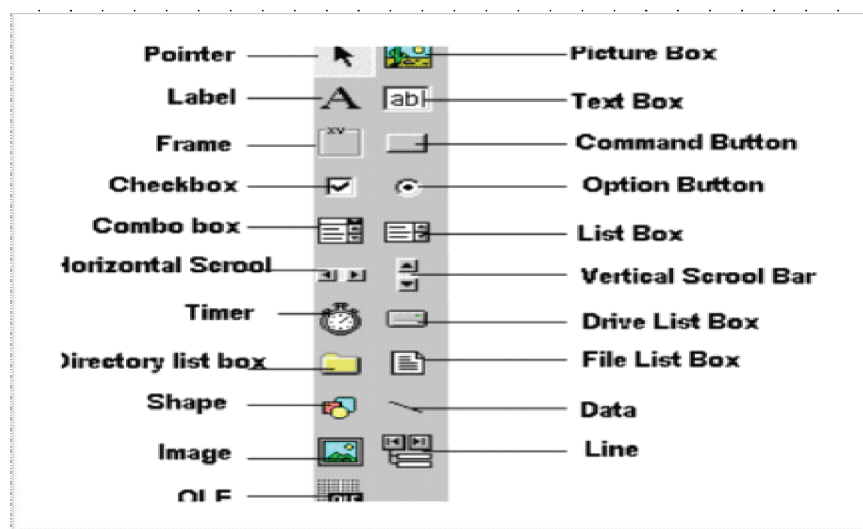
Some programmers prefer the Categorized view of the properties window. By defaulting, the properties window displays its properties alphabetically (with the exception of the name value) when you click on the categorized button the window changes to left picture.

The Default Layout

When we start Visual Basic, we are provided with a VB project. A VB project is a collection of the following modules and files.

- The global module(that contains declaration and procedures)
- The form module(that contains the graphic elements of the VB application along with the instruction)
- The general module (that generally contains general-purpose instructions not pertaining to anything graphic on-screen)
- The class module(that contains the defining characteristics of a class, including its properties and methods)
- The resource files(that allows you to collect all of the texts and bitmaps for an application in one place)
- On start up, Visual Basic will displays the following windows :
- The Blank Form window
- The Project window
- The Properties window

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.



Understanding the tool box.

Opening an existing Visual Basic project.

Microsoft have included some freebies with visual basic to show its capabilities and functions.

Dismantling or modifying these sample projects is a good way to understand what is happening at runtime. These files can be located at your default directory /SAMPLES/

To Open these projects choose 'Open Project' from the 'File' menu. Then Double click on the samples folder to open the directory then Double click on any project to load it.

Opening a new visual basic file & Inserting Source code.

From looking at the examples it time to make your own application. Choose 'New Project' from the 'File' menu. Use the blank form1 to design a simple interface for an estate agents database, have some textboxes for names and other details. Insert some controls and make it look professional. Textboxes can be used to store there name and other details, make sure you put a picture box in for a picture of the house.

Now insert the following source code for your application.

```
Private Sub Form_Load()
```

```
Picture1.Picture = LoadPicture("C:\Program Files\VB\Graphics\Icons\Misc\MISC42.ICO")
```

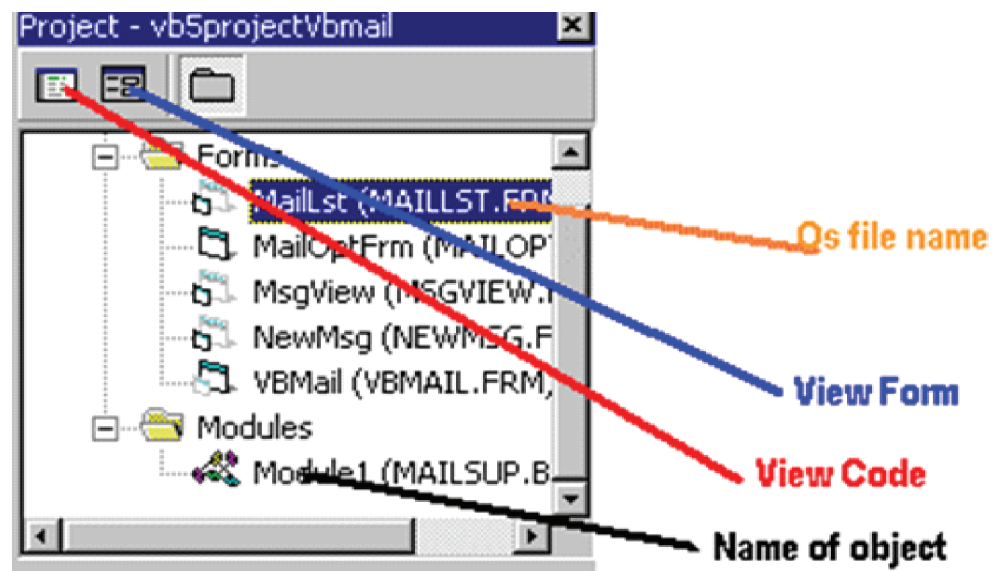
```
End Sub
```

Running and viewing the project in detail.

Once an application is loaded it can be run by click on the




Icon from the toolbar, to pause press and to terminate use



Once a project is loaded, the name of the form(s) that it contains is displayed in the project window. To view a form in design mode, select the form required by clicking with the mouse to highlight its name, then clicking on the view form button.

In this example the project has been loaded and the maillist.frm has been selected for viewing. This Ms Mail example project used 6 forms and 1 modules.

In Design mode, when the form is viewed, the code attached to any screen object may be inspected by double clicking on that object. The screen shots below show the interface of the Ms Mail example (.../samples/Comtool/VBMail/MaiLLST.FRM) to view the code for this form select  from the project window item.

```
Private Sub SetupOptionForm(BasePic As Control)
```

```
BasePic.Top = 0
```

```
BasePic.Left = 0
```

```
BasePic.Visible = True
```

```
BasePic.enabled = True
```

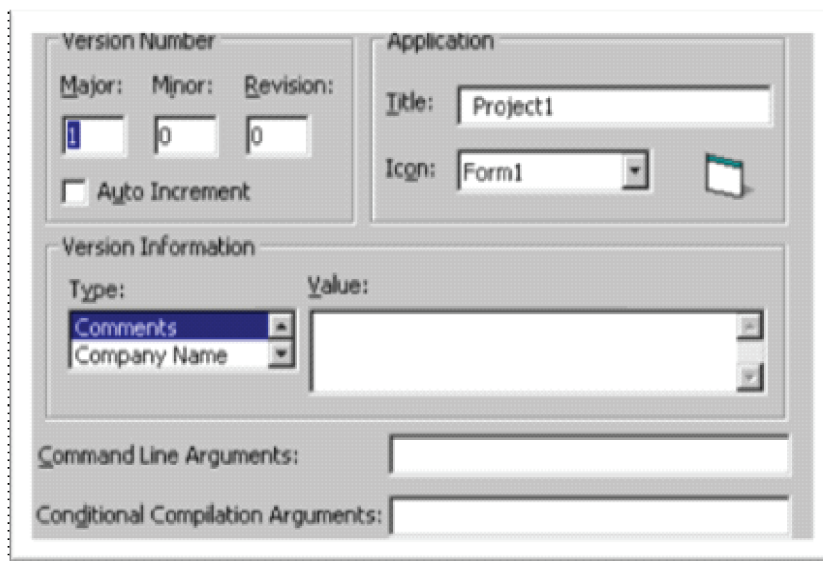
```
OKBt.Top = BasePic.Height + 120
```

```
Me.width = BasePic.Width + 120
```

```
Me.Heigh = OkBt.Top + OkBt.Height + 495
```

```
End Sub
```

Making your first *.exe



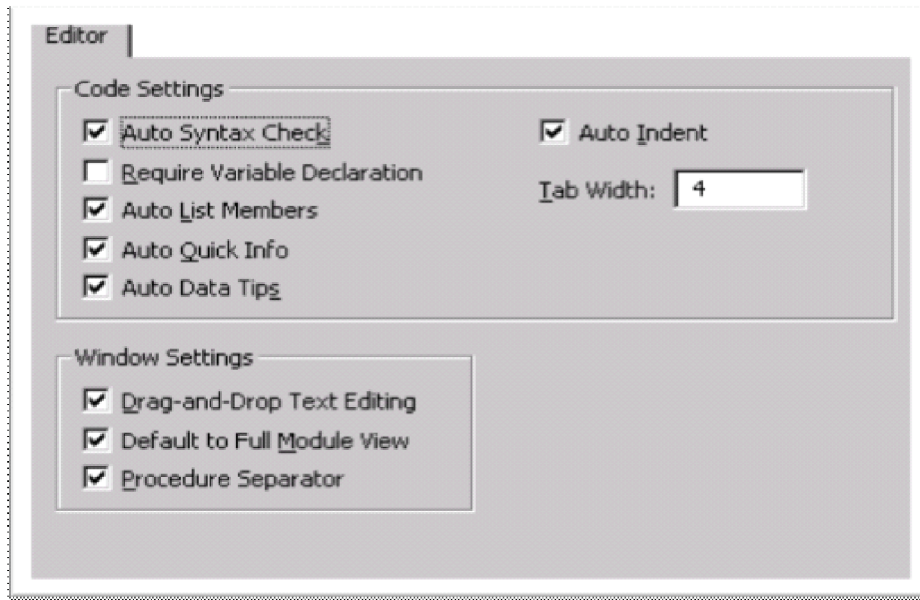
To make an executable from a project choose 'Make project.exe' from the 'File' menu. Then click once on the Make project.exe choose a default location to store your executable, you can also change some advanced options by clicking on the Options..tag before saving your exe .

The above image will be displayed in the comment's value type some comments company name name etc... The Title tag represents the caption you will see if you press Control + Alt + Del. And the icon is the icon that will be available on the execute icon. As you can see it is quite simple to understand. All the comments, data and name appear when you click on the compiled (execute) exe and click properties.

Saving your visual basic project.

Save your work to disk. Use the Windows Explorer or any desktop windows to check that all files have been saved. There should be one Visual Basic Project (.VBP) file and separate Form (.FRM) and Module (.BAS) files for each form and module used in the current project.

The Environment Editor Tab



Specifies the code window and project window settings.

Tab Options

Code Settings

- Auto Syntax Check — Determines whether Visual Basic should automatically verify correct syntax after you enter a line of code.
- Require Variable Declaration — Determines whether explicit variable declarations are required in modules. Selecting this adds the Option Explicit statement to general declarations in any new module.
- Auto List Members — Displays a box that displays information that would logically complete the statement at the current insertion point.
- Auto Quick Info — Displays information about functions and their parameters.
- Auto Data Tips — In the Code window while in Break mode, displays the value of the variable or object property over which your cursor is placed. The display is limited to variables and objects that are in the current scope.

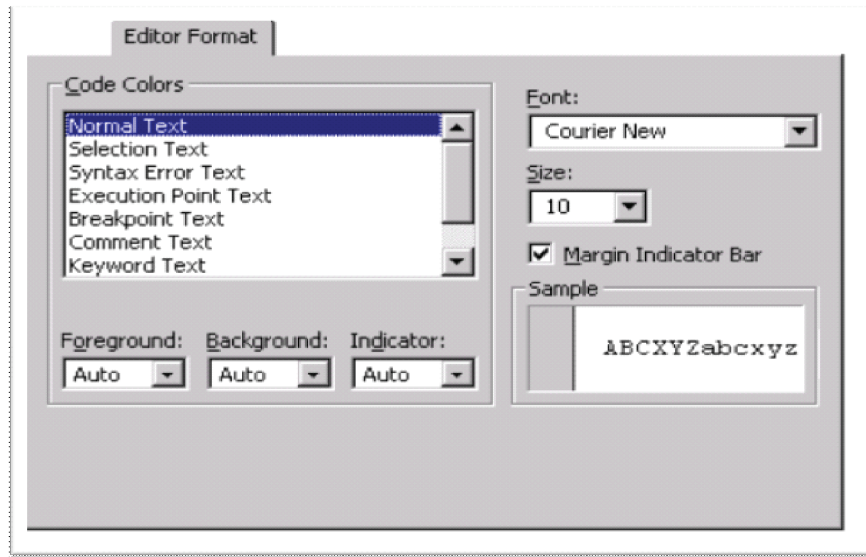
Data Tips are also available in the Immediate window while in Break mode. The values for object properties are displayed regardless of scope if a fully qualified object name is provided.

- Auto Indent — Allows you to tab the first line of code; all subsequent lines will start at that tab location.
- Tab Width — Sets the tab width, which can range from 1 to 32 spaces; the default is 4 spaces.

Window Settings

- Drag-and-Drop Text Editing — Allows you to drag and drop elements within the current code and from the Code window into the Immediate or Watch windows.
- Default to Full Module View — Sets the default state for new modules to allow you to look at procedures in the Code window either as a single scrollable list or one procedure at a time. It does not change the way currently open modules are viewed.

- **Procedure Separator** — Allows you to display or hide separator bars that appear at the end of each procedure in the Code window. Only available if Default to Full Module View is checked.



Editor Format Tab (Options)

Specifies the appearance of your Visual Basic code.

Tab Options

Code Colors Determines the foreground and background colors used for the type of text selected in the list box.

- **Text List** — Lists the text items that have customizable colors.
- **Foreground** — Specifies the foreground color for the text selected in the Color Text List.
- **Background** — Specifies the background color for text selected in the Color Text List.
- **Indicator** — Specifies the margin indicator color.

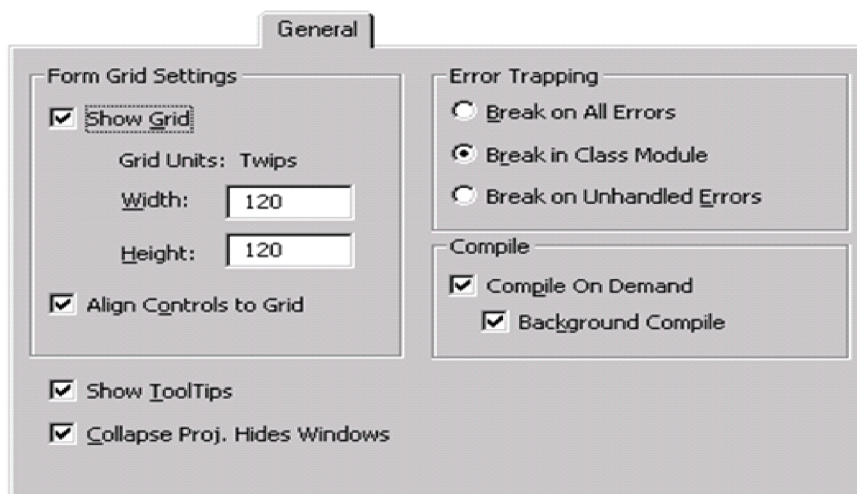
Font Specifies the font used for all code.

Size Specifies the size of the font used for code.

Margin Indicator Bar

Makes the margin indicator bar visible or not visible.

Sample Displays sample text for the font, size, and color settings.



General Tab

Specifies the settings, the error handling, and compile settings for your current Visual Basic project.

Tab Options

Form Grid Settings

Determines the appearance of the form grid at design time.

- **Show Grid** — Determines whether to show the grid at design time.
- **Grid Units** — Displays the grid units used for the form. The default is points.
- **Width** — Determines the width of grid cells on a form (2 to 60 points).
- **Height** — Determines the height of grid cells on a form (2 to 60 points).
- **Align Controls to Grid** — Automatically positions the outer edges of controls on grid lines.

Show ToolTips

Displays ToolTips for the toolbar and Toolbox items.

Collapse Proj. Hides Windows

Determines whether the window are hidden when a project is collapsed in the Project Explorer.

Error Trapping

Determines how errors are handled in the Visual Basic development environment and sets the default state of error trapping for all subsequent instances of Visual Basic. To set the error trapping option for only the current session of Visual Basic without changing the default for future sessions, use the Toggle command on the Code window's shortcut menu.

- **Break on All Errors** — Any error causes the project to enter break mode, whether or not an error handler is active and whether or not the code is in a class module.
- **Break in Class Module** — Any unhandled error produced in a class module causes the project to enter break mode at the line of code in the class module which produced the error.

When you debug an ActiveX component project by running an ActiveX client test program in another project, set this option in the ActiveX component project to break on errors in its class modules, instead of always returning the error to the client test program.

- **Break on Unhandled Errors** — If an error handler is active, the error is trapped without entering break mode. If there is no active error handler, the error causes the project to enter break mode. An unhandled error in a class module, however, causes the project to enter break mode on the line of code that invoked the offending procedure of the class.

Compile

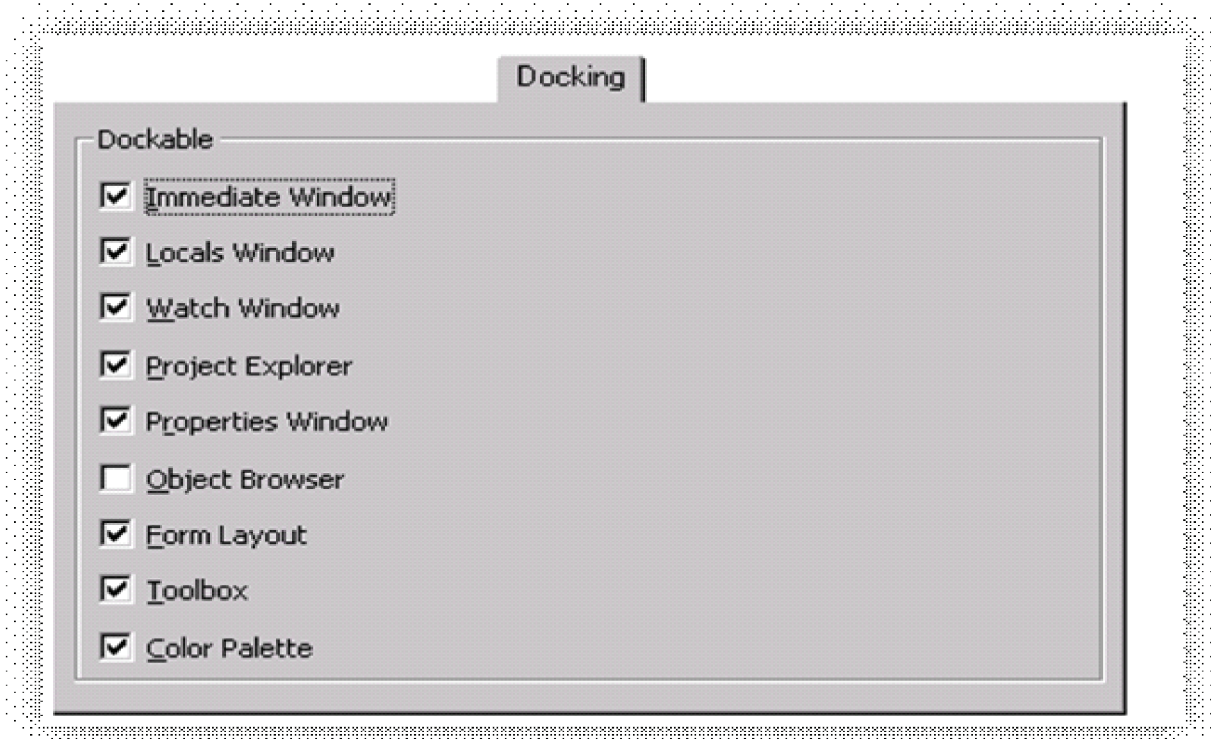
Determines how your project compiles.

- **Compile On Demand** — Determines whether a project is fully compiled before it starts, or whether code is compiled as needed, allowing the application to start sooner. If you choose the Start With Full Compile command on the Run menu, Visual Basic ignores the Compile on Demand setting and performs a full compile.

- **Background Compile** — Determines whether idle time is used during run time to finish compiling the project in the background. Background Compile can improve run time execution speed. This feature is not available unless Compile on Demand is also selected.

Docking Tab

Allows you to choose which windows you want to be dockable.



A window is docked when it is attached or “anchored” to other windows that are dockable or to the main window when you are in MDI mode. When you move a dockable window, it “snaps” to the location. A window is not dockable when you can move it anywhere on the screen and leave it there.

Tab Options

Dockable

Lists the windows that are dockable.

Select the windows you want to be dockable and clear those that you do not. You can have any, none, or all of the windows in the list dockable

1.5 Editions of Visual Basic

There are three editions of visual basic

1. Learning

Consists of all necessary tools required to build main stream Windows Applications

2. Professional

Includes advanced features such as tools to develop ActiveX and Internet controls.

3. Enterprise

In addition to all Professional features, it also includes tools such as Visual

SourceSafe and the Automation and Component Manger.

1.6 Features and Advantages of Visual Basic

Features of visual basic

- Form designer
- Create user Interface using design tools
- Add drawings to the forms
- Set properties for the individual object on the form
- Build an executable file
- Debug the applications
- Examine the objects in the forms
- Work with data in the programs (ODBC)
- Add more functionality
- Uses Crystal Report

Advantages of Visual Basic

There is quite a number of reasons for the enormous success of Visual Basic (VB):

- The structure of the Basic programming language is very simple, particularly as to the executable code.
- VB is not only a language but primarily an integrated, interactive development environment (“IDE”).
- The VB-IDE has been highly optimized to support rapid application development (“RAD”). It is particularly easy to develop graphical user interfaces and to connect them to handler functions provided by the application.
- The graphical user interface of the VB-IDE provides intuitively appealing views for the management of the program structure in the large and the various types of entities (classes, modules, procedures, forms, ...).
- VB provides a comprehensive interactive and context-sensitive online help system.
- When editing program texts the “IntelliSense” technology informs you in a little popup window about the types of constructs that may be entered at the current cursor location.
- VB is a component integration language which is attuned to Microsoft’s Component Object Model (“COM”).
- COM components can be written in different languages and then integrated using VB.
- Interfaces of COM components can be easily called remotely via Distributed COM (“DCOM”), which makes it easy to construct distributed applications.
- COM components can be embedded in / linked to your application’s user interface and also in/to stored documents (Object Linking and Embedding “OLE”, “Compound Documents”).

1.7 VB Compiler to Compile and Debug and Run the Programs

If you have the Professional or Enterprise edition of Visual Basic, you can compile your code either in standard Visual Basic p-code format or in native code format. Native code compilation provides several options for optimizing and debugging that aren't available with p-code.

P-code, or pseudo code, is an intermediate step between the high-level instructions in your Basic program and the low-level native code your computer's processor executes. At run time, Visual Basic translates each p-code statement to native code. By compiling directly to native code format, you eliminate the intermediate p-code step.

You can debug compiled native code using standard native code debugging tools, such as the debugging environment provided by Visual C++. You can also use options available in languages such as Visual C++ for optimizing and debugging native code. For example, you can optimize code for speed or for size.

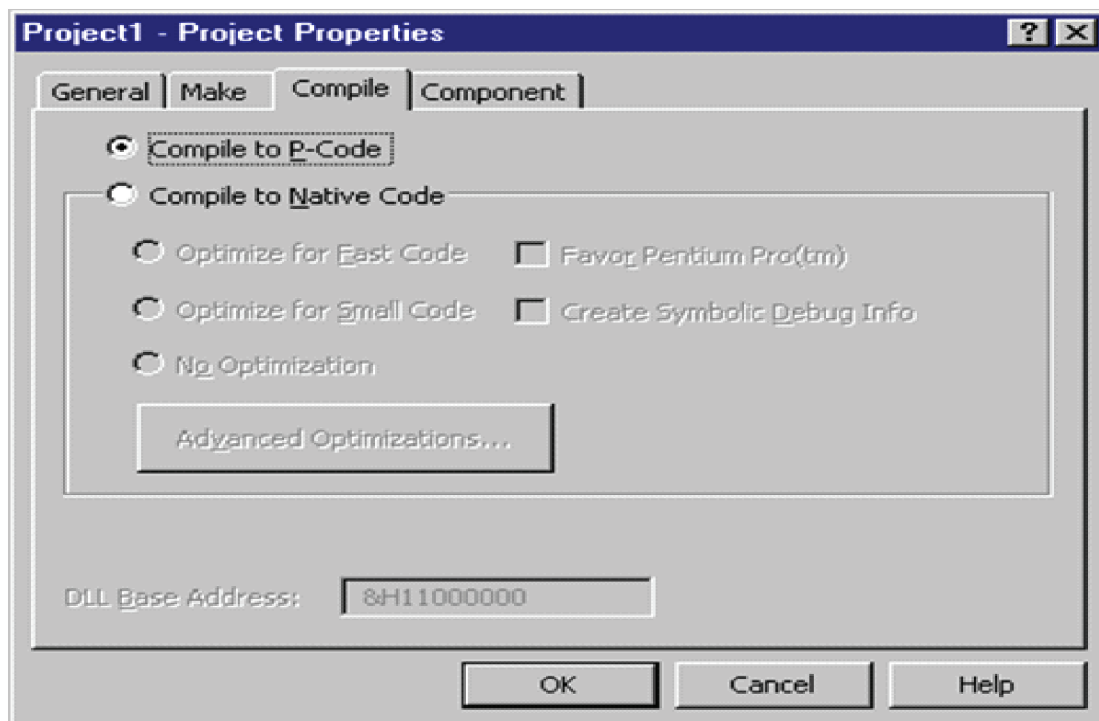
Note: All projects created with Visual Basic use the services of the run-time DLL (MSVBVM60.DLL). Among the services provided by this DLL are startup and shutdown code for your application, functionality for forms and intrinsic controls, and run-time functions like Format and CLng.

Compiling a project with the Native Code option means that the code you write will be fully compiled to the native instructions of the processor chip, instead of being compiled to p-code. This will greatly speed up loops and mathematical calculations, and may somewhat speed up calls to the services provided by MSVBVM60.DLL. However, it does not eliminate the need for the DLL.

To compile a project to native code

1. In the Project window, select the project you want to compile.
2. From the Project menu, choose Project Properties.
3. In the Project Properties dialog box, click the Compile tab.

Figure : The Compile tab in the Project Properties dialog box



4. Select Compile to Native Code.

Visual Basic enables several options for customizing and optimizing the executable file. For example, to create compiled code that will be optimized for size, select the Optimize for Small Code option.

For additional advanced optimization options, click the Advanced Optimizations button.

5. Select the options you want, then click OK.

6. From the File menu, choose Make Exe, or Make Project Group.

The following table describes the native code options for optimization.

Option	Description
Assume No Aliasing (Advanced Optimization)	Tells the compiler that your program does not use aliasing. Checking this option allows the compiler to apply optimization such as storing variables in registers and performing loop optimizations.
Create Symbolic Debug Info	Produces a .pdb file and .exe or .dll file containing information to allow for debugging using Microsoft Visual C++ 5.0 or another compatible debugger.
Favor Pentium Pro(tm)	Optimizes code to favor the Pentium Pro(tm) processor.
No Optimization	Disables all optimizations.
Optimize for Fast Code	Maximizes the speed of .exe and .dll files by telling the compiler to favor speed over size.
Optimize for Small Code	Minimizes the size of .exe and .dll files by telling the compiler to favor size over speed.
Remove Array Bounds Checks (Advanced Optimization)	Disables Visual Basic array bounds checking.
Remove Floating Point Error Checks (Advanced Optimization)	Disables Visual Basic floating-point error checking.
Remove Integer Overflow Checks (Advanced Optimization)	Disables Visual Basic integer overflow checking.
Remove Safe Pentium(tm) FDIV Checks (Advanced Optimization)	Disables checking for safe Pentium(tm) processor floating-point division.

1.8 Summary

- Visual Basic 6.0 offers developers the ability to create robust applications that reside on the client or server, or operate in a distributed n-tier environment.
- VB is a Graphical User Interface (GUI) language. This means that a VB program will always show something on the screen that the user can interact with (usually via mouse and keyboard) to get a job done.

- Object Oriented Programming (OOP) is a concept where the programmer thinks of the program in “objects” (however abstract the objects may be) that interact with each other.
- Event driven programming is a particular type of paradigm that functions as a result of some form of input.
- The Project explorer window gives you a tree-structured view of all the files inserted into the application. You can expand these and collapse branches of the views to get more or less detail (Project explorer).
- There are three editions of visual basic
 - a. **Learning**
 - b. **Professional**
 - c. **Enterprise**

1.9 Self Assessment Questions

1. What do you mean by GUI?
2. What is IDE? Explain it all components.
3. List Various features of visual basic.
4. List advantages of visual basic.
5. How will you compile visual basic program?

Unit - 2 : Introduction to VB Controls

Structure of Unit:

- 2.0 Objectives
- 2.1 Introduction to Toolbox
- 2.2 Object naming conventions
- 2.3 Setting properties
- 2.4 Methods and Events
- 2.5 Working with basic objects
 - 2.5.1 Forms
 - 2.5.2 Labels
 - 2.5.3 Textboxes
 - 2.5.4 Command buttons
 - 2.5.5 Option button
 - 2.5.6 Check box
 - 2.5.7 Image box
 - 2.5.8 Frame
- 2.6 Summary
- 2.7 SelfAssessment Questions

2.0 Objectives

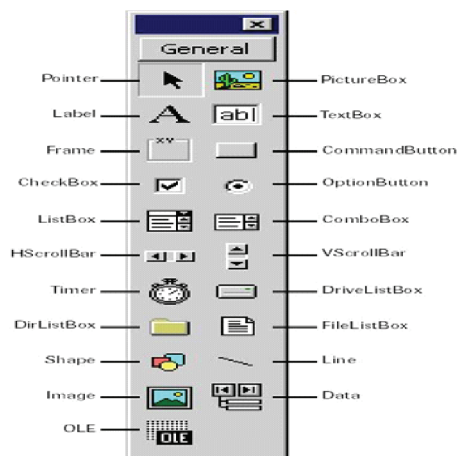
After completing this unit you will understand:

- Toolbox and its components, object naming conventions
- Properties, methods and events of form, textbox, label
- Imagebox, frame, check box controls.

2.1 Introduction to Toolbox

The Toolbox window is probably the first window you'll become familiar with because it lets you visually create the user interface for your applications. More specifically, the Toolbox contains the icons of all the intrinsic controls that is, all the controls that are included in the Visual Basic runtime.

If you have already programmed with a previous version of Visual Basic, you surely know the characteristics of all the controls that are present in the Toolbox. If you haven't, refer to below Figure while you read the following condensed descriptions.



- The Pointer isn't a control; click this icon when you want to select controls already on the form rather than create new ones.
- The PictureBox control is used to display images in any of the following formats: BMP, DIB (bitmap), ICO (icon), CUR (cursor), WMF (metafile), EMF (enhanced metafile), GIF, and JPEG.
- The Label control serves to display static text or text that shouldn't be edited by the user; it's often used to label other controls, such as TextBox controls.
- The TextBox control is a field that contains a string of characters that can be edited by the user. It can be single-line (for entering simple values) or multiline (for memos and longer notes). This is probably the most widely used control of any Windows application and is also one of the richest controls in terms of properties and events.
- The Frame control is typically used as a container for other controls. You rarely write code that reacts to events raised by this control.
- The CommandButton control is present in almost every form, often in the guise of the OK and Cancel buttons. You usually write code in the Click event procedure of this control.
- The CheckBox control is used when the user has to make a yes/no, true/false selection.
- OptionButton controls are always used in groups, and you can select only one control in the group at a time. When the user selects a control in the group, all other controls in the group are automatically deselected. OptionButton controls are useful for offering to the user a number of mutually exclusive selections. If you want to create two or more groups of OptionButton controls on a form, you must place each group inside another container control (most often a Frame control). Otherwise, Visual Basic can't understand which control belongs to which group.
- The ListBox control contains a number of items, and the user can select one or more of them (depending on the value of the control's MultiSelect property).
- The ComboBox control is a combination of a TextBox and a ListBox control, with the difference that the list portion is visible only if the user clicks on the down arrow to the right of the edit area. ComboBox controls don't support multiple selections.
- The HScrollBar and VScrollBar controls let you create stand-alone scroll bars. These controls are used infrequently because the majority of other controls display their own scroll bars if necessary. Stand-alone scroll bars are sometimes used as sliders, but in this case you'd better use other, more eye-catching controls, such as the Slider control.
- The Timer control is peculiar in that it isn't visible at run time. Its only purpose is to regularly raise an event in its parent form. By writing code in the corresponding event procedure, you can perform a task in the background for instance, updating a clock or checking the status of a peripheral device.
- The Drive ListBox, DirListBox, and FileListBox controls are often used together to create file-oriented dialog boxes. DriveListBox is a ComboBox-like control filled automatically with the names of all the drives in the system. DirListBox is a variant of the ListBox control; it shows all the subdirectories of a given directory. FileListBox is another special ListBox control; this control fills automatically with names of the files in a specified directory. While these three controls offer a lot of functionality, in a sense they have been superseded by the Common Dialog control, which displays a more modern user interface. If you want to write applications that closely conform to the Windows 9x look, you should avoid using these controls.

- The Shape and Line controls are mostly cosmetic controls that never raise any events and are used only to display lines, rectangles, circles, and ovals on forms or on other designers.
- The Image control is similar to the PictureBox control, but it can't act as a container for other controls and has other limitations as well. Nevertheless, you should use an Image control in place of a PictureBox control whenever possible because Image controls consume fewer system resources.
- The Data control is the key to data binding, a Visual Basic feature that lets you connect one or more controls on a form to fields in a database table. The Data control works with Jet databases even though you can also use attached tables to connect to data stored in databases stored in other formats. But it can't work with ActiveX Data Objects (ADO) sources and is therefore not suitable for exploiting the most interesting database-oriented Visual Basic 6 features.
- The OLE control can host windows belonging to external programs, such as a spreadsheet window generated by Microsoft Excel. In other words, you can make a window provided by another program appear as if it belongs to your Visual Basic application.

From this short description, you can see that not all the intrinsic controls are equally important. Some controls, such as the TextBox, Label, and CommandButton controls, are used in virtually every Visual Basic application, while other controls, such as the DriveListBox, DirListBox, and FileListBox controls, have been replaced, in practice, by newer controls. Similarly, you shouldn't use the Data control in any application that uses the ADO data sources.

2.2 Object naming conventions

One property that every control has and that's very important to Visual Basic programmers is the Name property. This is the string of characters that identifies the control in code. This property can't be an empty string, and you can't have two or more controls on a form with the same name. The special nature of this property is indirectly confirmed by the fact that it appears as (Name) in the Properties window, where the initial parenthesis serves to move it to the beginning of the property list.

When you create a control, Visual Basic assigns it a default name. For example, the first TextBox control that you place on the form is named Text1, the second one is named Text2, and so forth. Similarly, the first Label control is named Label1, and the first CommandButton control is named Command1. This default naming scheme frees you from having to invent a new, unique name each time you create a control. Notice that the Caption property of Label and CommandButton controls, as well as the Text property of TextBox controls, initially reflect the control's Name property, but the two properties are independent of each other. In fact, you have just modified the Caption and Text properties of the controls in the Rectangle Demo form without affecting their Name properties.

Because the Name property identifies the control in code, it's a good habit to modify it so that it conveys the meaning of the control itself. This is as important as selecting meaningful names for your variables. In a sense, most controls on a form are special variables whose contents are entered directly by the user.

Microsoft suggests that you always use the same three-letter prefix for all the controls of a given class. The control classes and their recommended prefixes are shown in below Table:

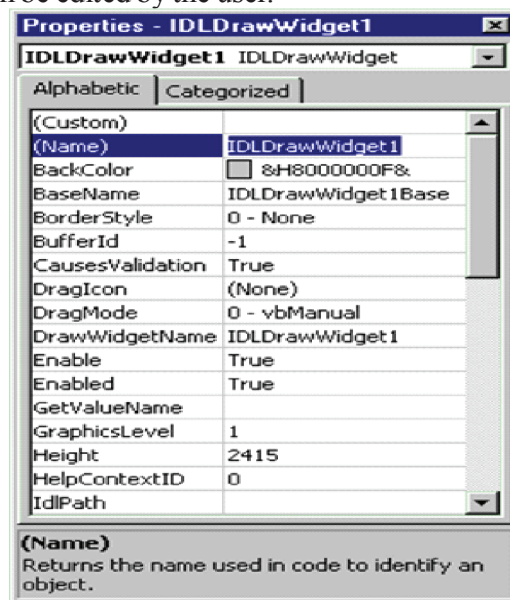
<i>Control Class</i>	<i>Prefix</i>	<i>Control Class</i>	<i>Prefix</i>
CommandButton	cmd	Data	dat
TextBox	txt	HScrollBar	hsb
Label	lbl	VScrollBar	vsb
PictureBox	pic	DriveListBox	drv
OptionButton	opt	DirListBox	dir
CheckBox	chk	FileListBox	fil
ComboBox	cbo	Line	lin
ListBox	lst	Shape	shp
Timer	tmr	OLE	ole
Frame	fra	Form	frm

For instance, you should prefix the name of a TextBox control with txt, the name of a Label control with lbl, and the name of a CommandButton control with cmd. Forms should also follow this convention, and the name of a form should be prefixed with the frm string. This convention makes a lot of sense because it lets you deduce both the control's type and meaning from its name. This tutorial sticks to this naming convention, especially for more complex examples when code readability is at stake.

In our example, we will rename the Text1 through Text4 controls as txtWidth, txtHeight, txtPerimeter, and txtArea respectively. The Command1 control will be renamed cmdEvaluate, and the four Label1 through Label4 controls will be renamed lblWidth, lblHeight, lblPerimeter, and lblArea, respectively. However, please note that Label controls are seldom referred to in code, so in most cases you can leave their names unmodified without affecting the code's readability.

2.3 Setting properties

Each control is characterized by a set of properties that define its behavior and appearance. For instance, Label controls expose a Caption property that corresponds to the character string displayed on the control itself, and a BorderStyle property that affects the appearance of a border around the label. The TextBox control's most important property is Text, which corresponds to the string of characters that appears within the control itself and that can be edited by the user.



In all cases, you can modify one or more properties of a control by selecting the control in the form designer and then pressing F4 to show the Properties window. You can scroll through the contents of the Properties window until the property you're interested in becomes visible. You can then select it and enter a new value.

Using this procedure, you can modify the Caption property of all four Label controls to &Width, &Height, &Perimeter, and &Area, respectively. You will note that the ampersand character doesn't appear on the control and that its effect is to underline the character that follows it. This operation actually creates a hot key and associates it with the control. When a control is associated with a hot key, the user can quickly move the focus to the control by pressing an Alt+x key combination, as you normally do within most Windows applications. Notice that only controls exposing a Caption property can be associated with a hot key. Such controls include the Label, Frame, CommandButton, OptionButton, and CheckBox.

Notice that once you have selected the Caption property for the first Label control, it stays selected when you then click on other controls. You can take advantage of this mechanism to change the Caption property of the CommandButton control to &Evaluate and the Caption property of the Form itself to Rectangle Demo, without having to select the Caption item in the Properties window each time. Note that ampersand characters within a form's caption don't have any special meaning.

As an exercise, let's change the font attributes used for the controls, which you do through the Font property. While you can perform this action on a control-by-control basis, it's much easier to select the group of controls that you want to affect and then modify their properties in a single operation. To select multiple controls, you can click on each one of them while you press either the Shift or the Ctrl key, or you can drag an imaginary rectangle around them. (This technique is also called lassoing the controls.)

When you select a group of controls and then press the F4 key, the Properties window displays only the properties that are common to all the selected controls. The only properties that are exposed by any control are Left, Top, Width, and Height. If you select a group of controls that display a string of characters, such as the TextBox, Label, and CommandButton controls in our Rectangle example, the Font property is also available and can therefore be selected. When you double-click on the Font item in the Properties window, a Font dialog box appears. Let's select a Tahoma font and set its size to 11 points.

Finally we must clear the Text property of each of the four Textbox controls so that the end user will find them empty when the program begins its execution. Oddly, when you select two or more Textbox controls, the Text property doesn't appear in the Properties window. Therefore, you must set the Text property to an empty string for each individual Textbox control on the form. To be honest, I don't know why this property is an exception to the rule stated earlier.

2.4 Methods and Events

Remember that a container is an object—such as a form or the Frame or Picture Box controls—that can contain other controls.

properties describe objects.

Methods cause an object to do something.

Events are what happens when an object does something

Every object, such as a form or control, has a set of properties that describe it. Although this set isn't identical for all objects, some properties—such as those listed in Table —are common to most controls. You can see every property for a given control by looking at the Properties window in the IDE.

Common Properties of Visual Basic Controls

<i>Property</i>	<i>Description</i>
Left	The position of the left side of a control with respect to its container
Top	The position of the top of a control with respect to its container
Height	A control's height
Width	A control's width
Name	The string value used to refer to a control
Enabled	The Boolean (True/False) value that determines whether users can manipulate the control
Visible	The Boolean (True/False) value that determines whether users can see the control

Another important property to consider is `BorderStyle`, which determines the window elements (title bar, Maximize and Minimize buttons, and so forth) a form will have.

Common Methods of Visual Basic Controls

<i>Method</i>	<i>Use</i>
Move	Changes an object's position in response to a code request
Drag	Handles the execution of a drag-and-drop operation by the user
SetFocus	Gives focus to the object specified in the method call
ZOrder	Determines the order in which multiple objects appear onscreen

Events are what happen in and around your program. For example, when a user clicks a button, many events occur: The mouse button is pressed, the `CommandButton` in your program is clicked, and then the mouse button is released. These three things correspond to the `MouseDown` event, the `Click` event, and the `MouseUp` event. During this process, the `GotFocus` event for the `CommandButton` and the `LostFocus` event for whichever object previously held the focus also occur.

Using `GotFocus` and `LostFocus`

The `GotFocus` and `LostFocus` events relate to most other events because they occur whenever a new control becomes active to receive user input. This makes `GotFocus` and `LostFocus` useful for data validation, the process of making sure that data is in the proper format for your program. Be careful, though! Improperly coding these two events can cause your program to begin an endless loop, which will cause your program to stop responding.

2.5 Working with basic objects

A control is an object that can be drawn on a Form object to enable or enhance user interaction with an application. Controls have properties that define aspects their appearance, such as position, size and colour, and aspects of their behavior, such as their response to the user input. They can respond to events initiated by the user or set off by the system. For instance, a code could be written in a CommandButton control's click event procedure that would load a file or display a result.

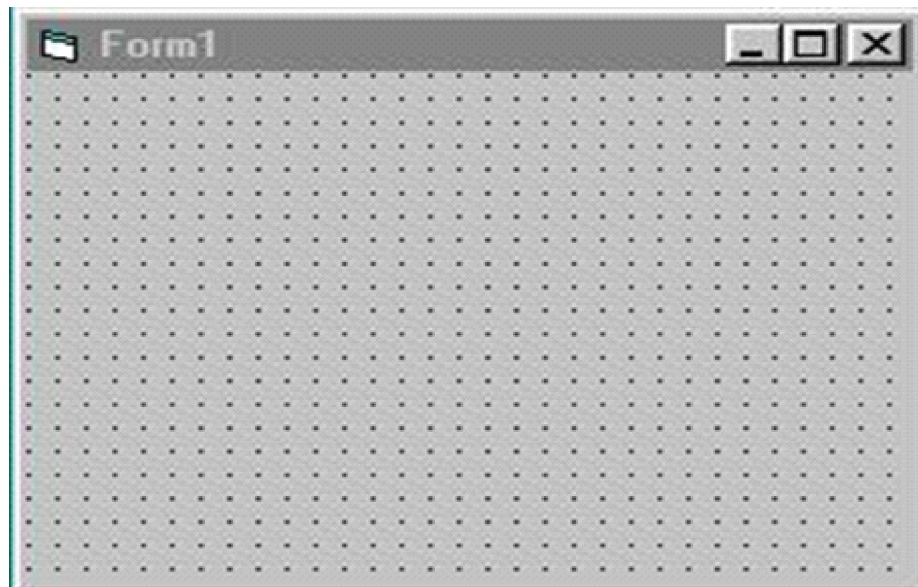
In addition to properties and events, methods can also be used to manipulate controls from code. For instance, the move method can be used with some controls to change their location and size.

2.5.1 Forms

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

What is a Form?

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



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

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

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

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

An MDI form is always contained within a parent form. This is exactly the same type of parent/child relationship which you see in Word. Each new Word document is contained in its own window, but is always framed within the larger window that is the Word application.

The MDI (multiple document interface) forms can be very useful in applications where multiple files/images/documents need to be open at the same time.

Properties / Events / Methods

Now is a good time to bring up the 3 categories of information which may be used to describe any object, including forms. Forms, like any object, have properties which you may set. The properties range from the caption that the form displays to the physical size of the form.

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

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

Given that there are over 20 controls available to you in the VB Pro edition, you might be concerned that learning all of the possible properties, events, and methods could be an overwhelming task. However, it's not at all that bad. Here's a very helpful piece of information that makes your task easier: all forms, controls, or objects share many of their properties, events, and methods!. Summary chart (available in Excel 97 and Excel 5.0 formats. The chart gives the complete list of VB controls (provided in VB Pro) and lists their properties, events, and methods. The chart is listed in such a way that you can see the common items, as well as those which are unique to that control. You'll see that many controls have no unique items at all! In chart then are 41 common properties, 20 common events, and 7 common methods. Please note that not every control uses all the common items! Some common items may be shared by only 2 or 3 controls.

It recommend that you look over the chart and become familiar with all of the items on it.

As a prelude to the larger chart, here's a simple listing of the entire set of properties, events, and

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

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

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

2.5.2 Labels

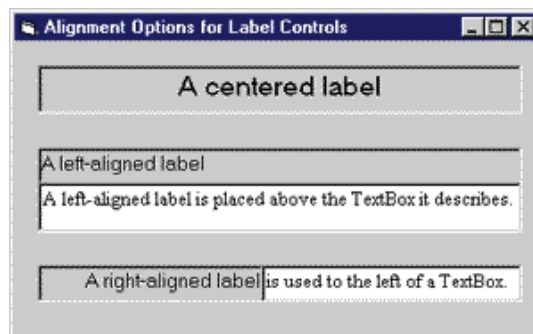
Label and Frame controls have a few features in common, so it makes sense to explain them together. First

they're mostly "decorative" controls that contribute to the user interface but are seldom used as programmable objects. In other words, you often place them on the form and arrange their properties as your user interface needs dictate, but you rarely write code to serve their events, generally, or manipulate their properties at run time.

Label Controls

Most people use Label controls to provide a descriptive caption and possibly an associated hot key for other controls, such as TextBox, ListBox, and ComboBox, that don't expose the Caption property. In most cases, you just place a Label control where you need it, set its Caption property to a suitable string (embedding an ampersand character in front of the hot key you want to assign), and you're done. Caption is the default property for Label controls. Be careful to set the Label's TabIndex property so that it's 1 minus the TabIndex property of the companion control.

Other useful properties are BorderStyle (if you want the Label control to appear inside a 3D border) and Alignment (if you want to align the caption to the right or center it on the control). In most cases, the alignment depends on how the Label control relates to its companion control: for example, if the Label control is placed to the left of its companion field, you might want to set its Alignment property to 1-Right Justify. The value 2-Center is especially useful for stand-alone Label controls.



Different settings for the Alignment property of Label controls.

You can insert a literal & character in a Label control's Caption property by doubling it. For example, to see Research & Development you have to type &Research && Development. Note that if you have multiple but isolated &s, the one that selects the hot key is the last one and all others are ignored. This tip applies to all the controls that expose a Caption property. (The & has no special meaning in forms' Caption properties, however.)

If the caption string is a long one, you might want to set the Label's WordWrap property to True so that it will extend for multiple lines instead of being truncated by the right border of the control. Alternatively, you might decide to set the AutoSize property to True and let the control automatically resize itself to accommodate longer caption strings.

You sometimes need to modify the default value of a Label's BackStyle property. Label controls usually cover what's already on the form's surface (other lightweight controls, output from graphic methods, and so on) because their background is considered to be opaque. If you want to show a character string somewhere on the form but at the same time you don't want to obscure underlying objects, set the BackStyle property to 0-Transparent.

If you're using the Label control to display data read from elsewhere—for example, a database field or a text file—you should set its UseMnemonics property to False. In this case, & characters have no special meaning to the control, and so you indirectly turn off the control's hot key capability. I mention this property because in older versions of Visual Basic, you had to manually double each & character to make the ampersand appear in text. I don't think all developers are aware that you can now treat ampersands like regular characters.

As I said before, you don't usually write code in Label control event procedures. This control exposes only a subset of the events supported by other controls. For example, because Label controls can never get the input focus, they don't support GotFocus, LostFocus, or any keyboard-related events. In practice, you can take advantage only of their mouse events: Click, DblClick, MouseDown, MouseMove, and MouseUp. If you're using a Label control to display data read from a database, you might sometimes find it useful to write code in its Change event. A Label control doesn't expose a specific event that tells programmers when users press its hot keys.

You can do some interesting tricks with Label controls. For example, you can use them to provide rectangular hot spots for images loaded onto the form. To create that context-sensitive ToolTip, the image is loaded on the form using the form's Picture property and then a Label control is placed over the Microsoft BackOffice logo, setting its Caption property to an empty string and the BackStyle property to 0-Transparent. These properties make the Label invisible, but it correctly shows its ToolTip when necessary. And because it still receives all mouse events, you can use its Click event to react to users' actions.

2.5.3 Textboxes

TextBox controls offer a natural way for users to enter a value in your program. For this reason, they tend to be the most frequently used controls in the majority of Windows applications. TextBox controls, which have a great many properties and events, are also among the most complex intrinsic controls. In this section, I guide you through the most useful properties of TextBox controls and show how to solve some of the problems that you're likely to encounter.

Setting properties to a TextBox

- Text can be entered into the text box by assigning the necessary string to the text property of the control
- If the user needs to display multiple lines of text in a TextBox, set the MultiLine property to True
- To customize the scroll bar combination on a TextBox, set the ScrollBars property.
- Scroll bars will always appear on the TextBox when it's MultiLine property is set to True and its ScrollBars property is set to anything except None(0)
- If you set the MultiLine property to True, you can set the alignment using the Alignment property. The text is left-justified by default. If the MultiLine property is set to False, then setting the Alignment property has no effect.

Run-Time Properties of a TextBox control

The Text property is the one you'll reference most often in code, and conveniently it's the default property for the TextBox control. Three other frequently used properties are these:

- **The SelStart property** sets or returns the position of the blinking caret (the insertion point where the text you type appears). Note that the blinking cursor inside TextBox and other controls is named caret, to distinguish it from the cursor (which is implicitly the mouse cursor). When the caret is at the beginning of the contents of the TextBox control, SelStart returns 0; when it's at the end of the string typed by the user, SelStart returns the value Len(Text). You can modify the SelStart property to programmatically move the caret.
- **The SelLength property** returns the number of characters in the portion of text that has been highlighted by the user, or it returns 0 if there's no highlighted text. You can assign a nonzero value to this property to programmatically select text from code. Interestingly, you can assign to this property a value larger than the current text's length without raising a run-time error.
- **The SelText property** sets or returns the portion of the text that's currently selected, or it returns an empty string if no text is highlighted. Use it to directly retrieve the highlighted text without having to query Text, SelStart, and SelLength properties. What's even more interesting is that you can assign a new value to this property, thus replacing the current selection with your own. If no text is currently selected, your string is simply inserted at the current caret position.

When you want to append text to a TextBox control, you should use the following code (instead of using the concatenation operator) to reduce flickering and improve performance:

```
Text1.SelStart = Len(Text1.Text)
```

```
Text1.SelText = StringToBeAdded
```

One of the typical operations you could find yourself performing with these properties is selecting the entire contents of a TextBox control. You often do it when the caret enters the field so that the user can quickly override the existing value with a new one, or start editing it by pressing any arrow key:

```
Private Sub Text1_GotFocus()
    Text1.SelStart = 0
    ' A very high value always does the trick.
    Text1.SelLength = 9999
End Sub
```

Always set the SelStart property first and then the SelLength or SelText properties. When you assign a new value to the SelStart property, the other two are automatically reset to 0 and an empty string respectively, thus overriding your previous settings.

The selected text can be copied to the Clipboard by using SelText:

```
Clipboard.SelText text, [format]
```

In the above syntax, text is the text that has to be placed into the Clipboard, and format has three possible values.

1. VbCFLink - conversation information
2. VbCFRTF - Rich Text Format
3. VbCFText - Text

We can get text from the clipboard using the GetText() function this way:

Clipboard.GetText ([format])

The following Figure summarizes the common TextBox control's properties and methods.

Property/ Method Description

Property/ Method	Description
<i>Properties</i>	
Enabled	specifies whether user can interact with this control or not
Index	Specifies the control array index
Locked	If this control is set to True user can use it else if this control is set to false the control cannot be used
MaxLength	Specifies the maximum number of characters to be input. Default value is set to 0 that means user can input any number of characters
MousePointer	Using this we can set the shape of the mouse pointer when over a TextBox
Multiline	By setting this property to True user can have more than one line in the TextBox
PasswordChar	This is to specify mask character to be displayed in the TextBox
ScrollBars	This to set either the vertical scrollbars or horizontal scrollbars to make appear in the TextBox. User can also set it to both vertical and horizontal. This property is used with the Multiline property.
Text	Specifies the text to be displayed in the TextBox at runtime
ToolTipIndex	This is used to display what text is displayed or in the control
Visible	By setting this user can make the Textbox control visible or invisible at runtime
<i>Method</i>	
SetFocus	Transfers focus to the TextBox
<i>Event procedures</i>	
Change	Action happens when the TextBox changes
Click	Action happens when the TextBox is clicked
GotFocus	Action happens when the TextBox receives the active focus
LostFocus	Action happens when the TextBox loses it focus
KeyDown	Called when a key is pressed while the TextBox has the focus
KeyUp	Called when a key is released while the TextBox has the focus

Trapping Keyboard Activity - Visual Basic 6 TextBox Control

TextBox controls support KeyDown, KeyPress, and KeyUp standard events. One thing that you will often do is prevent the user from entering invalid keys. A typical example of where this safeguard is needed is a numeric field, for which you need to filter out all nondigit keys:

```

Private Sub Text1_KeyPress(KeyAscii As Integer)

    Select Case KeyAscii

        Case Is < 32 ' Control keys are OK.

        Case 48 To 57 ' This is a digit.

        Case Else ' Reject any other key.

            KeyAscii = 0

        End Select

    End Sub

```

You should never reject keys whose ANSI code is less than 32, a group that includes important keys such as Backspace, Escape, Tab, and Enter. Also note that a few control keys will make your TextBox beep if it doesn't know what to do with them—for example, a single-line TextBox control doesn't know what to do with an Enter key.

Don't assume that the KeyPress event will trap all control keys under all conditions. For example, the KeyPress event can process the Enter key only if there's no CommandButton control on the form whose Default property is set to True. If the form has a default push button, the effect of pressing the Enter key is clicking on that button. Similarly, no Escape key goes through this event if there's a Cancel button on the form. Finally, the Tab control key is trapped by a KeyPress event only if there isn't any other control on the form whose TabStop property is True.

You can use the KeyDown event procedure to allow users to increase and decrease the current value using Up and Down arrow keys, as you see here:

```

Private Sub Text1_KeyDown(KeyCode As Integer, Shift As Integer)

    Select Case KeyCode

        Case vbKeyUp

            Text1.Text = CDbI(Text1.Text) + 1

        Case vbKeyDown

            Text1.Text = CDbI(Text1.Text) - 1

        End Select

    End Sub

```

There's a bug in the implementation of TextBox ready-only controls. When the Locked property is set to True, the Ctrl+C key combination doesn't correctly copy the selected text to the Clipboard, and you must manually implement this capability by writing code in the KeyPress event procedure

2.5.4 Command buttons

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


```

Private Sub Command1_Click()

    ' Save data, then unload the current form.

    Call SaveDataToDisk

    Unload Me

End Sub

```

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

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

This fires the button's Click event.

```
Command1.Value = True
```

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

Properties of a CommandButton control

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

```
Private Sub Command1_Click( )
```

```
.....
```


End Sub

2.5.5 Option button

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

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

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

```
If optWeekly.Value Then
    ' User prefers weekly frequency.

ElseIf optMonthly.Value Then
    ' User prefers monthly frequency.

ElseIf optYearly.Value Then
    ' User prefers yearly frequency.

End If
```

Strictly speaking, you can avoid the test for the last OptionButton control in its group because all choices are supposed to be mutually exclusive.

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

Example

Open a new Standard EXE project and save the Form as Option.frm and save the project as Option.vbp. Design the Form as per the following specifications table.

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

The application responds to the following events

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

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

Dim currentval as variant

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

```

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

```

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

```
Private Sub optOct_Click()  
    Text1.Text = Oct(currentval)  
End Sub  
  
Private Sub optHex_Click()  
    Text1.Text = Hex(currentval)  
End Sub  
  
Private Sub optDec_Click()  
    Text1.Text = Format(currentval)  
End Sub
```

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

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

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

2.5.6 Check box

The Check Box control lets the user select or unselect an option. When the Check Box is checked, its value is set to 1 and when it is unchecked, the value is set to 0. You can include the statements Check1.Value=1 to mark the Check Box and Check1.Value=0 to unmark the Check Box, as well as use them to initiate certain actions. For example, the program will change the background color of the form to red when the check box is unchecked and it will change to blue when the check box is checked. You will learn about the conditional statement If...Then...Elseif in later lesson. VbRed and vbBlue are color constants and BackColor is the background color property of the form.

Example

```
Private Sub Command1_Click()  
    If Check1.Value = 1 And Check2.Value = 0 Then  
        MsgBox "Apple is selected"  
    ElseIf Check2.Value = 1 And Check1.Value = 0 Then  
        MsgBox "Orange is selected"  
    Else
```

MsgBox “All are selected”

End If

End Sub

2.5.7 Imagebox

The Image Box is another control that handles images and pictures. It functions almost identically to the picture box. However, there is one major difference, the image in an Image Box is stretchable, which means it can be resized. This feature is not available in the Picture Box. Similar to the Picture Box, it can also use the LoadPicture method to load the picture. For example, the statement loads the picture grape.gif into the image box.

```
Image1.Picture=LoadPicture (“C:\VB program\Images\grape.gif”)
```

2.5.8 Frame

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

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

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

2.6 Summary

- Each control is characterized by a set of properties that define its behavior and appearance.
- The CommandButton control is present in almost every form, often in the guise of the OK and Cancel buttons. You usually write code in the Click event procedure of this control.
- The ListBox control contains a number of items, and the user can select one or more of them (depending on the value of the control’s MultiSelect property).
- The ComboBox control is a combination of a TextBox and a ListBox control, with the difference that the list portion is visible only if the user clicks on the down arrow to the right of the edit area. ComboBox controls don’t support multiple selections.
- The TextBox control is a field that contains a string of characters that can be edited by the user. It can be single-line (for entering simple values) or multiline (for memos and longer notes).

- The Image Box is another control that handles images and pictures. It functions almost identically to the picture box.

2.7 Self Assessment Questions

1. What are the different properties method and events of the form?
2. What is the difference between picture box and imagebox?
3. What is the difference between option box and check box?
4. What is the difference between list box and combo box?
5. Explain seltext and sellength and selstart properties of Textbox?

Unit – 3 : VB Programming Fundamentals

Structure of Unit:

- 3.0 Objectives
- 3.1 Introduction
- 3.2 Data types in VB
- 3.3 Variables and Declaration
- 3.4 Scope of variables
- 3.5 Operators in VB
- 3.6 Sub procedures and functions
- 3.7 Control structures
 - 3.7.1 IF Else Endif
 - 3.7.2 Select ..case,
 - 3.7.3 Do while ...loop,
 - 3.7.4 Do ... loop while
 - 3.7.5 Do ... loop until
 - 3.7.6 For ..Next,
 - 3.7.7 Exit For and Exit Do Statement in Visual basic
- 3.8 Summary
- 3.9 SelfAssessment Questions

3.0 Objectives

After completing this unit you will understand:

- Various data type available in visual basic.
- Operators in visual basic and variable type and scope.
- Implicit and explicit Variable declaration.
- Various control structure in visual basic.
- Procedure and function in visual basic

3.1 Introduction

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

3.2 Data types in VB

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

1. Numeric

Byte	Store integer values in the range of 0 - 255
Integer	Store integer values in the range of (-32,768) - (+ 32,767)
Long	Store integer values in the range of (- 2,147,483,468) - (+ 2,147,483,468)
Single	Store floating point value in the range of (-3.4x10 ⁻³⁸) - (+ 3.4x10 ³⁸)
Double	Store large floating value which exceeding the single data type value
Currency	store monetary values. It supports 4 digits to the right of decimal point and 15 digits to the left

2. String

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

3. Date

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

4. Boolean

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

5. Variant

Stores any type of data and is the default Visual Basic data type. In Visual Basic if we declare a variable without any data type by default the data type is assigned as default.

3.3 Variables and Declaration

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

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

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

Explicit Declaration

Declaring a variable tells Visual Basic to reserve space in memory. It is not must that a variable should be declared before using it. Automatically whenever Visual Basic encounters a new variable, it assigns the

default variable type and value. This is called implicit declaration. Though this type of declaration is easier for the user, to have more control over the variables, it is advisable to declare them explicitly. The variables are declared with a Dim statement to name the variable and its type. The As type clause in the Dim statement allows to define the data type or object type of the variable. This is called explicit declaration.

Syntax

Dim variable [As Type]

For example,

Dim strName As String

Dim intCounter As Integer

Using Option Explicit statement

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

Intcount = Intcount + 1

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

In Visual Basic, to prevent errors of this nature, we can declare a variable by adding the following statement to the general declaration section of the Form.

Option Explicit

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

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

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

3.4 Scope of variables

A variable is scoped to a procedure-level (local) or module-level variable depending on how it is declared. The scope of a variable, procedure or object determines which part of the code in our application is aware of the variable's existence. A variable is declared in general declaration section of a Form, and hence is available to all the procedures.

Local variables are recognized only in the procedure in which they are declared. They can be declared with Dim and Static keywords. If we want a variable to be available to all of the procedures within the same module, or to all the procedures in an application, a variable is declared with broader scope.

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

```
Dim sum As Integer
```

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

Static Variables

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

```
Static intPermanent As Integer
```

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

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

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

```
Static Function RunningTotal ()
```

Example

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

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

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

Module Level Variables

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

‘ In the declarative section of any module

Private LoginTime As Date ‘ A private module-level variable

Dim LoginPassword As String ‘ Another private module-level variable

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

‘ In the declarative section of Form1 module

Public CustomerName As String ‘ A Public property

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

‘ From outside Form1 module...

Form1.CustomerName = “John Smith”

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

Public Vs Local Variables

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

3.5 Operators in VB

Arithmetical Operators

Operators	Description	Example	Result
+	Add	5+5	10
-	Subtract	10-5	5
/	Divide	25/5	5
\	Integer Division	20\3	6
*	Multiply	5*4	20
^	Exponent (power of)	3^3	27
Mod	Remainder of division	20 Mod 6	2
&	String concatenation	"George"&" &"Bush"	"George Bush"

Relational Operator

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

Logical Operator

Operators	Description
OR	Operation will be true if either of the operands is true
AND	Operation will be true only if both the operands are true

3.6 Sub procedures and functions

Visual Basic offers different types of procedures to execute small sections of coding in applications. The various procedures are elucidated in details in this section. Visual Basic programs can be broken into smaller logical components called Procedures. Procedures are useful for condensing repeated operations

such as the frequently used calculations, text and control manipulation etc. The benefits of using procedures in programming are:

It is easier to debug a program a program with procedures, which breaks a program into discrete logical limits.

Procedures used in one program can act as building blocks for other programs with slight modifications.

A Procedure can be Sub, Function or Property Procedure.

Sub Procedures

A sub procedure can be placed in standard, class and form modules. Each time the procedure is called, the statements between Sub and End Sub are executed. The syntax for a sub procedure is as follows:

```
[Private | Public] [Static] Sub Procedurename [( arglist)]  
[ statements]  
End Sub
```

arglist is a list of argument names separated by commas. Each argument acts like a variable in the procedure. There are two types of Sub Procedures namely general procedures and event procedures.

Event Procedures

An event procedure is a procedure block that contains the control's actual name, an underscore(_), and the event name. The following syntax represents the event procedure for a Form_Load event.

```
Private Sub Form_Load()  
....statement block..  
End Sub
```

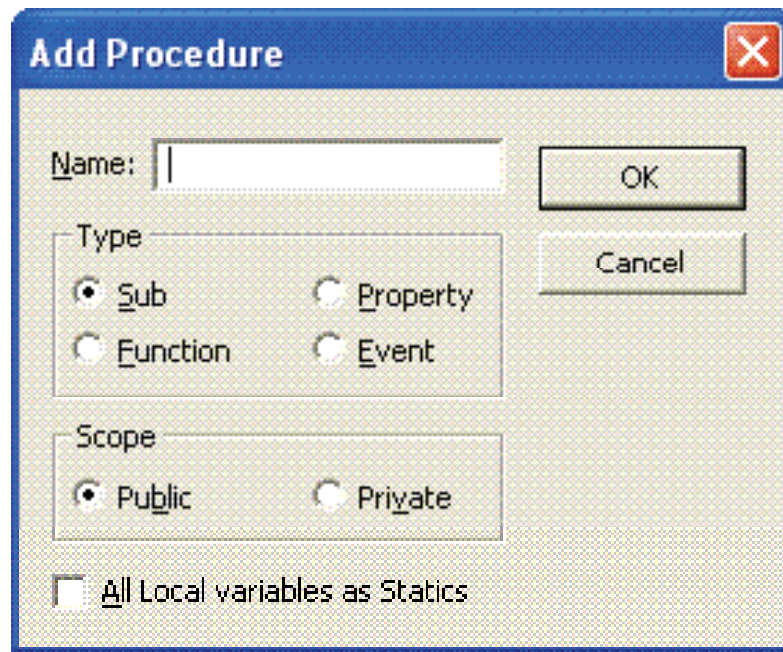
Event Procedures acquire the declarations as Private by default.

General Procedures

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

In order to add General procedure:

- The Code window is opened for the module to which the procedure is to be added.
- The Add Procedure option is chosen from the Tools menu, which opens an Add Procedure dialog box as shown in the figure given below.
- The name of the procedure is typed in the Name textbox
- Under Type, Sub is selected to create a Sub procedure, Function to create a Function procedure or Property to create a Property procedure.
- Under Scope, Public is selected to create a procedure that can be invoked outside the module, or Private to create a procedure that can be invoked only from within the module.



We can also create a new procedure in the current module by typing `Sub ProcedureName`, `Function ProcedureName`, or `Property ProcedureName` in the Code window. A Function procedure returns a value and a Sub Procedure does not return a value.

Function Procedures

Functions are like sub procedures, except they return a value to the calling procedure. They are especially useful for taking one or more pieces of data, called arguments and performing some tasks with them. Then the functions return a value that indicates the results of the tasks complete within the function.

The following function procedure calculates the third side or hypotenuse of a right triangle, where A and B are the other two sides. It takes two arguments A and B (of data type Double) and finally returns the results.

Function Hypotenuse (A As Double, B As Double) As Double

Hypotenuse = sqr (A^2 + B^2)

End Function

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

Property Procedures

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

3.7 Control structures

Control Statements are used to control the flow of program's execution. Visual Basic supports control structures such as `if... Then`, `if... Then ...Else`, `Select...Case`, and Loop structures such as `Do While...Loop`, `While...Wend`, `For...Next` etc method.

3.7.1 IF Else Endif

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

Syntax of the If...Then selection

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

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

If...Then...Else selection structure

The If...Then...Else selection structure allows the programmer to specify that a different action is to be performed when the condition is True than when the condition is False.

Syntax of the If...Then...Else selection

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

Nested If...Then...Else selection structure

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

Syntax of the Nested If...Then...Else selection structure

You can use Nested If either of the methods as shown above

Method 1

```
If < condition 1 > Then
```

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

Method 2

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

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

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

```
txtGrade.Text = "F"
```

```
End If
```

3.7.2 Select ..case

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

Syntax of the Select...Case selection structure

```
Select Case Index
```

```
Case 0
```

```
Statements
```

```
Case 1
```

```
Statements
```

```
End Select
```

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

```
Dim average as Integer
```

```
average = txtAverage.Text
```

```
Select Case average
```

```
Case 100 To 75
```

```
txtGrade.Text = "A"
```

```
Case 74 To 65
```

```
txtGrade.Text = "B"
```

```
Case 64 To 55
```

```
txtGrade.Text = "C"
```

```
Case 54 To 45
```

```
txtGrade.Text = "S"
```

```
Case 44 To 0
```

```
txtGrade.Text = "F"
```

```
Case Else
```

```
MsgBox "Invalid average marks"
```

```
End Select
```

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

3.7.3 Do while ...loop

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

```

Dim number As Integer
number = 1
Do While number <= 100
    number = number + 1
Loop

```

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

3.7.4 Do ... loop while

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

```

Dim number As Long
number = 0
Do
    number = number + 1
Loop While number < 201

```

The programs executes the statements between Do and Loop While structure in any case. Then it determines whether the counter is less than 501. If so, the program again executes the statements between Do and Loop While else exits the Loop.

3.7.5 Do ... loop until

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

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

```

Dim number As Long
number=0
Do Until number > 1000
    number = number + 1
    Print number
Loop

```

Numbers between 1 to 1000 will be displayed on the form as soon as you click on the command button.

3.7.6 For ..Next

The For...Next Loop is another way to make loops in Visual Basic. For...Next repetition structure handles all the details of counter-controlled repetition. The following loop counts the numbers from 1 to 100:

```

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

```

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

```

For x = 1 To 50 Step 2
Print x
Next

```

The following loop counts numbers as 1, 3, 5, 7..etc

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

```

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

```

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

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

```

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

```

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

3.7.7 Exit For and Exit Do Statement in Visual basic

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

```

Dim x As Integer
For x = 1 To 10

```



```

Print x
If x = 5 Then
Print "The program exited at x=5"
Exit For
End If
Next

```

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

```

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

```

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

```

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

```

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

3.8 Summary

- Visual Basic uses building blocks such as Variables, Data Types, Procedures, Functions and Control Structures in its programming environment.
- By default Visual Basic variables are of variant data types. The variant data type can store numeric, date/time or string data. When a variable is declared, a data type is supplied for it that determines the kind of data they can store.
- The different ways of declaring variables in Visual Basic are
 - a. Explicit Declaration
 - b. Using Option Explicit statement
 - c. Scope of Variables
- A variable is scoped to a procedure-level (local) or module-level variable depending on how it is declared. The scope of a variable, procedure or object determines which part of the code in our application is aware of the variable's existence.
- The Option Explicit statement checks in the module for usage of any undeclared variables and reports an error to the user.
- A static variable are not reinitialized each time Visual Invokes a procedure and therefore retains or preserves value even when a procedure ends.
- Functions are like sub procedures, except they return a value to the calling procedure. They are especially useful for taking one or more pieces of data, called arguments and performing some tasks with them.
- A property procedure is used to create and manipulate custom properties. It is used to create read only properties for Forms, Standard modules and Class modules.
- Control Statements are used to control the flow of program's execution. Visual Basic supports control structures such as if... Then, if...Then ...Else, Select...Case, and Loop structures such as Do While...Loop, While...Wend, For...Next etc method.

3.9 Self Assessment Questions

2. List various data type available in visual basic.
3. How a variable is declared in visual basic? Explain types of variable
4. declaration in visual basic?
5. What is variant data type?
6. What do you mean by scope and life time of variable?
7. What are sub procedure? Explain all.
8. What do you mean by control structures? Explain all.

Unit - 4 : Arrays and Built in Functions

Structure of Unit:

- 4.0 Objectives
- 4.1 Introduction
- 4.2 Fixed size and Dynamic Arrays
- 4.3 Data Type Conversion functions
- 4.4 VB Built in functions
 - 4.4.1 Date and time,
 - 4.4.2 Format and String.
- 4.5 Summary
- 4.6 Self Assessment Questions

4.0 Objectives

After completing this unit you will understand:

- Introduction to array in visual basic
- Various conversion function in visual basic
- Date and time function
- Format and string functions

4.1 Introduction

An array is a list of variables with the same data type and name. When we work with a single item, we only need to use one variable. However, if we have a list of items which are of similar type to deal with, we need to declare an array of variables instead of using a variable for each item.

For example, if we need to enter one hundred names, it is difficult to declare 100 different names, this is a waste of time and efforts. So, instead of declaring one hundred different variables, we need to declare only one array. We differentiate each item in the array by using subscript, the index value of each item, for example name(1), name(2), name(3)etc. , which will make declaring variables streamline and much systematic.

4.2 Fixed size and Dynamic Arrays

An array can be one dimensional or multidimensional. One dimensional array is like a list of items or a table that consists of one row of items or one column of items.

A two dimensional array is a table of items that make up of rows and columns. The format for a one dimensional array is ArrayName(x), the format for a two dimensional array is ArrayName(x,y) and a three dimensional array is ArrayName(x,y,z) . Normally it is sufficient to use one dimensional and two dimensional array ,you only need to use higher dimensional arrays if you need to deal more complex problems.

Declaring arrays

We could use Public or Dim statement to declare an array just as the way we declare a single variable. The Public statement declares an array that can be used throughout an application while the Dim statement declare an array that could be used only in a local procedure.

The general format to declare an array is as follows:

Dim arrayName(subs) as dataType

where subs indicates the last subscript in the array.

Dim CustName(10) as String

will declare an array that consists of 10 elements if the statement Option Base 1 appear in the declaration area, starting from CustName(1) to CustName(10). Otherwise, there will be 11 elements in the array starting from CusName(0) through to CusName(10).

Dim Count(100 to 500) as Integer declares an array that consists of the first element starting from Count(100) and ends at Count(500)

An Example

```
Dim CustName(10) As String
Dim num As Integer
Private Sub addName()
    For num = 1 To 10
        CustName(num) = InputBox("Enter the Customer name", "Enter Name", "", 1500, 4500)
        If CustName(num) <> "" Then
            Form1.Print CustName(num)
        Else
            End
        End If
    Next
End Sub

Private Sub Exit_Click() //exit command button
    End
End Sub

Private Sub Start_Click() //start command button
    Form1.Cls
    addName
End Sub
```

The above program accepts data entry through an input box and displays the entries in the form itself. As you can see, this program will only allow a user to enter 10 names each time when he click on the start button.

Array Limitations

The only data limitation of arrays is that all the elements of the array must be of the same data type. If this limitation is a problem to you, a collection might be a better solution than an array. Of course, when working

with an array of variants, each element of the array can hold a different data type value. However, the extra memory and processing requirements of using variants in an array limit this approach.

Declaring a variable-size array requires syntax similar to fixed-length arrays.

Dim ArrayName () [As DataType]

In his case, the UpperBound and LowerBound are purposely omitted. Visual Basic recognizes this declaration as a variable-size array and expects you to eventually establish the array size with the Redim statement:

Redim ArrayName ([LowerBound to [UpperBound]])

As soon as the Redim statement is applied, Visual Basic allocates as much memory as required to contain the array. You can Redim an array multiple times, dynamically altering the amount of memory occupied by the array.

The bounds of an array can be determined with the UBound (upper bound) and LBound (lower bound) functions. The syntax of these functions is as follows:

UBound (ArrayName)

LBound (ArrayName)

Because the bound values of arrays are long integers, both of these functions return long values.

You might use fixed and dynamic arrays in an application. Notice how the dynamic arrays are set to specific sizes by the Redim statement before they are used. Notice also that, because the Redim statements do not include the LowerBound specifier, each of the dynamic arrays begins at index 0.

```
Sub ArrayDemo ( )
    Dim aintInt1 (1 to 10) 'Fixed array of integers
    Dim astrStr1 (1 to 10) 'Fixed array of strings

    Dim aintInt2 ( ) 'Dynamic array of integers
    Dim astrStr2 ( ) 'Dynamic array of strings

    Dim i As Integer

    ReDim aintInt2 (5) 'Now a fixed array of integers
    ReDim astrStr2 (5) 'Now a fixed array of strings

    For i = 1 To 10
        aintInt1 (i) = i * 100
        astrStr1 (i) = Format (i * 100, "0000")
        MsgBox i & " " & aintInt1 (i) & " " & astrStr1(i)
    Next i

    For i = 0 To 5
        aintInt2 (i) = i * 100
        astrStr2(i) = Format ( i * 100, "0000")
        MsgBox i & " " & aintInt2 (i) & " " & astrStr2(i)
    Next i
End sub
```

The Redim statement discards the contents of an array if data is already present. Use the Preserve keyword when it is important to retain the contents of an array as it is redimensioned:

ReDim Preserve astrInt2 (1 to 15)

ReDim Preserve astrStr2 (1 to 15)

The Preserve statement instructs Visual Basic to create a new array with the bounds specified in the Redim statement and then copy the contents of the original array into the new array.

4.3 Data Type Conversion functions

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

Conversion To	Function
Boolean	Cbool
Byte	Cbyte
Currency	Ccur
Date	Cdate
Decimals	Cdec
Double	CDbl
Integer	Cint
Long	CLng
Single	CSng
String	CStr
Variant	Cvar
Error	CVErr

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

4.4 VB Built in functions

A function is similar to a normal procedure but the main purpose of the function is to accept a certain input from the user and return a value which is passed on to the main program to finish the execution. There are two types of functions, the built-in functions (or internal functions) and the functions created by the programmers.

The general format of a function is **FunctionName (arguments)** The arguments are values that are passed on to the function. In this lesson, we are going to learn two very basic but useful internal functions of Visual basic , i.e. the **MsgBox()** and **InputBox ()** functions.

MsgBox () Function

The objective of MsgBox is to produce a pop-up message box and prompt the user to click on a command button before he /she can continues. This format is as follows:

yourMsg=MsgBox(Prompt, Style Value, Title)

The first argument, Prompt, will display the message in the message box. The Style Value will determine what type of command buttons appear on the message box. The Title argument will display the title of the message board.

Table : Style Values

Style Value	Named Constant	Buttons Displayed
0	vbOkOnly	Ok button
1	vbOkCancel	Ok and Cancel buttons
2	vbAbortRetryIgnore	Abort, Retry and Ignore buttons.
3	vbYesNoCancel	Yes, No and Cancel buttons
4	vbYesNo	Yes and No buttons
5	vbRetryCancel	Retry and Cancel buttons

We can use named constant in place of integers for the second argument to make the programs more readable. In fact, VB6 will automatically shows up a list of names constant where you can select one of them.

Example: yourMsg=MsgBox("Click OK to Proceed", 1, "Startup Menu")

and yourMsg=Msg("Click OK to Proceed". vbOkCancel,"Startup Menu")

are the same.

yourMsg is a variable that holds values that are returned by the MsgBox () function. The values are determined by the type of buttons being clicked by the users. It has to be declared as Integer data type in the procedure or in the general declaration section. Table shows the values, the corresponding named constant and buttons.

Table : Return Values and Command Buttons

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

The InputBox() Function

An InputBox() function will display a message box where the user can enter a value or a message in the form of text. The format is

`myMessage=InputBox(Prompt, Title, default_text, x-position, y-position)`

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

- Prompt - The message displayed normally as a question asked.
- Title - The title of the Input Box.
- default-text - The default text that appears in the input field where users can use it as his intended input or he may change to the message he wish to key in.
- x-position and y-position - the position or the coordinate of the input box.



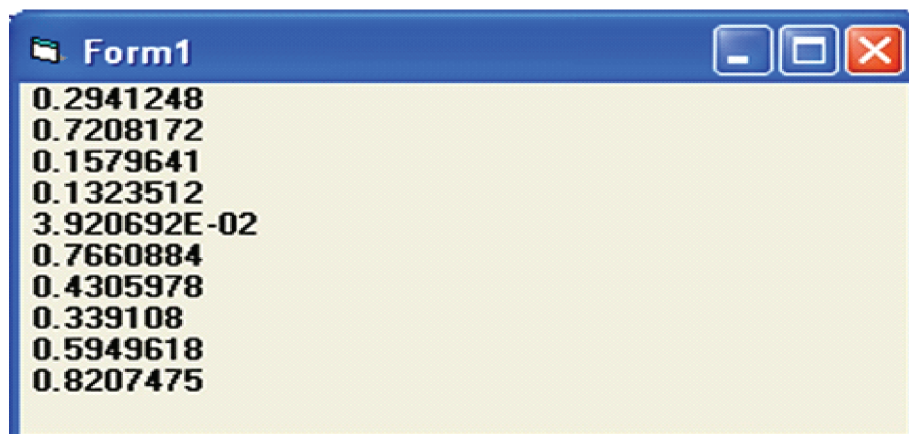
Mathematical Function

The mathematical functions are very useful and important in programming because very often we need to deal with mathematical concepts in programming such as chance and probability, variables, mathematical logics, calculations, coordinates, time intervals and etc. The common mathematical functions in Visual Basic are **Rnd, Sqr, Int, Abs, Exp, Log, Sin, Cos, Tan, Atn, Fix and Round.**

i) Rnd is very useful when we deal with the concept of chance and probability. The Rnd function returns a random value between 0 and 1. In Example 1. When you run the program, you will get an output of 10 random numbers between 0 and 1. Randomize Timer is a vital statement here as it will randomize the process.

Example 1: Private Sub Form_ActivateRandomize TimerFor x=1 to 10Print RndNext xEnd Sub

The Output for example 1 is shown below:



Random numbers in its original form are not very useful in programming until we convert them to integers. For example, if we need to obtain a random output of 6 random integers ranging from 1 to 6, which make the program behave as a virtual die, we need to convert the random numbers using the format `Int(Rnd*6)+1`. Let's study the following example:

In this example, `Int(Rnd*6)` will generate a random integer between 0 and 5 because the function `Int` truncates the decimal part of the random number and returns an integer. After adding 1, you will get a random number between 1 and 6 every time you click the command button. For example, let say the random number generated is 0.98, after multiplying it by 6, it becomes 5.88, and using the integer function `Int(5.88)` will convert the number to 5; and after adding 1 you will get 6.

In this example, you place a command button and change its caption to 'roll die'. You also need to insert a label into the form and clear its caption at the designing phase and make its font bigger and bold. Then set the border value to 1 so that it displays a border; and after that set the alignment to center. The statement `Label1.Caption=Num` means the integer generated will be displayed as the caption of the label.

Example 2:

```
Dim num as integer

Private Sub Command1_Click ()

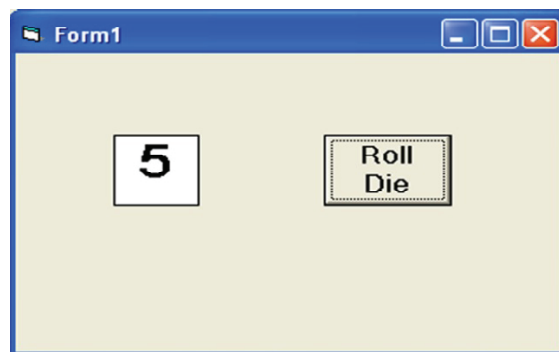
    Randomize Timer

    Num=Int(Rnd*6)+1

    Label1.Caption=Num

End Sub
```

Now, run the program and then click on the roll die button, you will get an output like the figure below:



The Numeric Functions

The numeric functions are `Int`, `Sqr`, `Abs`, `Exp`, `Fix`, `Round` and `Log`.

a) `Int` is the function that converts a number into an integer by truncating its decimal part and the resulting integer is the largest integer that is smaller than the number. For example, `Int(2.4)=2`, `Int(4.8)=4`, `Int(-4.6)=-5`, `Int(0.032)=0` and so on.

b) `Sqr` is the function that computes the square root of a number. For example, `Sqr(4)=2`, `Sqr(9)=3` and etc.

c) `Abs` is the function that returns the absolute value of a number. So `Abs(-8) = 8` and `Abs(8)= 8`.

d) Exp of a number x is the value of e^x . For example, $\text{Exp}(1) = e^1 = 2.7182818284590$

e) Fix and **Int** are the same if the number is a positive number as both truncate the decimal part of the number and return an integer. However, when the number is negative, it will return the smallest integer that is larger than the number. For example, $\text{Fix}(-6.34) = -6$ while $\text{Int}(-6.34) = -7$.

f) Round is the function that rounds up a number to a certain number of decimal places. The Format is $\text{Round}(n, m)$ which means to round a number n to m decimal places. For example, $\text{Round}(7.2567, 2) = 7.26$

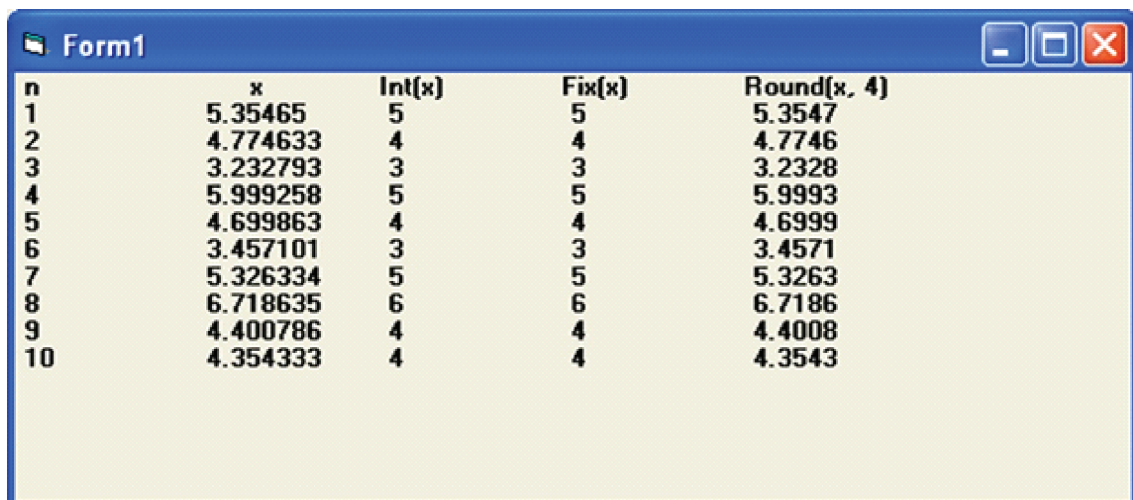
g) Log is the function that returns the natural Logarithm of a number. For example,

$\text{Log } 10 = 2.302585$

Example 3

This example computes the values of $\text{Int}(x)$, $\text{Fix}(x)$ and $\text{Round}(x, n)$ in a table form. It uses the Do Loop statement and the Rnd function to generate 10 numbers. The statement $x = \text{Round}(\text{Rnd} * 7, 7)$ rounds a random number between 0 and 7 to 7 decimal places. Using commas in between items will create spaces between them and hence a table of values can be created. The program and output are shown below.

```
Private Sub Form_Activate ()  
    n = 1  
    Print "n", "x", "Int(x)", "Fix(x)", "Round(x, 4)"  
    Do While n < 11  
        Randomize Timer  
        x = Round (Rnd * 7, 7)  
        Print n, x, Int(x), Fix(x), Round(x, 4)  
        n = n + 1  
    Loop  
End Sub
```



n	x	Int(x)	Fix(x)	Round(x, 4)
1	5.35465	5	5	5.3547
2	4.774633	4	4	4.7746
3	3.232793	3	3	3.2328
4	5.999258	5	5	5.9993
5	4.699863	4	4	4.6999
6	3.457101	3	3	3.4571
7	5.326334	5	5	5.3263
8	6.718635	6	6	6.7186
9	4.400786	4	4	4.4008
10	4.354333	4	4	4.3543

4.4.1 Date and time,

VB function for current date and/or time:

Now Returns the current date and time together

Date Returns the current date

Time Returns the current time

For the examples that follow, assume that the current Date/Time (Now) is Friday, August 31, 2001 at 9:15:20 PM.

Function	Description
DateValue	<p>Returns the date portion of a Date/Time value, with the time portion "zeroed out". (Note: When the time portion of a date/time variable is "zeroed out", the time would be interpreted as 12:00 AM.)</p> <p>Example:</p> <pre>Dim dtmTest As Date</pre> <pre>dtmTest = DateValue(Now)</pre> <p>At this point, the date portion of dtmTest is 8/31/2001, with a time portion of 0 (12:00 AM midnight).</p>
TimeValue	<p>Returns the time portion of a Date/Time value, with the date portion "zeroed out". (Note: When a date/time variable is "zeroed out", the date will actually be interpreted as December 30, 1899.)</p> <p>Example:</p> <pre>Dim dtmTest As Date</pre> <pre>dtmTest = TimeValue(Now)</pre> <p>At this point, the time portion of dtmTest is 9:15:20 PM, with a date portion of 0 (12/30/1899).</p>
Weekday	<p>Returns a number from 1 to 7 indicating the day of the week for a given date, where 1 is Sunday and 7 is Saturday.</p> <p>Example:</p> <pre>intDOW = Weekday(Now) ' intDOW = 6</pre> <p>Note:</p> <p>When necessary to refer to a day of the week in code, VB has a set of built-in constants that can be used instead of the hard-coded values 1 thru 7:</p> <p>Constant Value</p> <pre>vbSunday 1</pre> <pre>vbMonday 2</pre> <pre>vbTuesday 3</pre>

Function	Description
Weekday Name	<p>Returns a string containing the weekday name ("Sunday" thru "Saturday"), given a numeric argument with the value 1 through 7.</p> <p>Example:</p> <pre>strDOW = WeekdayName(6) ' strDOW = "Friday"</pre> <p>The WeekdayName function takes an optional, second argument (Boolean) indicating whether or not to abbreviate the weekday name. By default, the second argument is False, meaning do not abbreviate and return the full name. If True, the first three letters of the weekday name will be returned:</p> <p>Example:</p> <pre>strDOW = WeekdayName(6, True) ' strDOW = "Fri"</pre> <p>You can nest the Weekday function within the WeekdayName function to get the weekday name for a given date:</p> <p>Example:</p> <pre>strDOW = WeekdayName(Weekday(Now)) ' strDOW = "Friday"</pre>
Month	<p>Returns a number from 1 to 12 indicating the month portion of a given date.</p> <p>Example:</p> <pre>intMonth = Month(Now) ' intMonth = 8</pre>
MonthName	<p>Returns a string containing the month name ("January" thru "December"), given a numeric argument with the value 1 through 12.</p> <p>Example:</p> <pre>strMoName = MonthName(8) ' strMoName = "August"</pre> <p>The MonthName function takes an optional, second argument (Boolean) indicating whether or not to abbreviate the month name. By default, the second argument is False, meaning do not abbreviate and return the full name. If True, the first three letters of the month name will be returned:</p>

The following functions are used to isolate a particular part of a time:

Function	Description
Hour	Returns an integer specifying a whole number between 0 and 23 representing the hour of the day. Example: intHour = Hour(Now) ' intHour = 21 (for 9 PM)
Minute	Returns an integer specifying a whole number between 0 and 59 representing the minute of the hour. Example: intMinute = Minute(Now) ' intMinute = 15
Second	Returns an integer specifying a whole number between 0 and 59 representing the second of the minute. Example: intSecond = Second(Now) ' intSecond = 20

To demonstrate the date functions shown thus far, set up a "Try It" project, and place the following code in the cmdTryIt_Click event:

```
Private Sub cmdTryIt_Click()

Print "Now:"; Tab(30); Now

Print "Using DateValue:"; Tab(30); DateValue(Now)

Print "Using TimeValue:"; Tab(30); TimeValue(Now)

Print "Using Weekday:"; Tab(30); Weekday(Now)

Print "Using WeekdayName:"; Tab(30); WeekdayName(Weekday(Now))

Print "Using WeekdayName (abbrev.):"; Tab(30); WeekdayName(Weekday(Now), True)

Print "Using Month:"; Tab(30); Month(Now)

Print "Using MonthName:"; Tab(30); MonthName(Month(Now))

Print "Using MonthName (abbrev.):"; Tab(30); MonthName(Month(Now), True)

Print "Using Day:"; Tab(30); Day(Now)

Print "Using Year:"; Tab(30); Year(Now)

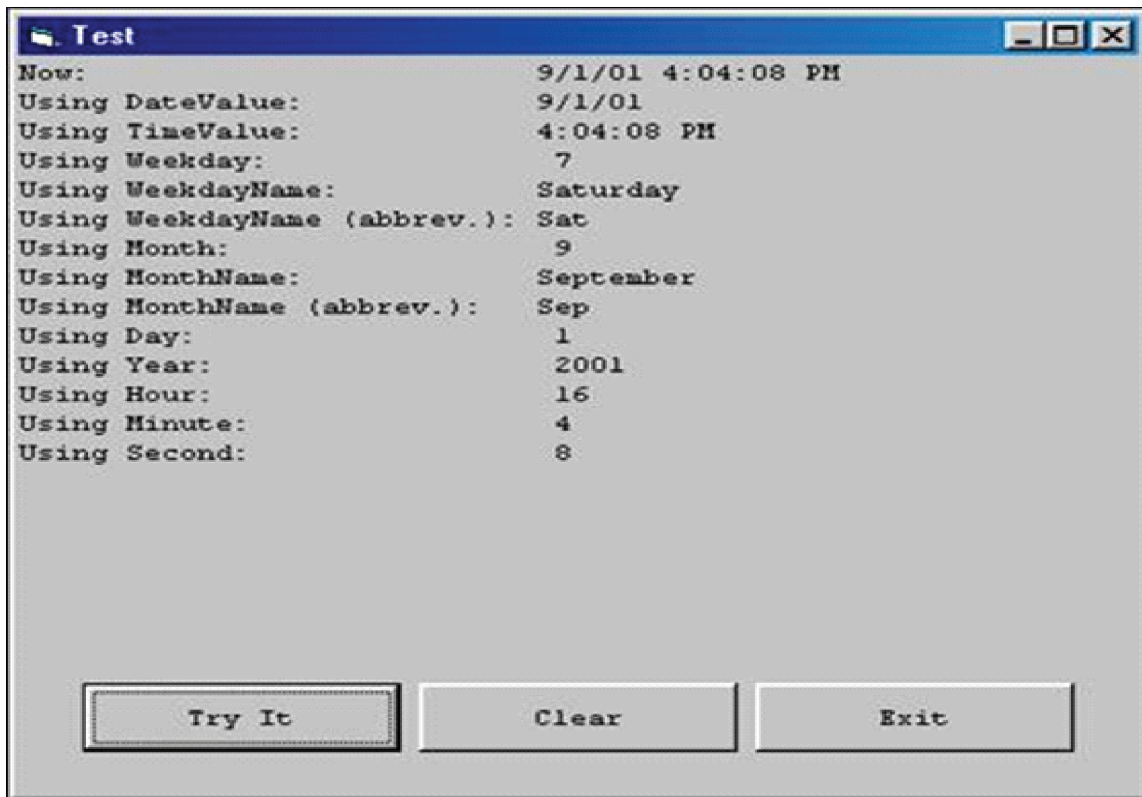
Print "Using Hour:"; Tab(30); Hour(Now)

Print "Using Minute:"; Tab(30); Minute(Now)

Print "Using Second:"; Tab(30); Second(Now)

End Sub
```

Run the project and click the "Try It" button. The output should look similar to the following:



The DatePart Function

The generic **DatePart** function returns an Integer containing the specified part of a given date/time value. Thus, it incorporates the functionality of the Weekday, Month, Day, Year, Hour, Minute, and Second functions. In addition, it can be used to get the quarter of a given date (1 through 4), the "Julian" date (the day of the year from 1 to 366), and the week number (1 through 53).

Syntax:

DatePart(interval, date[,firstdayofweek[,firstweekofyear]])

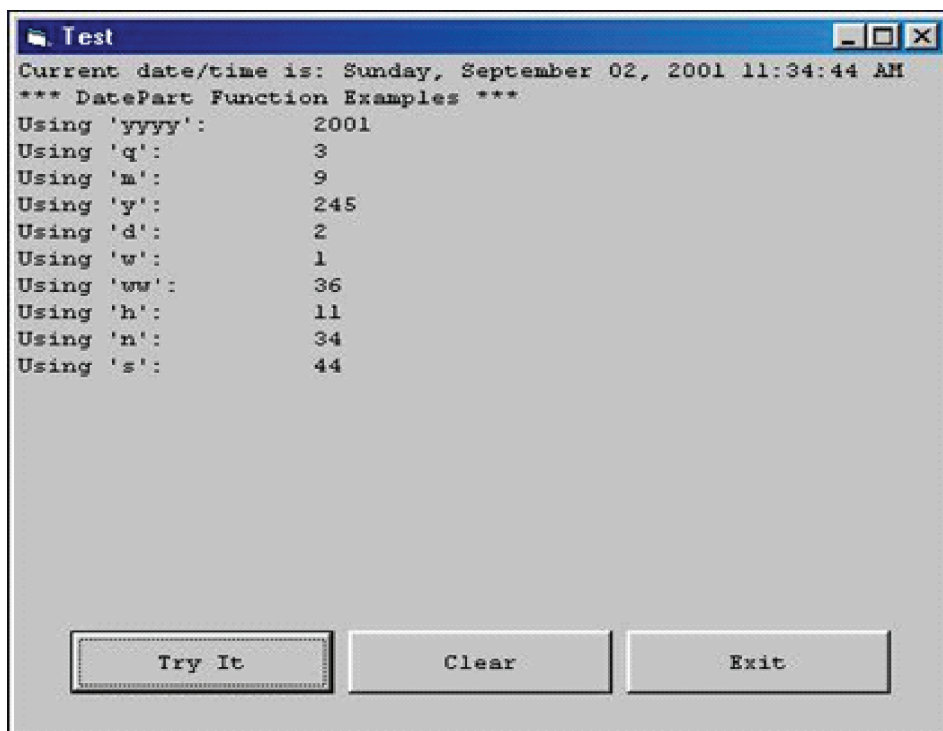
The DatePart function syntax has these parts:

Part	Description																																	
<i>interval</i>	<p>Required. String expression that is the interval of time you want to return.</p> <p>The string expression can be any of the following:</p> <table><tr><th>Expression</th><th>Description</th><th>Possible Range of Values</th></tr><tr><td>"yyyy"</td><td>Year</td><td>100 to 9999</td></tr><tr><td>"q"</td><td>Quarter</td><td>1 to 4</td></tr><tr><td>"m"</td><td>Month</td><td>1 to 12</td></tr><tr><td>"y"</td><td>Day of year</td><td>1 to 366 (a "Julian" date)</td></tr><tr><td>"d"</td><td>Day</td><td>1 to 31</td></tr><tr><td>"w"</td><td>Weekday</td><td>1 to 7</td></tr><tr><td>"ww"</td><td>Week</td><td>1 to 53</td></tr><tr><td>"h"</td><td>Hour</td><td>0 to 23</td></tr><tr><td>"n"</td><td>Minute</td><td>0 to 59</td></tr><tr><td>"s"</td><td>Second</td><td>0 to 59</td></tr></table>	Expression	Description	Possible Range of Values	"yyyy"	Year	100 to 9999	"q"	Quarter	1 to 4	"m"	Month	1 to 12	"y"	Day of year	1 to 366 (a "Julian" date)	"d"	Day	1 to 31	"w"	Weekday	1 to 7	"ww"	Week	1 to 53	"h"	Hour	0 to 23	"n"	Minute	0 to 59	"s"	Second	0 to 59
Expression	Description	Possible Range of Values																																
"yyyy"	Year	100 to 9999																																
"q"	Quarter	1 to 4																																
"m"	Month	1 to 12																																
"y"	Day of year	1 to 366 (a "Julian" date)																																
"d"	Day	1 to 31																																
"w"	Weekday	1 to 7																																
"ww"	Week	1 to 53																																
"h"	Hour	0 to 23																																
"n"	Minute	0 to 59																																
"s"	Second	0 to 59																																
<i>date</i>	Required. Date value that you want to evaluate.																																	
<i>firstdayofweek</i>	Optional. A constant that specifies the first day of the week. If not specified, Sunday is assumed.																																	
<i>firstweekofyear</i>	Optional. A constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs.																																	

To demonstrate DatePart, set up a “Try It” project, and place the following code in the cmdTryIt_Click event:

```
Private Sub cmdTryIt_Click()  
    Print “Current date/time is: “; _  
    Format$(Now, “Long Date”); _  
    Spc(1); _  
    Format$(Now, “Long Time”)  
    Print “*** DatePart Function Examples ***”  
    Print “Using ‘yyyy’:”; Tab(20); DatePart(“yyyy”, Now)  
    Print “Using ‘q’:”; Tab(20); DatePart(“q”, Now)  
    Print “Using ‘m’:”; Tab(20); DatePart(“m”, Now)  
    Print “Using ‘y’:”; Tab(20); DatePart(“y”, Now)  
    Print “Using ‘d’:”; Tab(20); DatePart(“d”, Now)  
    Print “Using ‘w’:”; Tab(20); DatePart(“w”, Now)  
    Print “Using ‘ww’:”; Tab(20); DatePart(“ww”, Now)  
    Print “Using ‘h’:”; Tab(20); DatePart(“h”, Now)  
    Print “Using ‘n’:”; Tab(20); DatePart(“n”, Now)  
    Print “Using ‘s’:”; Tab(20); DatePart(“s”, Now)  
End Sub
```

Run the project and click the “Try It” button. The output should look similar to the following:



Piecing Separate Numbers Together to Form a Date or Time Value

In the previous examples, we saw ways to isolate parts of a date/time value. What if you need to go the “other way”? If you have the separate parts of a date/time value in different variables and want to piece them together to formulate a date or time, there are two functions you can use to do this: **DateSerial** and **TimeSerial**.

The **DateSerial** takes three numeric arguments: year, month, and day respectively. It returns a date based on those values.

Example:

```
Dim intYear As Integer
Dim intMonth As Integer
Dim intDay As Integer
Dim dtmNewDate As Date
intYear = 2001
intMonth = 9
intDay = 2
dtmNewDate = DateSerial(intYear, intMonth, intDay)
‘ returns 9/2/2001
```

The **TimeSerial** takes three numeric arguments: hour, minute, and second respectively. It returns a time based on those values.

Example:

```
Dim intHour As Integer
Dim intMinute As Integer
Dim intSecond As Integer
Dim dtmNewTime As Date
intHour = 11
intMinute = 34
intSecond = 44
dtmNewTime = TimeSerial(intHour, intMinute, intSecond)
‘ returns 11:34:44 (AM)
```

4.4.2 Format function and String.

Formatting output is a very important part of programming so that the data can be presented systematically and clearly to the users. Data in the previous lesson were presented fairly systematically through the use of commas and some of the functions like Int, Fix and Round. However, to have better control of the output format, we can use a number of formatting functions in Visual basic.

The three most common formatting functions in VB are **Tab**, **Space**, and **Format**

(i) The Tab function

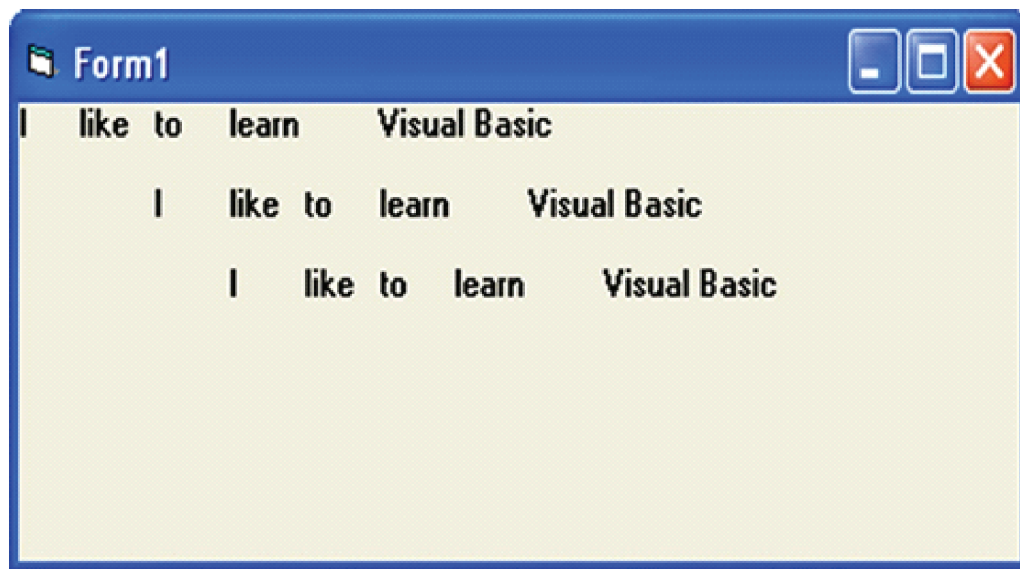
Tab (n); x

The item x will be displayed at a position that is n spaces from the left border of the output form. There must be a semicolon in between Tab and the items you intend to display (VB will actually do it for you automatically).

Example1

```
.Private Sub Form_Activate  
    Print "I"; Tab(5); "like"; Tab(10); "to"; Tab(15); "learn"; Tab(20); "VB"  
    Print  
    Print Tab(10); "I"; Tab(15); "like"; Tab(20); "to"; Tab(25); "learn"; Tab(20); "VB"  
    Print  
    Print Tab(15); "I"; Tab(20); "like"; Tab(25); "to"; Tab(30); "learn"; Tab(35); "VB"  
End sub
```

The Output for example 1 is shown below:



ii) The Space function

The Space function is very closely linked to the Tab function. However, there is a minor difference. While Tab (n) means the item is placed n spaces from the left border of the screen, the Space function specifies the number of spaces between two consecutive items. For example, the procedure

Example 2

```
Private Sub Form_Activate()  
    Print "Visual"; Space(10); "Basic"  
End Sub
```

Means that the words Visual and Basic will be separated by 10 spaces

(iii) The Format function

The Format function is a very powerful formatting function which can display the numeric values in various forms. There are two types of Format function, one of them is the built-in or predefined format while another one can be defined by the users.

(i) The format of the predefined Format function is

Format (n, “style argument”)

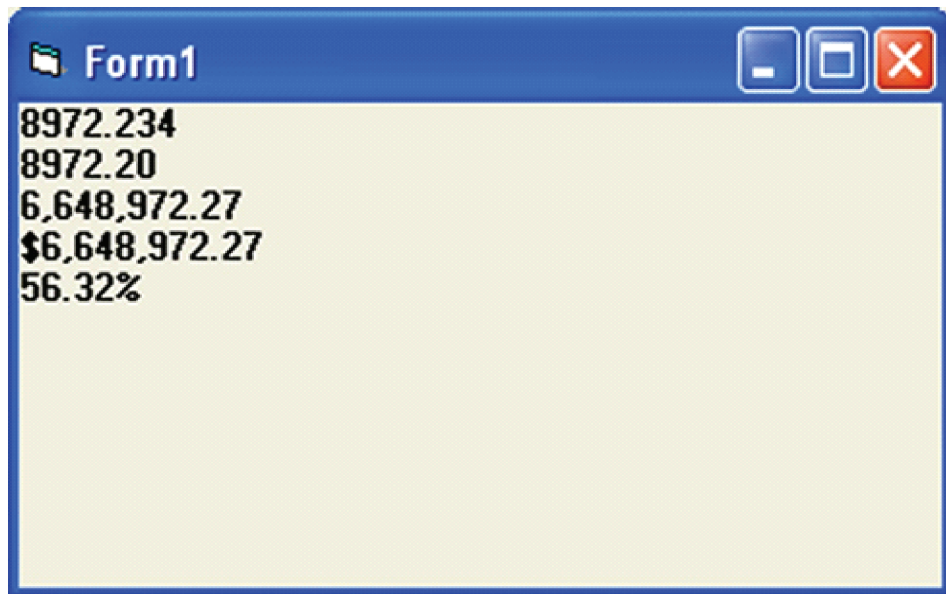
where n is a number and the list of style arguments is given in the table

Style argument	Explanation	Example
General Number	To display the number without having separators between thousands.	Format(8972.234, “General Number”)=8972.234
Fixed	To display the number without having separators between thousands and rounds it up to two decimal places.	Format(8972.2, “Fixed”)=8972.23
Standard	To display the number with separators or separators between thousands and rounds it up to two decimal places.	Format(6648972.265, “Standard”)=6,648,972.27
Currency	To display the number with the dollar sign in front, has separators between thousands as well as rounding it up to two decimal places.	Format(6648972.265, “Currency”)= \$6,648,972.27
Percent	Converts the number to the percentage form and displays a % sign and rounds it up to two decimal places.	Format(0.56324, “Percent”)=56.32 %

Example 3

```
Private Sub Form_Activate()  
Print Format (8972.234, “General Number”)  
Print Format (8972.2, “Fixed”)  
Print Format (6648972.265, “Standard”)  
Print Format (6648972.265, “Currency”)  
Print Format (0.56324, “Percent”)  
End Sub
```

Now, run the program and you will get an output like the figure below:



VB has numerous built-in functions for processing strings. Most VB string-handling functions return a string, although some return a number (such as the Len function, which returns the length of a string and functions like Instr and InstrRev, which return a character position within the string). The functions that return strings can be coded with or without the dollar sign (\$) at the end, although it is more efficient to use the version with the dollar sign.

Function:	Len
Description:	Returns a Long containing the length of the specified string
Syntax:	Len(<i>string</i>) Where <i>string</i> is the string whose length (number of characters) is to be returned.
Example:	lngLen = Len("Visual Basic") ' lngLen = 12

Function:	Mid\$ (or Mid)								
Description:	Returns a substring containing a specified number of characters from a string.								
Syntax:	Mid\$(<i>string</i>, <i>start</i>[, <i>length</i>]) The Mid\$ function syntax has these parts: <table border="1"> <thead> <tr> <th>Part</th><th>Description</th></tr> </thead> <tbody> <tr> <td><i>string</i></td><td>Required. String expression from which characters are returned.</td></tr> <tr> <td><i>start</i></td><td>Required; Long. Character position in string at which the part to be taken begins. If start is greater than the number of characters in string, Mid returns a zero-length string ("").</td></tr> <tr> <td><i>length</i></td><td>Optional; Long. Number of characters to return. If omitted or if there are fewer than length characters in the text (including the character at start), all characters from the start position to the end of the string are returned.</td></tr> </tbody> </table>	Part	Description	<i>string</i>	Required. String expression from which characters are returned.	<i>start</i>	Required; Long. Character position in string at which the part to be taken begins. If start is greater than the number of characters in string, Mid returns a zero-length string ("").	<i>length</i>	Optional; Long. Number of characters to return. If omitted or if there are fewer than length characters in the text (including the character at start), all characters from the start position to the end of the string are returned.
Part	Description								
<i>string</i>	Required. String expression from which characters are returned.								
<i>start</i>	Required; Long. Character position in string at which the part to be taken begins. If start is greater than the number of characters in string, Mid returns a zero-length string ("").								
<i>length</i>	Optional; Long. Number of characters to return. If omitted or if there are fewer than length characters in the text (including the character at start), all characters from the start position to the end of the string are returned.								
Example:	strSubstr = Mid\$("Visual Basic", 3, 4) ' strSubstr = "sual" Note: Mid\$ can also be used on the left side of an assignment statement, where you can replace a substring within a string. <u>Example:</u> strTest = "Visual Basic" Mid\$(strTest, 3, 4) = "xxxx" 'strTest now contains "Vixxxx Basic" In VB6, the Replace\$ function was introduced, which can also be used to replace characters within a string.								

Function:	Left\$ (or Left)						
Description:	Returns a substring containing a specified number of characters from the beginning (left side) of a string.						
Syntax:	<p>Left\$(string, length)</p> <p>The Left\$ function syntax has these parts:</p> <table> <tr> <th><u>Part</u></th><th><u>Description</u></th></tr> <tr> <td><i>string</i></td><td>Required. String expression from which the leftmost characters are returned.</td></tr> <tr> <td><i>length</i></td><td>Required; Long. Numeric expression indicating how many characters to return. If 0, a zero-length string ("") is returned. If greater than or equal to the number of characters in string, the entire string is returned.</td></tr> </table>	<u>Part</u>	<u>Description</u>	<i>string</i>	Required. String expression from which the leftmost characters are returned.	<i>length</i>	Required; Long. Numeric expression indicating how many characters to return. If 0, a zero-length string ("") is returned. If greater than or equal to the number of characters in string, the entire string is returned.
<u>Part</u>	<u>Description</u>						
<i>string</i>	Required. String expression from which the leftmost characters are returned.						
<i>length</i>	Required; Long. Numeric expression indicating how many characters to return. If 0, a zero-length string ("") is returned. If greater than or equal to the number of characters in string, the entire string is returned.						
Example:	<pre>strSubstr = Left\$("Visual Basic", 3) ' strSubstr = "Vis"</pre> <p>' Note that the same thing could be accomplished with Mid\$:</p> <pre>strSubstr = Mid\$("Visual Basic", 1, 3)</pre>						

Function:	Right\$ (or Right)						
Description:	Returns a substring containing a specified number of characters from the end (right side) of a string.						
Syntax:	<p>Right\$(string, length)</p> <p>The Right\$ function syntax has these parts:</p> <table> <tr> <th><u>Part</u></th><th><u>Description</u></th></tr> <tr> <td><i>string</i></td><td>Required. String expression from which the rightmost characters are returned.</td></tr> <tr> <td><i>length</i></td><td>Required; Long. Numeric expression indicating how many characters to return. If 0, a zero-length string ("") is returned. If greater than or equal to the number of characters in string, the entire string is returned.</td></tr> </table>	<u>Part</u>	<u>Description</u>	<i>string</i>	Required. String expression from which the rightmost characters are returned.	<i>length</i>	Required; Long. Numeric expression indicating how many characters to return. If 0, a zero-length string ("") is returned. If greater than or equal to the number of characters in string, the entire string is returned.
<u>Part</u>	<u>Description</u>						
<i>string</i>	Required. String expression from which the rightmost characters are returned.						
<i>length</i>	Required; Long. Numeric expression indicating how many characters to return. If 0, a zero-length string ("") is returned. If greater than or equal to the number of characters in string, the entire string is returned.						
Example:	<pre>strSubstr = Right\$("Visual Basic", 3) ' strSubstr = "sic"</pre> <p>' Note that the same thing could be accomplished with Mid\$:</p> <pre>strSubstr = Mid\$("Visual Basic", 10, 3)</pre>						

Function:	UCase\$ (or UCase)
Description:	Converts all lowercase letters in a string to uppercase. Any existing uppercase letters and non-alpha characters remain unchanged.
Syntax:	UCase\$(string)
Example:	<pre>strNew = UCase\$("Visual Basic") ' strNew = "VISUAL BASIC"</pre>

Function:	LCase\$ (or LCase)
Description:	Converts all uppercase letters in a string to lowercase. Any existing lowercase letters and non-alpha characters remain unchanged.
Syntax:	LCase\$(string)
Example:	strNew = LCase\$("Visual Basic") ' strNew = "visual basic"

Function:	Instr										
Description:	Returns a Long specifying the position of one string within another. The search starts either at the first character position or at the position specified by the <i>start</i> argument, and proceeds forward toward the end of the string (stopping when either <i>string2</i> is found or when the end of the <i>string1</i> is reached).										
Syntax:	<p>InStr(<i>start</i>, string1, string2 [, compare])</p> <p>The InStr function syntax has these parts:</p> <table> <thead> <tr> <th><u>Part</u></th><th><u>Description</u></th></tr> </thead> <tbody> <tr> <td><i>start</i></td><td>Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the first character position. The start argument is required if compare is specified.</td></tr> <tr> <td><i>string1</i></td><td>Required. String expression being searched.</td></tr> <tr> <td><i>string2</i></td><td>Required. String expression sought.</td></tr> <tr> <td><i>compare</i></td><td>Optional; numeric. A value of 0 (the default) specifies a binary (case-sensitive) search. A value of 1 specifies a textual (case-insensitive) search.</td></tr> </tbody> </table>	<u>Part</u>	<u>Description</u>	<i>start</i>	Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the first character position. The start argument is required if compare is specified.	<i>string1</i>	Required. String expression being searched.	<i>string2</i>	Required. String expression sought.	<i>compare</i>	Optional; numeric. A value of 0 (the default) specifies a binary (case-sensitive) search. A value of 1 specifies a textual (case-insensitive) search.
<u>Part</u>	<u>Description</u>										
<i>start</i>	Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the first character position. The start argument is required if compare is specified.										
<i>string1</i>	Required. String expression being searched.										
<i>string2</i>	Required. String expression sought.										
<i>compare</i>	Optional; numeric. A value of 0 (the default) specifies a binary (case-sensitive) search. A value of 1 specifies a textual (case-insensitive) search.										
Examples:	<p>lngPos = Instr("Visual Basic", "a")</p> <p>' lngPos = 5</p>										

Function:	InstrRev										
Description:	Returns a Long specifying the position of one string within another. The search starts either at the last character position or at the position specified by the <i>start</i> argument, and proceeds backward toward the beginning of the string (stopping when either <i>string2</i> is found or when the beginning of the <i>string1</i> is reached).										
Syntax:	<p>InStrRev(<i>string1</i>, <i>string2</i>[, <i>start</i>, [, <i>compare</i>]])</p> <p>The InStr function syntax has these parts:</p> <table> <thead> <tr> <th><u>Part</u></th><th><u>Description</u></th></tr> </thead> <tbody> <tr> <td><i>string1</i></td><td>Required. String expression being searched.</td></tr> <tr> <td><i>string2</i></td><td>Required. String expression sought.</td></tr> <tr> <td><i>start</i></td><td>Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the last character position.</td></tr> <tr> <td><i>compare</i></td><td>Optional; numeric. A value of 0 (the default) specifies a binary (case-sensitive) search. A value of 1 specifies a textual (case-insensitive) search.</td></tr> </tbody> </table>	<u>Part</u>	<u>Description</u>	<i>string1</i>	Required. String expression being searched.	<i>string2</i>	Required. String expression sought.	<i>start</i>	Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the last character position.	<i>compare</i>	Optional; numeric. A value of 0 (the default) specifies a binary (case-sensitive) search. A value of 1 specifies a textual (case-insensitive) search.
<u>Part</u>	<u>Description</u>										
<i>string1</i>	Required. String expression being searched.										
<i>string2</i>	Required. String expression sought.										
<i>start</i>	Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the last character position.										
<i>compare</i>	Optional; numeric. A value of 0 (the default) specifies a binary (case-sensitive) search. A value of 1 specifies a textual (case-insensitive) search.										
Examples:	<pre> lngPos = InstrRev("Visual Basic", "a") ' lngPos = 9 lngPos = InstrRev("Visual Basic", "a", 6) ' lngPos = 5 (starting at position 6) lngPos = InstrRev("Visual Basic", "A") ' lngPos = 0 (case-sensitive search) lngPos = InstrRev("Visual Basic", "A", , 1) </pre>										

Notes on Instr and InstrRev:

- Something to watch out for is that while Instr and InstrRev both accomplish the same thing (except that InstrRev processes a string from last character to first, while Instr processes a string from first character to last), the arguments to these functions are specified in a different order. The Instr arguments are (start, string1, string2, compare) whereas the InstrRev arguments are (string1, string2, start, compare).

- The Instr function has been around since the earlier days of BASIC, whereas InstrRev was not introduced until VB 6.
- Built-in “vb” constants can be used for the compare argument:
vbBinaryCompare for 0 (case-sensitive search)
vbTextCompare for 1 (case-insensitive search)

Function:	String\$ (or String)						
Description:	Returns a string containing a repeating character string of the length specified.						
Syntax:	String\$(number, character) The String\$ function syntax has these parts: <table> <tr> <th><u>Part</u></th><th><u>Description</u></th></tr> <tr> <td><i>number</i></td><td>Required; Long. Length of the returned string.</td></tr> <tr> <td><i>character</i></td><td>Required; Variant. This argument can either be a number from 0 to 255 (representing the ASCII character code* of the character to be repeated) or a string expression whose first character is used to build the return string.</td></tr> </table>	<u>Part</u>	<u>Description</u>	<i>number</i>	Required; Long. Length of the returned string.	<i>character</i>	Required; Variant. This argument can either be a number from 0 to 255 (representing the ASCII character code* of the character to be repeated) or a string expression whose first character is used to build the return string.
<u>Part</u>	<u>Description</u>						
<i>number</i>	Required; Long. Length of the returned string.						
<i>character</i>	Required; Variant. This argument can either be a number from 0 to 255 (representing the ASCII character code* of the character to be repeated) or a string expression whose first character is used to build the return string.						
Examples:	strTest = String\$(5, "a") ' strTest = "aaaaa" strTest = String\$(5, 97) ' strTest = "aaaaa" (97 is the ASCII code for "a")						

* A list of the ASCII character codes is presented at the end of this topic.

Function:	Space\$ (or Space)
Description:	Returns a string containing the specified number of blank spaces.
Syntax:	Space\$(number) Where <i>number</i> is the number of blank spaces desired.
Examples:	strTest = Space\$(5) ' strTest = " "

Function:	Replace\$ (or Replace)														
Description:	Returns a string in which a specified substring has been replaced with another substring a specified number of times.														
Syntax:	Replace\$(expression, find, replacewith[, start[, count[, compare]]]) The Replace\$ function syntax has these parts: <table> <tr> <th><u>Part</u></th><th><u>Description</u></th></tr> <tr> <td><i>expression</i></td><td>Required. String expression containing substring to replace.</td></tr> <tr> <td><i>find</i></td><td>Required. Substring being searched for.</td></tr> <tr> <td><i>replacewith</i></td><td>Required. Replacement substring.</td></tr> <tr> <td><i>start</i></td><td>Optional. Position within <i>expression</i> where substring search is to begin. If omitted, 1 is assumed.</td></tr> <tr> <td><i>count</i></td><td>Optional. Number of substring substitutions to perform. If omitted, the default value is -1, which means make all possible substitutions.</td></tr> <tr> <td><i>compare</i></td><td>Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. (0 = case sensitive, 1 = case-insensitive) Built-in "vb" constants can be used for the <i>compare</i> argument: vbBinaryCompare for 0 (case-sensitive search) vbTextCompare for 1 (case-insensitive search)</td></tr> </table>	<u>Part</u>	<u>Description</u>	<i>expression</i>	Required. String expression containing substring to replace.	<i>find</i>	Required. Substring being searched for.	<i>replacewith</i>	Required. Replacement substring.	<i>start</i>	Optional. Position within <i>expression</i> where substring search is to begin. If omitted, 1 is assumed.	<i>count</i>	Optional. Number of substring substitutions to perform. If omitted, the default value is -1, which means make all possible substitutions.	<i>compare</i>	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. (0 = case sensitive, 1 = case-insensitive) Built-in "vb" constants can be used for the <i>compare</i> argument: vbBinaryCompare for 0 (case-sensitive search) vbTextCompare for 1 (case-insensitive search)
<u>Part</u>	<u>Description</u>														
<i>expression</i>	Required. String expression containing substring to replace.														
<i>find</i>	Required. Substring being searched for.														
<i>replacewith</i>	Required. Replacement substring.														
<i>start</i>	Optional. Position within <i>expression</i> where substring search is to begin. If omitted, 1 is assumed.														
<i>count</i>	Optional. Number of substring substitutions to perform. If omitted, the default value is -1, which means make all possible substitutions.														
<i>compare</i>	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. (0 = case sensitive, 1 = case-insensitive) Built-in "vb" constants can be used for the <i>compare</i> argument: vbBinaryCompare for 0 (case-sensitive search) vbTextCompare for 1 (case-insensitive search)														
Examples:	strNewDate = Replace\$("08/31/2001", "/", "-") ' strNewDate = "08-31-2001"														

Function:	StrReverse\$ (or StrReverse)
Description:	Returns a string in which the character order of a specified string is reversed. Introduced in VB 6.
Syntax:	StrReverse\$(string)
Examples:	strTest = StrReverse\$("Visual Basic") ' strTest = "cisaB lausiV"

Function:	LTrim\$ (or LTrim)
Description:	Removes leading blank spaces from a string.
Syntax:	LTrim\$(string)
Examples:	strTest = LTrim\$(" Visual Basic ") ' strTest = "Visual Basic "

Function:	RTrim\$ (or RTrim)
Description:	Removes trailing blank spaces from a string.
Syntax:	RTrim\$(string)
Examples:	strTest = RTrim\$(" Visual Basic ") ' strTest = " Visual Basic"

Function:	Trim\$ (or Trim)
Description:	Removes both leading and trailing blank spaces from a string.
Syntax:	Trim\$(string)
Examples:	strTest = Trim\$(" Visual Basic ") ' strTest = "Visual Basic" ' Note: Trim\$(x) accomplishes the same thing as LTrim\$(RTrim\$(x))

Function:	Asc
Description:	Returns an Integer representing the ASCII character code corresponding to the first letter in a string.
Syntax:	Asc(string)
Examples:	intCode = Asc("*") ' intCode = 42 intCode = Asc("ABC") ' intCode = 65

Unit - 5 : Using Additional Controls and Control Array

Structure of Unit:

- 5.0 Objective
- 5.1 Introduction
- 5.2 Working with ListBoxes and ComboBoxes
- 5.3 Scroll Bars
- 5.4 Picture Boxes
- 5.5 The Shape Control
- 5.6 The Line Control
- 5.7 The Timer Control
- 5.8 Control Arrays
- 5.9 File List Box Control
- 5.10 Summary
- 5.11 Self Assessment Questions

5.0 Objective

After completion of this unit, you will be able to:

- Discuss various controls.
- Add the controls on the form and can change their properties.

5.1 Introduction

List boxes do just what their name implies: display a list of items. The user can make a selection from that list, and Visual Basic will inform our program what's going on. Because list boxes can use scroll bars if a list gets too long, these controls are very useful to present long lists of items in a way that doesn't take up too much space.

Combo boxes are list boxes combined with text boxes. With combo boxes, you can give users the option of selecting from a list (usually a drop-down list activated when users click the downwards-pointing arrow at right in a combo box) or typing their selections directly into the text box part of the combo box.

Every Windows user is familiar with scroll bars. If computers had wall-sized displays, we might not need scroll bars, but as it is, scroll bars help control what parts of your program's data are visible at any one time.

Picture boxes are more complete controls than image controls. Just as the rich text control provides a sort of word-processor-in-a-control, so the picture box does for graphics in Visual Basic.

5.2 Working with ListBoxes and ComboBoxes

List Box

ListBoxes can either be set up to allow to user to select one item from the list, or to allow multiple selections. Multiple selections are not allowed if you want to display check boxes next to the list items. If your list is too long for the size you've drawn the ListBox, scroll bars will appear automatically. When the user selects (clicks) a list item, it is highlighted and the ListBox's **ListIndex** property is automatically changed to hold a

value corresponding to the item. The first item in the list is numbered 0. Use the `ListIndex` property by itself to determine what the user selected, or use it in combination with the **List** property.

```
Private Sub lstChoices_Click( )
```

```
    MsgBox "You selected " & lstChoices.List(lstChoices.ListIndex)
```

```
End Sub
```

You can select an item in code by simply setting the `ListIndex`.

```
lstChoices.ListIndex = 2
```

Text property

The above on the `List` and `ListIndex` properties serves as an introduction to these properties, but there is an easier way to get the selected item.

```
lstChoices.Text
```

This code is equivalent to

```
lstChoices.List(lstChoices.ListIndex)
```

When no item is selected, the value of `ListIndex` will equal to -1. You could use that in an `If` statement to run code which required a selected item.

```
If lstChoices.ListIndex = -1 Then
```

```
    MsgBox "No item selected"
```

```
Else
```

```
    MsgBox "You selected " & lstChoices.Text
```

```
End If
```

AddItem

Creating the list for the `ListBox` can be done at design time in the Properties Window or at run time with the `AddItem` method. This method has one required argument and one optional argument.

```
lstChoices.AddItem "Rolls Royce"
```

```
lstChoices.AddItem txtCars.Text
```

```
lstChoices.AddItem sModel$, 5
```

The first line adds some literal text to the end of the list. The second line adds to the end of the list what the user has entered in a Text Box. The third example adds the value of a string variable to the list and uses the optional argument to give it a certain place within the list at the same time. This value would become the new item's `ListIndex` and all items already in the list with this `ListIndex` or higher would be pushed down one in the list and automatically given a new `ListIndex`.

RemoveItem method

The `RemoveItem` method deletes a list item; its one argument is the `ListIndex` of the item you wish to remove. Important to consider when removing list items is that the `ListIndex` of the remaining items will

change to reflect their new positions in the list. For example, to remove the first five items in a list all at once, you'd need to use a loop and delete `ListIndex 0` five times as shown in this example:

```
For i% = 1 To 5
    lstChoices.RemoveItem {0}
Next i%
```

Clear method

The `Clear` method deletes the entire list at once and requires no arguments. This would be useful if you were rebuilding the entire list at run time.

```
lstChoices.Clear
```

Sorted property

Setting the `Sorted` property to `True` will alphabetize your list, or in the case of numeric list items, arrange them from lowest to highest (but only by the first digit).

Multiple column ListBoxes

The standard “Open” dialog box, which most applications have, is an example of a `ListBox` with multiple columns. Setting the **`Columns`** property to the number of columns desired will create this for you and horizontal scroll bars are automatically added when the list doesn't entirely fit.

Once you create a multiple-column `ListBox` at design time, you cannot revert back to a single column at run time, but you can change the number of columns.

Multiple selection ListBoxes

You can set the `MultiSelect` property one of two ways to allow the user to select more than one item. With the *Simple* setting, a mouse click will select (highlight) or deselect any number of items in any order. With the *Extended* setting, the user can hold down the mouse and drag it to select several **consecutive** items. For keyboard fans, holding down the *Shift* key and pressing any of the arrow keys will accomplish the same thing.

Multiple selection can be used with multiple columns.

Detecting multiple selections is much different. The `ListIndex` property is no longer in play here. You must loop through the entire list, checking the **`Selected`** property for a `True` value.

```
For i% = 0 To lstChoices.ListCount - 1
    If lstChoices.Selected(i%) = True Then
        MsgBox "Item number " & i% & " is selected"
    End If
Next i%
```

To determine the number of items selected use the **`SelCount`** property.

```
If lstChoices.SelCount > 5 Then  
    MsgBox "Maximum 5 selections allowed"  
End If
```

NewIndex property

The NewIndex property holds the ListIndex value of the most recently added item. This could be useful after using AddItem on a long sorted list. Without worrying about the new item's ListIndex, you could identify the list item and further deal with it. The most common use for this is in conjunction with the **ItemData** property.

```
lstChoices.AddItem "Rhode Island"  
  
lstChoices.ItemData(lstChoices.NewIndex) = 13
```

The number 13 is now associated with the "Rhode Island" list item. (Rhode Island was the 13th state).

The value of NewIndex becomes -1 after using RemoveItem.

Scroll event

This event triggers when the user scrolls through the list, regardless of the scrolling method. By scrolling method, I mean clicking the up or down arrow, clicking within the scroll bar for faster up/down scrolling, or dragging the position indicator.

ComboBox

Most of what you already know about ListBoxes will apply to a ComboBox. Items are added, removed and cleared with the AddItem, RemoveItem, and Clear methods. List, ListIndex, ListCount and ItemData properties all the work the same way, however ComboBoxes cannot have multiple columns or handle multiple selections.

A ComboBox control combines the features of a TextBox control and a ListBox control—users can enter information in the text box portion or select an item from the list box portion of the control.

Style

The **Style** property is read only. That means you must decide which style to use at design time and an error will occur if you try to set the value of the property at run time. There are three styles to consider.

- The Simple style appears like you've drawn a TextBox just over a ListBox. If the user highlights (clicks) one of the items in the ListBox portion, the text automatically appears in the TextBox portion. Alternatively, the user can elect to input their own text directly into the TextBox.
- The Dropdown Combo is probably the most used style. This style can be drawn as wide as it needs to be but it's Height is limited to the space needed for one line of text. By default, the Text property will be set to "Combo1", so you should set it's initial value in the Properties Windows or in code (Form_Load event).
- The Dropdown List does not take user input. The Text property is read-only with this style, unless you assign a string in code which is already a list item.

```
Private Sub Form_Load()
```

```
    cboList.Text = cboList.List(3)
```

```
End Sub
```

Since the Dropdown Combo and Simple styles are part TextBox, you can use some of the TextBox properties like Locked, SelText, SelLength, and SelStart.

Events

Visual Basic programs are event-driven, meaning code will run only in response to events. Each object has its own set of events, but many are common to all objects. Events can be triggered by the user, such as clicking a button, triggered by your own code, and even by the actions of your program, such as start-up and shut-down. Visual Basic supports two type of events: Mouse Events & Keyboard Events.

Change

The most common use for this event is with a TextBox, but it does occur in other controls, like ComboBox, Label and some others. This event fires when the contents (most often the text) of a control have changed, so for every key a user types, you can check for valid data entry, or limit what can be entered.

Click

The most common event. Users expect something to happen when they click things, so you'll be writing a lot of code for this event. Just about every control has a click event.

Double Click

Abbreviated DblClick in a code window. Keep in mind that DblClick triggers two Click events, so don't write conflicting code in both events.

DragDrop

Triggered at the completion of a drag and drop operation. Or in other words, when the user releases the mouse after dragging an object to another location.

DragOver

Triggered constantly as an object is being dragged. Use this event to control or limit the drag and drop operation. Don't write any long drawn out code here.

Got Focus and Lost Focus

GotFocus occurs when the focus has switched to an object. This can be triggered by the user clicking on it, or tabbing to it. Since only one object can have the focus, another control will receive a LostFocus event at the same time.

Key Down, Key Up, and Key Press

Use these events to capture what the user is typing. More often, you'll use KeyDown, but you may run into a situation where you need to respond to releasing of a key. These two events can detect just about any key typed including the state (up or down) of the Ctrl, Alt, and Shift keys. To check for standard letter and number keys, use KeyPress, just remember one press of a key triggers all three events.

Mouse Down, Mouse Move, and Mouse Up

Similar to the Key events above, these events are triggered by mouse actions. MouseMove can be triggered several times per second, so let's not write complex calculations in this event.

Scroll

If an object has scroll bars (ListBox, ComboBox, etc.) this event fires when the user clicks the up or down arrow or drags the position indicator. This event does not apply to a TextBox with scroll bars.

Form Events

Several Form events will fire starting with the moment of creation. Each has its own purpose and triggers in a certain order. When the Form is first created, an Initialize event will occur, followed by Load, Resize, and Paint. Upon destruction of the Form, events triggered are QueryUnload, Unload, and Terminate.

Initialize

This event occurs only once when the Form is created. It will not fire again unless the Form is Unloaded and Loaded again.

Load

Occurs when a Form is loaded. This event can only be fired again if the form is unloaded and loaded again. If a Form loses the focus (thereby losing its status as the ActiveForm) and then becomes active again, an Activate event will fire.

Resize

Even though no physical resizing has occurred, this event triggers after Load. It will also be set off by minimizing, maximizing, restoring, and resizing.

Paint

This event fires for the purpose of drawing graphics on the form.

Activate and Deactivate

If a form becomes the ActiveForm (has the focus), Activate will fire, followed by GotFocus. If another form becomes active, the LostFocus event will occur, followed by Deactivate. However, GotFocus and LostFocus will only trigger when the form has no enabled controls.

QueryUnload

This event is used for checking to make sure it's OK to shut down. In this event, you can actually stop shut-down and even check if the user is trying to shut-down, your code invoked shut-down, or if Windows itself is responsible. QueryUnload has two built-in parameters for dealing with these situations- *Cancel* and *UnloadMode*.

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer
```

```
    If MsgBox("Are you sure you want to shut down?", vbYesNoCancel) <> vbYes Then
```

```
        Cancel = True
```

```
    End If
```

```
End Sub
```

Where Cancel = True means “cancel the unloading of this form”. You would probably check some program condition before deciding to set Cancel to True, or you could use a simple message like the above example.

Unload

After having passed through QueryUnload, this event fires and any code which aids in having that form out of memory is processed.

Terminate

Like Initialize, this event will only occur once during the existing of the Form. It won't fire if the program is terminated abnormally.

5.3 Scroll Bars

Standard scroll bars are intrinsic controls in Visual Basic, which means that they appear in the toolbox as soon as you start Visual Basic. You'll find both the Vertical and the Horizontal Scroll Bar tools in the toolbox; to add those controls to a form, just paint them as you need them in that form. You add the other controls in this chapter with the Project|Components menu item (click the Controls tab in the Components box that opens). To add flat scroll bars, you select the Microsoft Flat Scrollbar Control item; to add sliders, you select the Microsoft Windows Common Controls item; and to add the updown control, you click the Microsoft Windows Common Controls-2 item. The toolbox tools for these controls appear in Figure 5.1. The Horizontal Scroll Bar tool is fourth down in the middle, the Vertical Scroll Bar tool is fourth down on the right. The Updown tool is eighth down in the middle, the Slider tool is eleventh down on the right, and the Flat Scroll Bar tool is twelfth down in the middle.

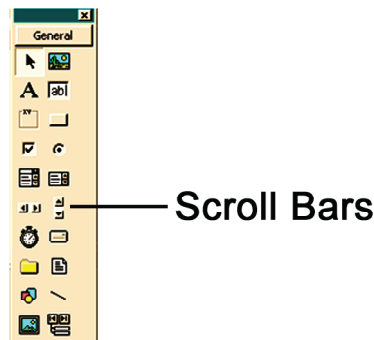


Figure 5.1 The Horizontal Scroll Bar, Vertical Scroll Bar, Updown, Slider, and Flat Scroll Bar tools.

In overview, these controls work in more or less the same way: you add them to a form, use Min and Max properties to set the possible range of values the user can set, then read the Value property to get the control's setting in a Change event handler to interpret actions undertaken by the user. Change events occur after the user is finished changing the control's setting; you can also use the Scroll event to handle events as the user works with the control, as we'll see in this chapter. In fact, we'll see how all this works and more in the Immediate Solutions, and we'll turn to that now.

Adding Horizontal or Vertical Scroll Bars To A Form

Many programmers think that there is one Scroll Bar tool that you add to a form and then set its orientation-vertical or horizontal. In fact, those are two different controls, as you see in the toolbox in Figure 5.2. To add a horizontal scroll bar to a form, you use the Horizontal Scroll Bar tool, and to add a vertical scroll bar, you use the Vertical Scroll Bar tool. A horizontal scroll bar,

HScroll1, and a vertical scroll bar, VScroll1, appear in Figure 5.2

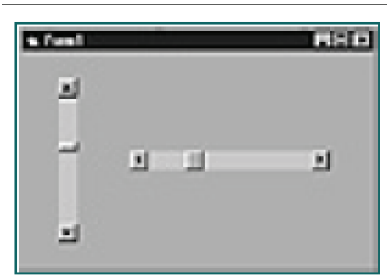


Figure 5.2 A horizontal and a vertical scroll bar.

Setting Scroll Bars' Minimum And Maximum Values

The Testing Department is calling again. The Field Testing Unit loves the new program you've written to help them record in-the-field performance of the company's products, but there's just one problem: performance is measured on a scale of 1 to 100, and the scroll bars in your program seem to go from 0 to 32767. It's been very hard for the users of your program to operate with only 1/32 of the whole scroll bar. Can you rescale it? Yes, you can. After you place a scroll bar in a program, the first thing to do is to set its range of possible values, which by default

is 0 to 32767. The minimum value a scroll bar can be set to is stored in its Min property, and the maximum value in the Max property. You can set the Min and Max properties for scroll bars at design time or at runtime; here's how we change those properties in a vertical scroll bar:

```
Private Sub Form_Load()  
    VScroll1.Min = 1  
    VScroll1.Max = 100  
End Sub
```

5.4 Picture Boxes

Picture boxes are more complete controls than image controls. Just as the rich text control provides a sort of word-processor-in-a-control, so the picture box does for graphics in Visual Basic. You can load images into a picture box, save images to disk, draw with some rudimentary graphics methods, print images, work pixel-by-pixel, set an image's scale, and more. Besides graphics handling, the picture box can also act as a container for other controls-and besides toolbars and status bars, it's the only control that can appear by itself in an MDI form.

As with image controls, you load pictures into a picture box's Picture property, and you can do that at design time or runtime (at runtime you use the LoadPicture() method). When you load an image into a picture box, the picture box does not resize itself by default to fit that image as the image control does-but it will if you set its AutoSize property to True. The picture box has a 3D border by default, so it doesn't look like an image control-unless you set its BorderStyle property to 0 for no border (instead of 1, the default). In other words, you can make a picture box look and behave just like an image control if you wish, but keep in mind that picture boxes use a lot more memory and processor time, so if you just want to display an image, stick with image controls. Like image controls, picture boxes are intrinsic controls in Visual Basic; the Picture Box tool is at right in the first row of tools in Figure 5.3.

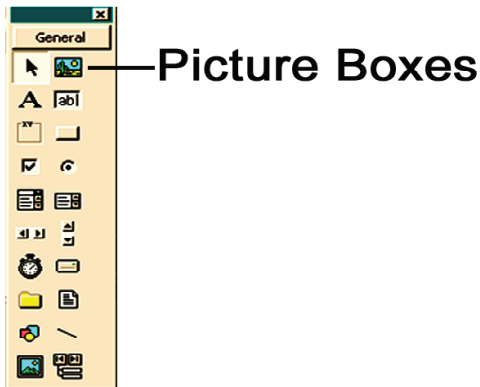


Figure 5.3 The Picture Box tool

5.5 The Shape Control

The shape control is a graphical control. You can use this control to draw predefined colored and filled shapes, including rectangles, squares, ovals, circles, rounded rectangles, or rounded squares.

You use the shape control at design time to draw shapes in a form. There's no great programming complexity here—you just use this control as a design element to add rectangles, circles, and so on to your forms. In this way, the shape control is a little like the frame control; however, shapes can't act as control containers (for example, you can't group option buttons together with shapes or move the controls inside them en masse). Still, shapes certainly come in more varieties than frames do. Although shape controls are one of the Visual Basic intrinsic controls, Visual Basic programmers remain largely ignorant of them. That's too bad, because you can create some nice effects with shapes, as we'll see here. The Shape Control tool appears in the Visual Basic toolbox in Figure 5.4 as the ninth tool down on the left.

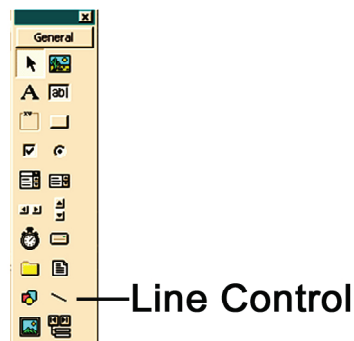


Figure 5.4 The Shape Control and Line Control tools.

5.6 The Line Control

Like the shape control, the line control is a graphical control. You use it to display horizontal, vertical, or diagonal lines in a form. You can use these controls at design time as a design element or at runtime to alter the original line you drew.

Drawing lines is easy—you just click the Line Control tool in the toolbox, press the mouse button when the cursor is at the line's start location on the form, and drag the mouse to the end position of the line. When you release the mouse, the line appears with sizing handles at each end that you can use to change the line as you like. You can also change a line at runtime by changing its X1, X2, Y1, and Y2 properties. You can draw

lines with this control in forms, picture boxes, and frames. In fact, lines drawn with the line control stay visible even if its container's `AutoRedraw` property is set to `False`. (The line control even has its own `Visible` property, which means you can make lines appear and disappear.) The Line Control tool appears in the toolbox in Figure 5.4 as the ninth tool down on the right.

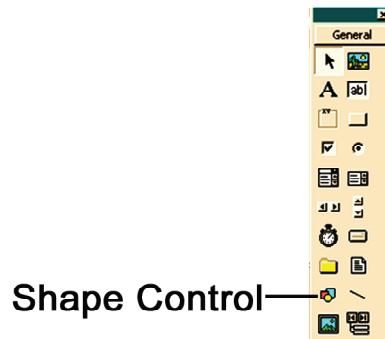


Figure 5.5 The Shape Control and Line Control tools.

5.7 The Timer Control

You use a timer control when you want to execute code at specific intervals. To use a timer, you add a timer control to your program (timers are one of the intrinsic controls that appear in the toolbox when you start Visual Basic) and set its `Interval` property. From then on, while the timer is enabled, it creates `Timer` events, which are handled in an event handling procedure, like `Timer1_Timer()`. You place the code you want executed each interval in that procedure. To add a timer to your program, use the Timer Control tool in the toolbox, which is the seventh tool down on the left in Figure 5.6.

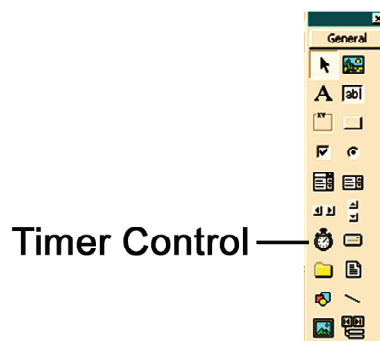


Figure 5.6 The Timer Control tool.

We should note, that however, that there are a few issues about using the `Interval` property. Although measured in milliseconds (1/1000s of a second), `Timer` events cannot actually occur faster than 18.2 times a second (this is the period of the computer's timer interrupt). The interval can be set to values between 0 (in which case nothing happens) and 64,767, which means that even the longest interval can't be much longer than 1 minute (about 64.8 seconds). Of course, you can design your code to wait for several intervals to pass before doing anything. You shouldn't count on a timer too closely if your task execution is dependent on exact intervals; if the system is busy executing long loops, intensive calculations, or drive, network, or port access (in which case software routinely disables the timer interrupt), your application may not get `Timer` events as often as the `Interval` property specifies. That is to say, `Timer` events are not guaranteed to happen exactly on time. If you need to be sure, your software should check the system clock when it needs to (using, for example, the Visual Basic `Time$` function), rather than try to keep track of time internally.

Another point here has to do with Windows programming philosophy. Using a timer can easily full programmers back to thinking in terms of sequential programming (as in the DOS days), rather than event-oriented programming. When you use a timer, your code has a lot of control and can get a lot of execution time, because your code is called each time the timer ticks. However, that doesn't mean you should set a timer interval short and put in all kinds of loops. Remember that Windows is built around user events, not programs that are designed to retain control for long periods of time. Other programs will probably be running at the same time as yours, so it's

considerate not to use timers simply to wrest control from the environment.

5.8 Creating Button Control Arrays

You've decided that your new game program really does need 144 buttons in the main form, arranged in a grid of 12×12. But what a pain it is to write 144 sub-routines to handle the click event for each of them! Isn't there a better way? There is. You use a control array and one event handler function (the control array index of the button that was clicked is passed to the event handler, so you can tell which button you need to respond to). To create a control array, just give two controls of the same type the same name (in the Name property); when you do, Visual Basic will ask if you want to create a control array, as in Figure 5.7.



Figure 5.7 Creating a control array.

When you create an event handler subroutine for a button in the control array, Visual Basic will automatically pass the index of the control in the control array to that subroutine:

```
Private Sub GamePiece_Click(Index As Integer)

End Sub
```

You can then refer to the control that caused the event as a member of an array, using the index passed to the subroutine:

```
Private Sub GamePiece_Click(Index As Integer)

    GamePiece(Index).Caption = "You clicked me!"

End Sub
```

5.9 Using The File List Box Control

The file list box control lets you display the files in a directory as a list of names. This control's tool appears as the eighth tool down on the right in Figure 5.8. To add this control to a form, just draw it as you want it with its tool in the toolbox.

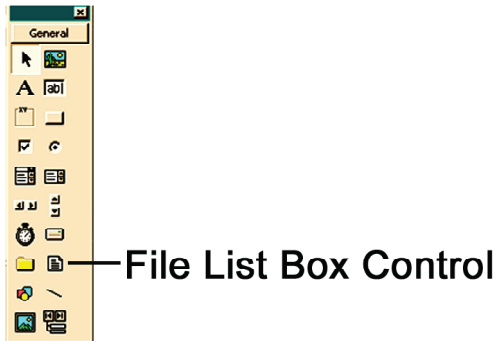


Figure 5.8 The File List Box Control tool.

The important properties of the file list box are the Path and FileName properties. Let's see an example using the drive, directory, and file list boxes. When the user selects a file and clicks a button labeled Display File, or double-clicks the file's name in the file list box, we'll display the contents of the selected file in a text box.

We start by adding the controls we'll need: a drive list box, Drive1; a directory list box, Dir1; a file list box, File1; a command button, Command1, which is labeled Display File; and a text box with its MultiLine property set to True and its Scrollbars property set to Both (if the file you are displaying is too long for a text box, use a rich text box). When the user changes the drive, we pass that new drive to the directory list box as the new directory in Drive1_Change():

```
Sub Drive1_Change()
    Dir1.Path = Drive1.Drive
End Sub
```

When the user changes the directory, we pass that new path to the file list box in Dir1_Change():

```
Sub Drive1_Change()
    Dir1.Path = Drive1.Drive
End Sub
```

When the user changes the directory, we pass that new path to the file list box in Dir1_Change():

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

When the user clicks the button, we want to display the contents of the selected file in the text box, and we'll do that in the command button's Click event handler, Command1_Click(). We'll also call the button's Click event handler to let the user open a file by double-clicking it in the file control:

```
Sub File1_DblClick()
    Command1_Click
End Sub
```

When the user wants to open a file, we put together the file's name and path this way:

```

Sub Command1_Click()
Dim FileName As String
On Error GoTo FileError
If(Right$(Dir1.Path, 1) = "\") Then
FileName = File1.Path & File1.FileName
Else
FileName = File1.Path & "\" & File1.FileName
End If
...

```

Then we simply open the file and display it in the text box, Text1:

```

Sub Command1_Click()
Dim FileName As String
On Error GoTo FileError
If(Right$(Dir1.Path, 1) = "\") Then
FileName = File1.Path & File1.FileName
Else
FileName = File1.Path & "\" & File1.FileName
End If
Open FileName For Input As #1
Text1.Text = Input$(LOF(1), #1)
Close #1
Exit Sub
FileError:
MsgBox "File Error!"
End Sub

```

That's it-when you run the program, the user can use the file controls to open a file, as shown in Figure 5.9. Now we're using the Visual Basic file controls.

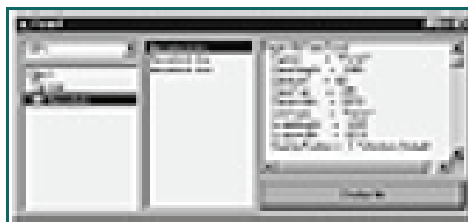


Figure 5.9 Displaying a file using the Visual Basic file controls.

5.10 Summary

A Control is an object that enables users to interact with the application. These are the building blocks of an application. Each control has a unique set of properties, which determine its appearance and behavior.

A control has a name property that distinguishes it from other objects in a project. There are other properties as well which can be changed.

5.11 Self Assessment Questions

- 1 On your computer, start the visual basic software. Once the Visual Basic is loaded, perform the following operations:

Place a Picture control on the form. Change its location.

Load any picture on Picture control. Set its properties as per your choice.

Place one or two Label5 and Text boxes on the form.

Change label caption.

Change label size.

Place a command button on the form.

Write code, for the command button - whenever click on it, Project must be close.

- 2 Write a program to move elements of listbox to another given listbox.
- 3 Create an application that shows current system date and time. Add option buttons to select the format of the time and date to be displayed.
- 4 Create an executable file of any application (say Text Editor application). Hint: Look out for Make Exe option in the File menu.
- 5 Can status bar display an icon? If yes, then create an application that shows an icon at the status bar.

In the Text Editor application above, which shows the use of scrollbars, option buttons, and check boxes, implement the combo boxes for the choice, instead of check boxes.

Unit - 6 : Menu

Structure of Unit:

- 6.0 Objectives
- 6.1 Introduction
- 6.2 Menu Basic and Editor
- 6.3 Accessing Keys and Shortcut Keys
- 6.4 Separator bar
- 6.5 Pop-Up Menus
- 6.6 Adding And Deleting Menu Items At Runtime
- 6.7 Summary
- 6.8 SelfAssessment Questions

6.0 Objectives

After the completion of this unit, you will be able to:

- Use data control and data aware control;
- Create a form using data controls and can access data;
- Create a menu-bar and can add different options.

6.1 Introduction

Menus give your application a professional look. Menus can contains a large number of logically organized user options yet they take up only a small amount of screen space. Everything your application does should be accessible through the menus.

A menu system consists of a bar that displays a row of menu titles. When programmer clicks a menu title, its menu, which includes a list of menu items, is displayed. When a menu items is clicked, a menu click event is triggered to allow a program response to the menu the menu selection.

Each menu item can have its own submenu, which appears when the mouse is moved horizontally to the next child menu. There can be up to four menus. Items that have child menus are identified by a graphical arrow on the right side of the item. Menu titles and items can be disabled to make them unavailable, or made invisible to hide them from the user.

In order to make your menus user-friendly, you should adhere to some simple guidelines, like using File, Edit, View, Options, and Help as top-level menu titles. VB has a built-in menu editor that you will use and it's pretty much a no-brainer.

6.2 Menu Basic and Editor

Menus are so tightly integrated into the Visual Basic application that very little programming is required to create and program them. Visual Basic provides a tool called the Menu Editor to make it easy to create and modify menus.

To bring up the Menu Editor, make sure the Form designer is the active window and click Tools/Menu Editor or press Ctrl+E. The caption and the name fields are required. There are three logical sections in the Menu Editor Dialog box.

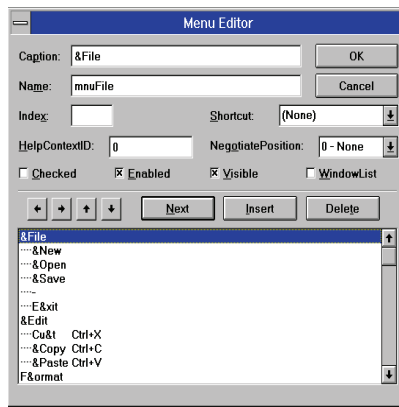


Figure 6.1: Menu Editor

Caption:

This is what the user will see. Make the caption as short *and* as descriptive of its action as you possibly can. Use proper case. Allows us to enter the menu or command name that we want to appear on our menu bar or in a menu.

Name:

Menu items are controls and must have a name, the same as any other control. Allows us to enter a control name for the menu item. A control name is an identifier used only to access the menu item in code; it doesn't appear in a menu. But the Menu Editor does not supply a default, so use the mnu prefix.

Example: mnuFile, mnuEdit

You may want to layout the entire menu on paper before beginning with the Menu Editor. In a carefully planned menu, you'll type the Caption, then the Name, the Next Button and the left or right arrow to indent or outdent the next menu item within the structure.

6.3 Accessing Keys and Shortcut Keys

Look at the menu of your browser. You'll see menu titles with one letter underlined. This enables users to press ALT, then the underlined letter in the menu title, thereby displaying that menu. Furthermore, the user can type the underlined letter in one of the menu items in order to perform that action.

To actually make access keys, you type your caption with an ampersand (&) just before the designated letter. For example, the File title's caption is &File. Most often, the first letter of the caption is the access key, but you may need to use a different letter to avoid conflict with another menu item. Each menu title is separate when considering access keys.

Assign shortcut keys in the Menu Editor's Shortcut field. There are some standard Windows shortcut keys to consider like Ctrl+X, Ctrl+C, and Ctrl+V for cut, copy and pasting actions.

6.4 Separator bar

Putting a separator bar in the menu is simple, just use a dash for the menu item's Caption. The separator bar is actually still a menu control, so it must have a name.

Checked, Enabled, and Visible

Like other controls, menu items can be grayed out by setting Enabled to False, or completely removed by setting Visible to False. (At least one sub-menu must be visible.) You can also display a check mark next to a menu item indicating whether a toggle option is turned on or off.

Enabled, Allows us to select whether we want the menu item to respond to events, or clear if we want the item to be unavailable and appear dimmed.

Visible, Allows us to have the menu item appear on the menu.

Index

Menu items can be part of a control array.

WindowList

A menu can be a list of MDI child windows. The list is handled automatically by the MDI Form.

NegotiatePosition

Used with object linking and embedding (OLE). It allows us to select the menu's NegotiatePosition property. This property determines whether and how the menu appears in a container form.

PopupMenu method

Your menus don't have to be located in the menu bar. With the PopupMenu method, you can make a menu appear anywhere on the form. You can also make the menu appear upon right-clicking the mouse. If you don't want the menu in the menu bar, just set the **menu title's** Visible property to False. Let's say you want a menu to pop up when the user right-clicks or left-clicks on the form.

```
Private Sub Form_Click()  
    PopupMenu mnuPopup, vbPopupMenuRightButton  
End Sub
```

To make the menu respond to *only* a right-click, use the MouseDown event instead and inspect the Button argument.

By default, the pop-up menu will appear below and to the right of the mouse cursor, but this can be controlled by specifying alignment constants.

‘Menu appears below and centered on the mouse cursor

```
PopupMenu mnuPopup, vbPopupMenuCenterAlign
```

‘Menu appears below and to the left of the mouse cursor

```
PopupMenu mnuPopup, vbPopupMenuRightAlign
```

‘Menu is centered and responds to right-clicks

‘Use the Or operator to combine settings

```
PopupMenu mnuPopup, vbPopupMenuCenterAlign Or vbPopupMenuRightButton
```

HelpContextID

Allows us to assign a unique numeric value for the context ID. This value is used to find the appropriate Help topic in the Help file identified by the HelpFile property.

HELP Menu

When you create a program, the IDE allows you to define the .hlp file which is displayed by WinHelp when you press F1. But be sure to include a Help menu on the top of your form. Users will expect it. In that menu, you should include at least two selections - “Contents” and “About”. The contents should go to the contents section of the help file, but the about selection is normally used to call up a small window which gives version and author information about the program.

Also, put the Help menu at the far right of the menu selections. Your users are used to seeing it there and they will expect to find it in the same place as all other Windows programs that they use.

Example: Note Editor

1. We will build a note editor with a menu structure that allows us to control the appearance of the text in the editor box.

Start a new project.

2. Place a large text box on a form. Set the properties of the form and text box:

Form1:

BorderStyle	1-Fixed Single
Caption	Note Editor
Name	frmEdit

Text1:

BorderStyle	1-Fixed Single
MultiLine	True
Name	txtEdit
ScrollBars	2-Vertical
Text	[Blank]

The form should look something like this when you're done:

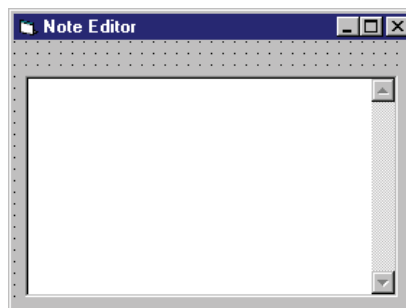


Figure 6.2

3. Now to add this menu structure to the Note Editor:

<u>F</u>ile	<u>F</u>ormat
<u>N</u> ew	<u>B</u> old
	<u>I</u> talic
<u>E</u> xit	<u>U</u> nderline
	<u>S</u> ize
	<u>S</u> mall
	<u>M</u> edium
	<u>L</u> arge

Note the identified access keys. Bring up the Menu Editor and assign the following Captions, Names, and Shortcut Keys to each item. Make sure each menu item is at its proper location in the hierarchy.

Caption	Name	Shortcut
&File	mnuFile	[None]
&New	mnuFileNew	[None]
-	mnuFileBar	[None]
E&xit	mnuFileExit	[None]
F&ormat	mnuFmt	[None]
& Bold	mnuFmt Bold	Ctrl+B
&Italic	mnuFmtItalic	Ctrl+I
&Underline	mnuFmtUnderline	Ctrl+U
&Size	mnuFmtSize	[None]
&Small	mnuFmtSizeSmall	Ctrl+S
&Medium	mnuFmtSizeMedium	Ctrl+M
&Large	mnuFmtSizeLarge	Ctrl+L

The **Small** item under the **Size** sub-menu should also be **Checked** to indicate the initial font size. When done, look through your menu structure in design mode to make sure it looks correct. With a menu, the form will appear like:

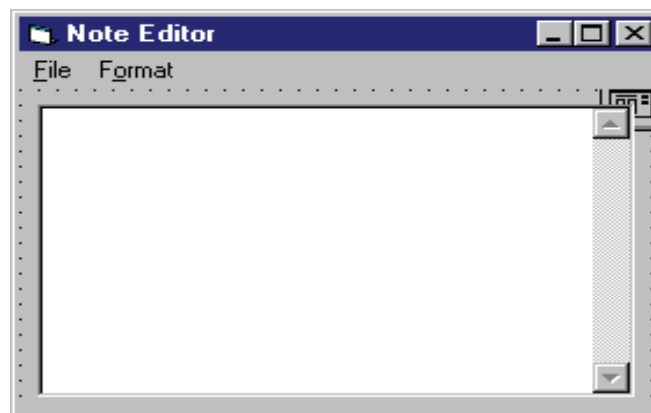


Figure 6.3

4. Each menu item that performs an action requires code for its **Click** event. The only menu items that do not have events are the menu and sub-menu headings, namely File, Format, and Size. All others need code. Use the following code for each menu item **Click** event.

If **mnuFileNew** is clicked, the program checks to see if the user really wants a new file and, if so (the default response), clears out the text box:

```
Private Sub mnuFileNew_Click()  
    'If user wants new file, clear out text  
  
    Dim Response As Integer  
  
    Response = MsgBox("Are you sure you want to start a new file?", vbYesNo + vbQuestion,  
        "New File")  
  
    If Response = vbYes Then txtEdit.Text = ""  
  
End Sub
```

If **mnuFileExit** is clicked, the program checks to see if the user really wants to exit. If not (the default response), the user is returned to the program:

```
Private Sub mnuFileExit_Click()  
    'Make sure user really wants to exit  
  
    Dim Response As Integer  
  
    Response = MsgBox("Are you sure you want to exit the note editor?", vbYesNo + vbCritical +  
        vbDefaultButton2, "Exit Editor")  
  
    If Response = vbNo Then  
  
        Exit Sub  
  
    Else  
  
        End  
  
    End If  
  
End Sub
```

If **mnuFmtBold** is clicked, the program toggles the current bold status:

```
Private Sub mnuFmtBold_Click()  
    'Toggle bold font status  
  
    mnuFmtBold.Checked = Not (mnuFmtBold.Checked)  
  
    txtEdit.FontBold = Not (txtEdit.FontBold)  
  
End Sub
```

If **mnuFmtItalic** is clicked, the program toggles the current italic status:

```
Private Sub mnuFmtItalic_Click()
```

‘Toggle italic font status

```
mnuFmtItalic.Checked = Not (mnuFmtItalic.Checked)
```

```
txtEdit.FontItalic = Not (txtEdit.FontItalic)
```

```
End Sub
```

If **mnuFmtUnderline** is clicked, the program toggles the current underline status:

```
Private Sub mnuFmtUnderline_Click()
```

‘Toggle underline font status

```
mnuFmtUnderline.Checked = Not (mnuFmtUnderline.Checked)
```

```
txtEdit.FontUnderline = Not (txtEdit.FontUnderline)
```

```
End Sub
```

If either of the three size sub-menus is clicked, indicate the appropriate check mark location and change the font size:

```
Private Sub mnuFmtSizeSmall_Click()
```

‘Set font size to small

```
mnuFmtSizeSmall.Checked = True
```

```
mnuFmtSizeMedium.Checked = False
```

```
mnuFmtSizeLarge.Checked = False
```

```
txtEdit.FontSize = 8
```

```
End Sub
```

```
Private Sub mnuFmtSizeMedium_Click()
```

‘Set font size to medium

```
mnuFmtSizeSmall.Checked = False
```

```
mnuFmtSizeMedium.Checked = True
```

```
mnuFmtSizeLarge.Checked = False
```

```
txtEdit.FontSize = 12
```

```
End Sub
```

```
Private Sub mnuFmtSizeLarge_Click()
```

‘Set font size to large

```
mnuFmtSizeSmall.Checked = False
```

```
mnuFmtSizeMedium.Checked = False
```

```
mnuFmtSizeLarge.Checked = True
```

txtEdit.FontSize = 18

End Sub

6.5 Pop-Up Menus

Assigning Icons to Forms

- * Notice that whenever you run an application, a small icon appears in the upper left hand corner of the form. This icon is also used to represent the form when it is minimized at run-time. The icon seen is the default Visual Basic icon for forms. Using the **Icon** property of a form, you can change this displayed icon.
- * The idea is to assign a unique icon to indicate the form's function. To assign an icon, click on the Icon property in the Property Window for the form. Click on the ellipsis (...) and a window that allows selection of icon files will appear.
- * The icon file you load must have the **.ico** filename extension and format. When you first assign an icon to a form (at design time), it will not appear on the form. It will only appear after you have run the application once.

Designing Your Own Icon with IconEdit

- * Visual Basic offers a wealth of icon files from which you could choose an icon to assign to your form(s). But, it's also fun to design your own icon to add that personal touch.
- * *PC Magazine* offers a free utility called **IconEdit** that allows you to design and save icons. Included with these notes is this program and other files (directory IconEdit). To install these files on your machine, copy the folder to your hard drive.
- * To run IconEdit, click **Start** on the Windows 95 task bar, then click **Run**. Find the **IconEdit.exe** program (use Browse mode). You can also establish an shortcut to start IconEdit from your desktop, if desired. The following Editor window will appear:

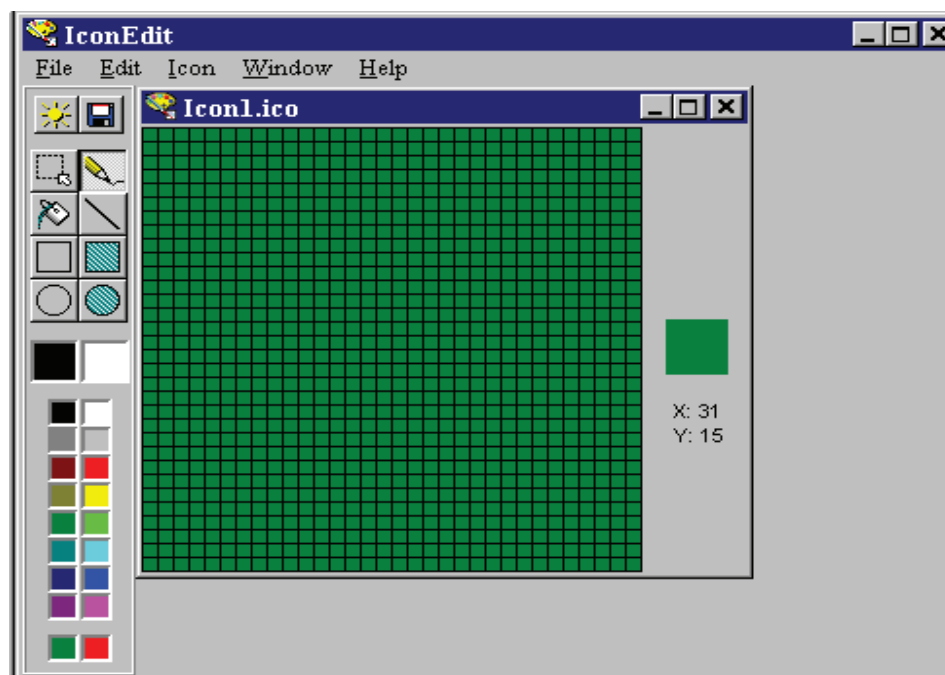


Figure 6.4

- * The basic idea of IconEdit is to draw an icon in the 32 x 32 grid displayed. You can draw single points, lines, open rectangles and ovals, and filled rectangles and ovals. Various colors are available. Once completed, the icon file can be saved for attaching to a form.
- * Another fun thing to do with IconEdit is to load in Visual Basic icon files and see how much artistic talent really goes into creating an icon.
- * We won't go into a lot of detail on using the IconEdit program.

6.6 Adding And Deleting Menu Items At Runtime

We've all seen menus that change as a program runs, and that can be a sophisticated effect. It's also impressive if the menu can change in response to user input (for example, adding a new item with the caption "Create Progame.exe", where Progame is the name given the program). You can add this capability to your program in Visual Basic.

Here, we'll just add new items-Item 1, Item 2, and so on-to the File menu with the user clicks a button. We start by designing our menu system, giving it a File menu with two items: New and Items, as you can see in Figure 6.5.



Figure 6.5 Designing an extendable menu.

The Items item is actually a placeholder for the items we'll add to the File menu. Make this item into a control array by giving it an index, 0, in the Index box, as shown in Figure 6.5. This item is just a placeholder—we don't want it to be visible before the user adds items to this menu—so set its Visible property to False, as also shown in Figure 6.5.

Now add a button to the program, and give it a Click event-handling function:

Private Sub Command1_Click()

End Sub

We'll keep track of the items in the File menu with a variable named `intItemCount`, which we increment each time the button is clicked:

Private Sub Command1_Click()

Static intItemCount

```
intItemCount = intItemCount + 1
```

...

```
End Sub
```

To add a new item to the Items control array, we use Load():

```
Private Sub Command1_Click()  
    Static intItemCount  
    intItemCount = intItemCount + 1  
    Load mnuFileItems(intItemCount)  
    ...  
End Sub
```

Finally, we set the caption of the item to indicate what its item number is, and make it visible:

```
Private Sub Command1_Click()  
    Static intItemCount  
    intItemCount = intItemCount + 1  
    Load mnuFileItems(intItemCount)  
    mnuFileItems(intItemCount).Caption = "Item " & intItemCount  
    mnuFileItems(intItemCount).Visible = True  
End Sub
```

You can also add a Click event handler to the Items menu item (because it's not visible in the menu bar, find mnuFileItems in the code window and add the event handler to it there). This event handler is passed the index of the clicked item in the control array, so we can indicate to the user which item he has clicked:

```
Private Sub mnuFileItems_Click(Index As Integer)  
    MsgBox ("You clicked item " + Str(Index))  
End Sub
```

That's it-now the File menu can grow as you like, as shown in Figure 6.6.



Figure 6.6 Adding items to a menu at runtime.

To remove items from the menu, just use Unload() statement like this (and make sure you adjust the total item count):

```
Unload mnuFileItems(intItemCount)
```

6.7 Summary

Data controls are used in conjunction with other controls to display, add, modify, and delete data from a database. Visual Basic provides a visual tool for building menu bars.

6.8 Self Assessment Questions

- 1 Create an application having menu bar and tool bar to create a text file, navigate and open text files, edit text file and save changes made by the user.

 Place a Data Control on the form.

 Select DatabaseName 'MYDB'.

 Set Recordset type with Dynaset.

 Set Recordsource with ITEMS table.

 Place a Grid control on the form.

 Set Datasource with datacontrol name.
- 2 Create a menu bar for insert a blank record, Save the record, Delete a record and Close the Project.
- 3 How do you move a menu item around with in the menu editor?
- 4 What is default location behavior of a pop-up or short cut menu?
- 5 What is a sub-menu?

Unit - 7 : Dialog boxes and Mouse Events

Structure of Unit:

- 7.0 Objectives
- 7.1 Introduction
- 7.2 Concept of Dialog boxes
 - 7.2.1 Standard Dialog box
 - 7.2.2 Custom Dialog box
 - 7.2.3 Common Dialog Control
- 7.3 Concept of Mouse Events
 - 7.3.1 Mouse events
- 7.4 Summary
- 7.5 Self Assessment Questions

7.0 Objectives

After completing this unit, you will be able to:

- Understand the concept of Dialog Box and Mouse Events.
- Understand the standard dialog box
- Know about custom and control dialog box
- Learn the Use mouse events.

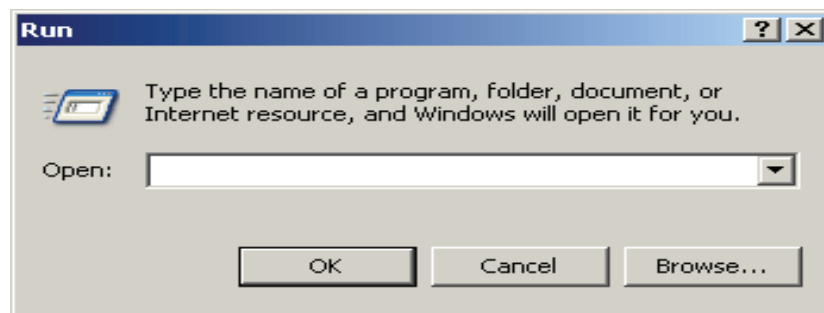
7.1 Introduction

A dialog box is a form defined with particular properties. Like a form, a dialog box is referred to as a container. Like a form, a dialog box is mostly used to host child controls, insuring the role of dialog between the user and the machine.


Visual Basic responds to various mouse events, which are recognized by most of the controls. The main events are MouseDown, MouseUp and MouseMove. MouseDown occurs when the user presses any mouse button and MouseUp occurs when the user releases any mouse button. These events use the arguments button, Shift, X, Y and they contain information about the mouse's condition when the button is clicked.

7.2 Concept of Dialog boxes

A dialog box is a form defined with particular properties. Like a form, a dialog box is referred to as a container. Like a form, a dialog box is mostly used to host child controls, insuring the role of dialog between the user and the machine. Here is an example of a dialog box:



A dialog box has the following characteristics:

- The only system button it is equipped with is Close . As the only system button, this button allows the user to dismiss the dialog and ignore whatever the user would have done on the dialog box
- It cannot be minimized, maximized, or restored. A dialog box does not have any other system button but Close
- It is usually modal, in which case the user is not allowed to continue any other operation on the same application until the dialog box is dismissed
- It provides a way for the user to close or dismiss it

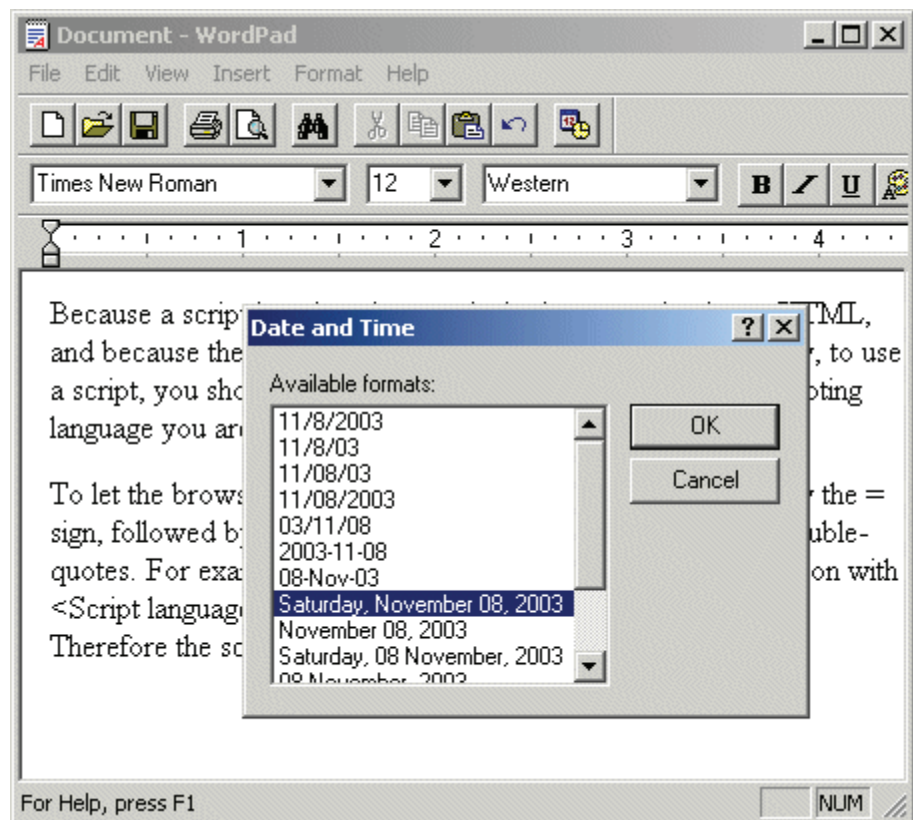
Modal and Modeless Dialog Box

There are two types of dialog boxes: modal and modeless.

A Modal dialog box is one that the user must first close in order to have access to any other framed window or dialog box of the same application. One of the scenarios in which you use a dialog box is to create an application that is centered around one. In this case, if either there is no other form or dialog box in your application or all the other forms or dialog boxes depend on this central dialog box, it must be created as modal. Such an application is referred to as dialog-based.

Some applications require various dialog boxes to complete their functionality. When in case, you may need to call one dialog box from another and display it as modal. Here is an example:

The Date and Time dialog box of WordPad is modal: when it is displaying, the user cannot use any other part of WordPad unless he or she closes this object first



After creating a dialog used as an addition to an existing form or an existing dialog box, to call it as modal, use the ShowDialog() method.

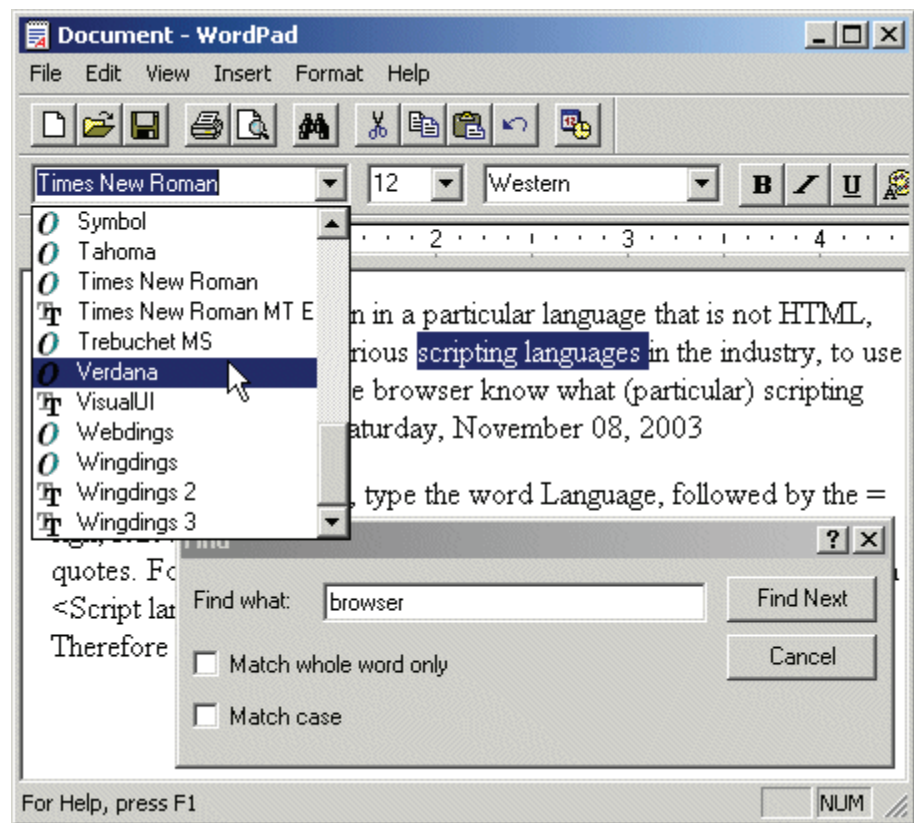
Modeless Dialog Box

A dialog box is referred to as modeless if the user does not have to close it in order to continue using the application that owns the dialog box. A modeless dialog box has the following characteristics

- It has a thin border
- It can be neither minimized nor maximized. This means that it is not equipped with the Minimize or the Maximize buttons
- It is not represented on the taskbar with a button
- It must provide a way for the user to close it

Here is an example:

The Find (and the Replace) dialog box of WordPad (also the Find and the Replace dialog boxes of most applications) is an example of a modeless dialog box. If it is opened, the user does not have to close it in order to use the application or the document in the background.



Since the modeless dialog box does not display its button on the task bar, the user should know that the dialog box is opened. To make the presence of a modeless dialog box obvious to the user, it typically displays on top of its host application until the user closes it.

A modeless dialog box is created from a form but it should look like a regular dialog box or a tool window. Therefore, to create a modeless dialog box, set the `FormBorderStyle` property to an appropriate value such as `FixedSingle`, `FixedToolWindow`, `Sizable` or `SizableToolWindow`. Also, set its `ShowInTaskbar` property to `False`.

After creating the dialog box, to display it as modeless, call the `Show()` method. The fundamental difference between the `ShowDialog()` and the `Show()` methods is that the former displays a modal dialog box, which makes sure that the called dialog box cannot go in the background of the main application. By contrast, the `Show()` method only calls the dialog box every time it is requested. For this reason, it is up to you to make sure that the modeless dialog box always remains on top of the application. This is easily taken care of by

setting the Boolean TopMost property of the form to True.

There are two main ways a normal modeless dialog box can be dismissed:

- If the user has finished using it, he or she can close it and recall it at will
- When the form or application that owns the modeless dialog box is closed, the form or application closes the modeless dialog if it is opened; this means that you don't need to find out whether a modeless dialog box is still opened when the application is being destroyed: either the user or the application itself will take care of closing it

7.2.1 Standard Dialog box

Message Box

The MsgBox function displays a message in a dialog box, waits for the user to click a button, and returns an Integer indicating which button the user clicked.

Syntax:

MsgBox(prompt[, buttons] [, title] [, helpfile, context])

The MsgBox function syntax has these parts:

Part	Description
prompt	Required. String expression displayed as the message in the dialog box. The maximum length of prompt is approximately 1024 characters, depending on the width of the characters used.
buttons	Optional. Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. If omitted, the default value for buttons is 0 (which causes only an OK button to be displayed with no icon). The buttons argument is explained in more detail below.
Title	Optional. String expression displayed in the title bar of the dialog box. If you omit title , the application name is placed in the title bar.
helpfile and context	Both optional. These arguments are only applicable when a Help file has been set up to work with the application.





The buttons argument

The first group of values (0 to 5) describes the number and type of buttons displayed in the dialog box; the second group (16, 32, 48, 64) describes the icon style; the third group (0, 256, 512, 768) determines which button is the default; and the fourth group (0, 4096) determines the modality of the message box. When adding numbers to create a final value for the buttons argument, use only one number from each group.

First Group - Determines which buttons to display:

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

Second Group - Determines which icon to display:

Constant	Value	Description	Icon
vbCritical	16	Display Critical Message icon.	
vbQuestion	32	Display Warning Query (question mark) icon.	
vbExclamation	48	Display Warning Message icon.	
vbInformation	64	Display Information Message icon.	

Third Group - Determines which button is the default:

Constant	Value	Description
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default (applicable only if a Help button has been added).

Fourth Group Determines the modality of the message box. Note the generally, you would not need to use a constant from this group, as you would want to use the default (application modal). If you specified "system modal", you would be "hogging" Windows i.e., if a user had another app open, like Word or Excel, they would not be able to get back to it until they responded to your app's message box.

Constant	Value	Description
vbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application.
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box.

There is a fifth group of constants that can be used for the buttons argument which would only be used under special circumstances:

Constant	Value	Description
vbMsgBoxHelpButton	16384	Adds Help button to the message box
VbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window
vbMsgBoxRight	524288	Text is right aligned
vbMsgBoxRtlReading	1048576	Specifies text should appear as right-to-left reading on Hebrew and Arabic systems

When you use MsgBox to with the option to display more than one button (i.e., from the first group, anything other than "vbOKOnly"), you can test which button the user clicked by comparing the return value of the MsgBox function with one of these values:

Constant	Value	Description
vbOK	1	The OK button was pressed
vbCancel	2	The Cancel button was pressed
vbAbort	3	The Abort button was pressed
vbRetry	4	The Retry button was pressed
vbIgnore	5	The Ignore button was pressed
vbYes	6	The Yes button was pressed
vbNo	7	The No button was pressed

Note: To try any of the MsgBox examples, you can simply start a new project, double-click on the form, and place the code in the Form_Load event.

There are two basic ways to use MsgBox, depending on whether or not you need to know which button the user clicked.

If you do NOT need to test which button the user clicked (i.e., you displayed a message box with only an OK button), then you can use MsgBox as if you were calling a Sub. You can use the following syntax:

Msgbox arguments

-or-

Call MsgBox(arguments)

Examples:

The statement

MsgBox "Hello there!" causes the following box to be displayed:



This is the simplest use of MsgBox: it uses only the required prompt argument. Since the buttons argument was omitted, the default (OK button with no icons) was used; and since the title argument was omitted, the default title (the project name) was displayed in the title bar.

The statement

MsgBox "The Last Name field must not be blank.", _
vbExclamation, _
"Last Name"

causes the following box to be displayed:



This is how a data entry error might be displayed. Note that vbExclamation was specified for the buttons argument to specify what icon should be displayed – the fact that we did not add a value from the first group still causes only the OK button to be displayed. If you wanted to explicitly indicate that only the OK button should be displayed along with the exclamation icon, you could have coded the second argument as

```
vbExclamation + vbOKOnly  
making the full statement read:  
MsgBox "The Last Name field must not be blank.", _  
vbExclamation + vbOKOnly, _  
"Last Name"
```

Remember, for the buttons argument, you can add one value from each of the four groups.

An alternative (not recommended) is to use the hard-coded number for the buttons argument, as in:

```
MsgBox "The Last Name field must not be blank.", 48, "Last Name"
```

Note: also that this example provided a value for the title argument ("Last Name"), which causes that text to be displayed in the box's title bar.

The format of the MsgBox statement used in this example could also be used for more critical errors (such as a database problem) by using the vbCritical icon. You may also want to use the name of the Sub or Function in which the error occurred for your title argument.

Example:

```
MsgBox "A bad database error has occurred.", _  
vbCritical, _  
"UpdateCustomerTable"
```

Result:



Note: If you DO need to test which button the user clicked (i.e., you displayed a message box with more than one button), then you must use MsgBox as a function, using the following syntax:

```
IntegerVariable = MsgBox (arguments)
```

One of the more common uses of MsgBox is to ask a Yes/No question of the user and perform processing based on their response, as in the following example:

```
Dim intResponse As Integer  
intResponse = MsgBox("Are you sure you want to quit?", _
```



```

vbYesNo + vbQuestion, _
"Quit")
If intResponse = vbYes Then
End
End If

```

The following message box would be displayed:



After the user clicks a button, you would test the return variable (intResponse) for a value of vbYes or vbNo (6 or 7).

Note that the use of the built-in constants makes the code more readable. The statement

```

intResponse = MsgBox("Are you sure you want to quit?", _
vbYesNo + vbQuestion, _
"Quit")

```

is more readable than

```

intResponse = MsgBox("Are you sure you want to quit?", 36, "Quit")

```

and

```

If intResponse = vbYes Then
is more readable than
If intResponse = 6 Then

```

In that you can use a function anywhere a variable can be used, you could use the MsgBox function directly in an if statement without using a separate variable to hold the result ("intResponse" in this case). For example, the above example could have been coded as:

```

If MsgBox("Are you sure you want to quit?", _
vbYesNo + vbQuestion, _
"Quit") = vbYes Then
End
End If

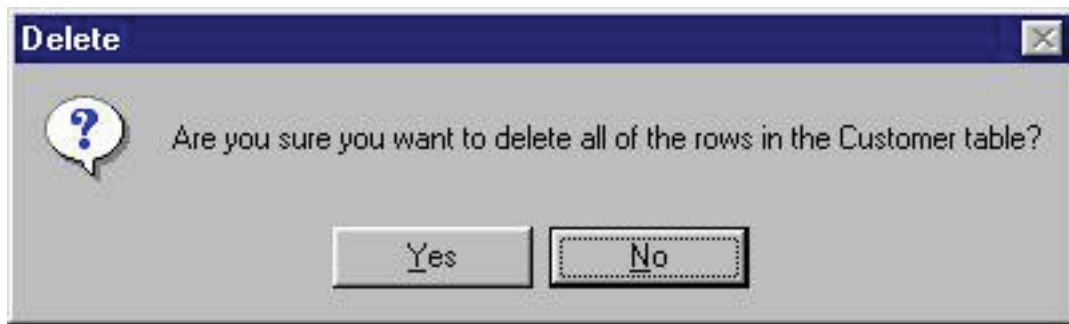
```

Note: If desired you could place the code for this example in the cmdExit_Click event of any of the "Try It" projects.

Following is an example using the vbDefaultButton2 constant:

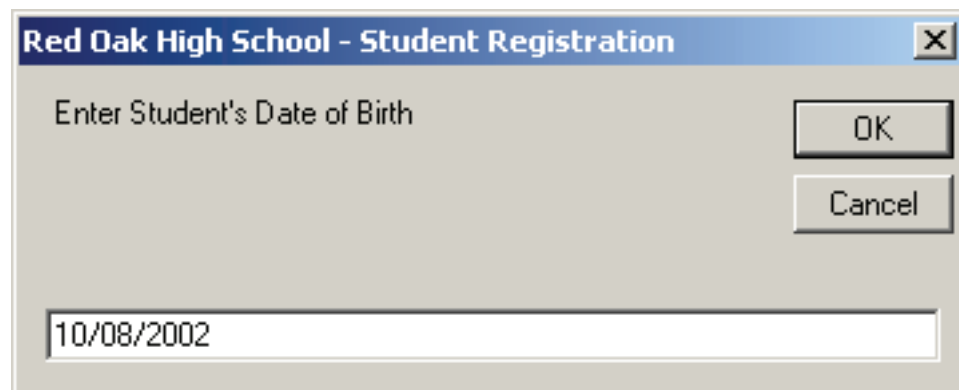
```
Dim intResponse As Integer  
intResponse = MsgBox("Are you sure you want to delete all of the rows " _  
& "in the Customer table?", _  
vbYesNo + vbQuestion + vbDefaultButton2, _  
"Delete")  
If intResponse = vbYes Then  
' delete the rows ...  
End If
```

The message box displayed by this example would look like this:



Input Box

An input box is a specially designed dialog box that allows the programmer to request a value from the user and use that value as necessary. An input box displays a title, a message to indicate the requested value, a text box for the user, and two buttons: OK and Cancel. Here is an example:



When an input box displays, it presents a request to the user who can then provide a value. After using the input box, the user can change his or her mind and press Esc or click Cancel. If the user provided a value and want to acknowledge it, he or she can click OK or press Enter. This would transfer the contents of the text box to the application that displayed the input box.

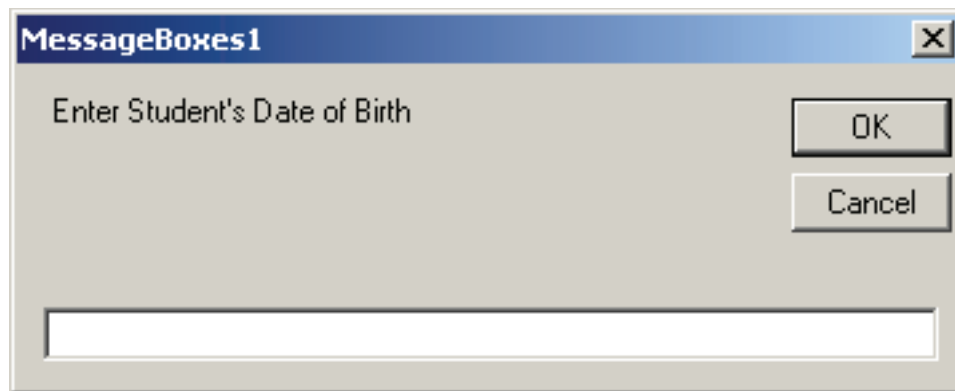
To support input boxes, the Visual Basic library provides a function named `InputBox`. Its syntax is:

```
Public Function InputBox( _  
    ByVal Prompt As String, _  
    Optional ByVal Title As String = "", _  
    Optional ByVal DefaultResponse As String = "", _  
    Optional ByVal Xpos As Integer = -1, _  
    Optional ByVal YPos As Integer = -1 _  
) As String
```

The only required argument of this function is the message that prompts the user. Here is an example:

```
Private Sub btnMessage_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnMessage.Click  
    InputBox("Enter Student's Date of Birth")  
End Sub
```

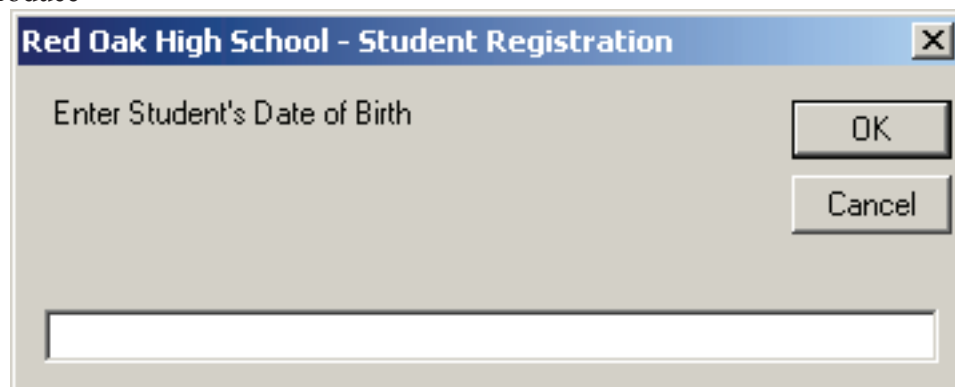
This would produce



When calling the `InputBox()` function, if you pass only the first argument, the input box would display the name of the application in the title bar. If you want, you can specify your own caption through the `Title` argument. Here is an example:

```
Private Sub btnMessage_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnMessage.Click  
    InputBox("Enter Student's Date of Birth", _  
        "Red Oak High School - Student Registration")  
End Sub
```

This would produce

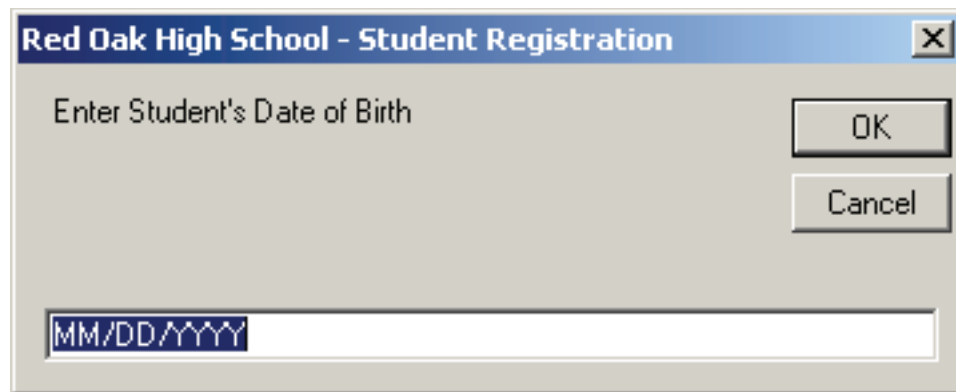


When reading the message on the Input box, the user is asked to enter a piece of information. The type of information the user is supposed to provide depends on you, the programmer. Therefore, there are two important things you should always do. First you should let the user know the type of information requested. Is it a number (what type of number)? Is it a string (such as the name of a country or a customer's name)? Is it the location of a file (such as C:\Program Files\Homework)? Are you expecting a Yes/No True/False type of answer (if so how should the user provide it)? Is it a date (if it is a date, what format is the user supposed to enter)? These questions indicate that you should state a clear request to the user.

To assist the user with the type of value you are expecting, you can give an example or a type. To support this, the `InputBox()` function is equipped with the third argument as a string. When passing it, you can provide a sample value that the user would follow. Here is an example:

```
Private Sub btnMessage_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnMessage.Click  
    InputBox("Enter Student's Date of Birth", _  
        "Red Oak High School - Student Registration", _  
        "MM/DD/YYYY")  
End Sub
```

This would produce:



The last two arguments, `XPos` and `YPos`, allow you to specify the default position of the input box when it comes up the first time.

After typing a value, the user would click one of the buttons: `OK` or `Cancel`. If the user clicks `OK`, you can retrieve the value the user would have typed. It is also your responsibility to find out whether the user typed a valid value or not. Because the `InputBox()` function returns a string, it has no mechanism of validating the user's entry. Therefore, if necessary, you must convert the return value of the input box when the user clicks `OK`. Here is an example:

```
Private Sub btnMessage_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnMessage.Click  
    Dim strDOB As String  
  
    strDOB = InputBox("Enter Student's Date of Birth", _  
        "Red Oak High School - Student Registration", _  
        "MM/DD/YYYY")
```

```

If IsDate(strDOB) Then
    MsgBox("The student was born on " & CDate(strDOB).ToString("D"), _
        MsgBoxStyle.OkOnly Or MsgBoxStyle.Information, _
        "Red Oak High School - Student Registration")
Else
    MsgBox("You provided an invalid value", _
        MsgBoxStyle.OkOnly Or MsgBoxStyle.Information, _
        "Red Oak High School - Student Registration")
End If
End Sub

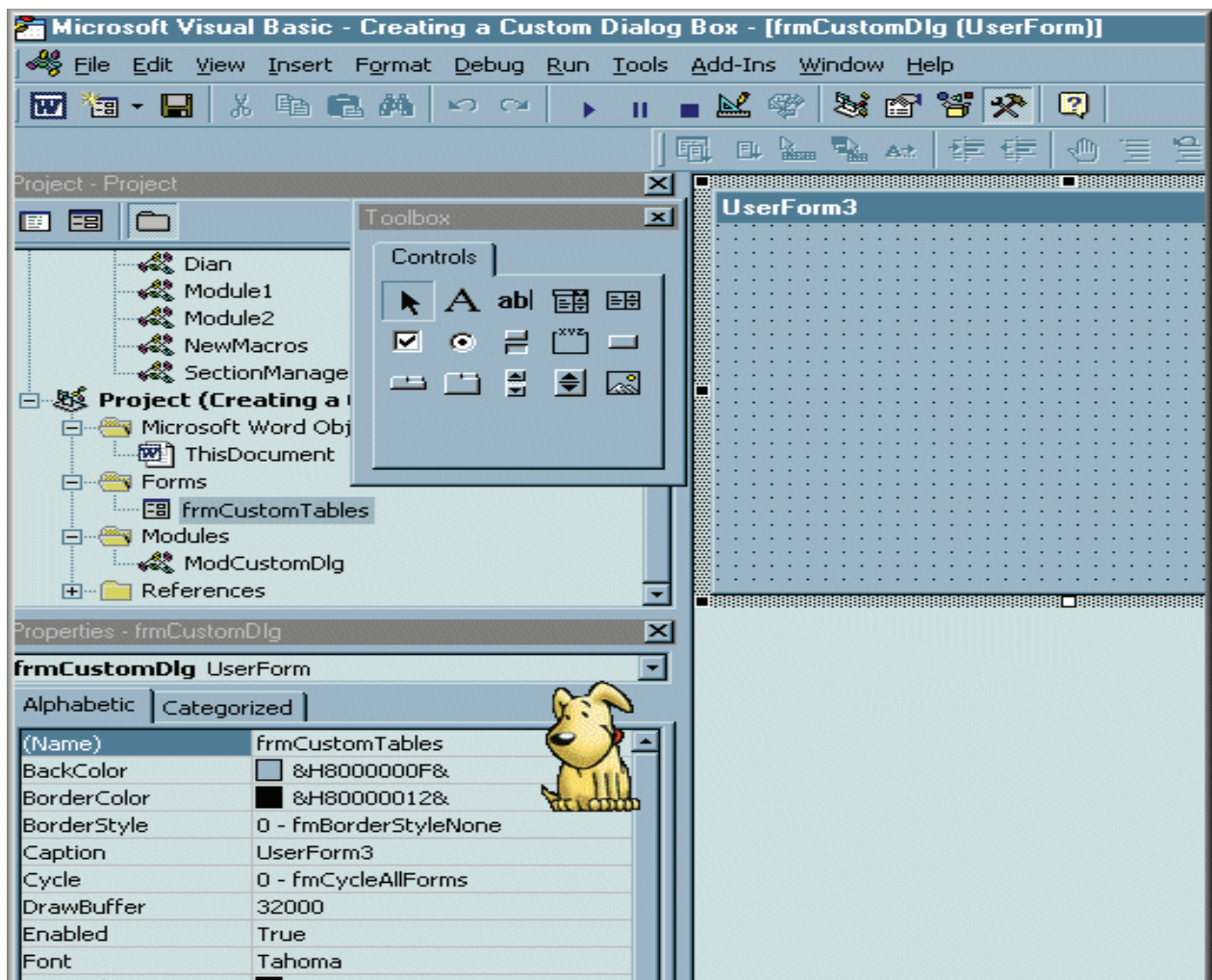
```

7.2.2 Custom Dialog box

While common dialog boxes are useful, and should be used when possible, they do not support the requirements of domain-specific dialog boxes. In these cases, you need to create your own dialog boxes. As we'll see, a dialog box is a window with special behaviors. Window implements those behaviors and, consequently, you use Window to create custom modal and modeless dialog boxes.

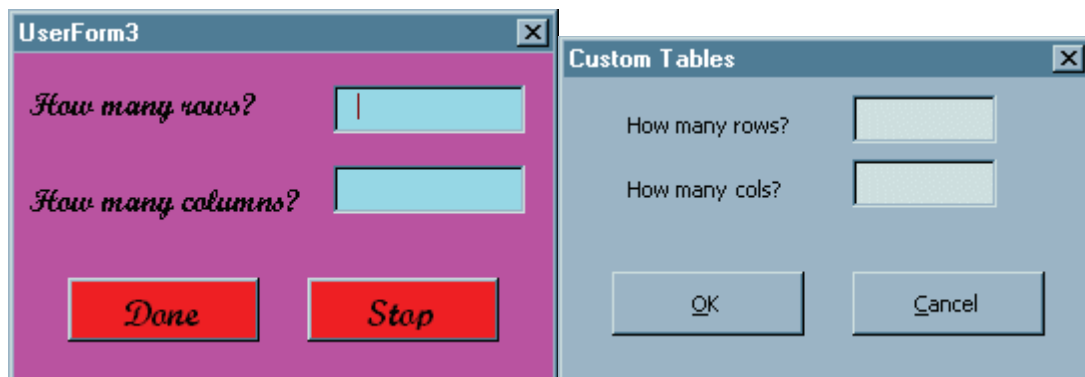
- Building a Custom Dialog Box (User Form)

Click Insert/UserForm to insert a blank user form. This is what you'll build into a dialog box. When you click on the user form, the Toolbox will appear. The toolbox contains ActiveX controls that you'll insert into the form to build your dialog box. These controls are the input boxes, drop downs, command buttons, check boxes, and so forth, that you'd find on a dialog box.



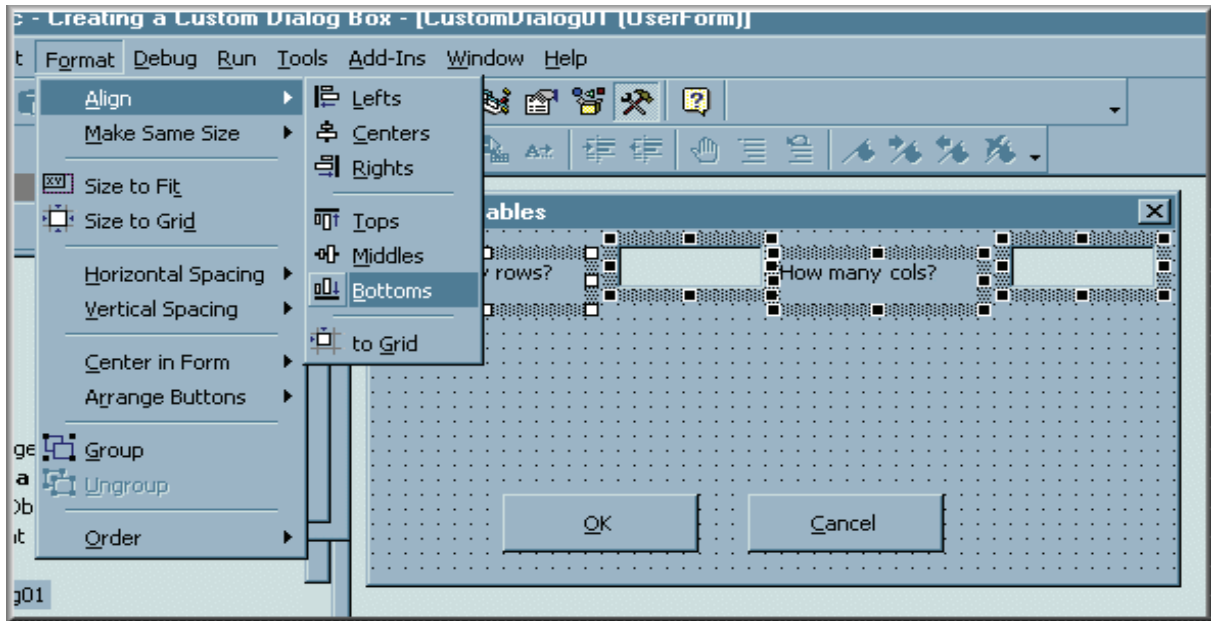
Notice that Rocky is now checking out the form's Properties. The properties are the items you can customize to make your form look the way you want. Each control has its own properties. When you click on a control, the property box changes to display the properties for that control. There are a few items in the property box for this user form on which you need to concentrate. You may have noticed that the form was originally called CustomDialog01. But there are certain naming conventions that you should learn to follow. Such as txt for TextBox, ck for CheckBox, or opt for OptionButton. You want to learn to code with consistency so that two years from now you'll be able to read the code you wrote. By using the prefix frm to help identify my form, I'll know that any code referencing this name is related to a user form (dialog box). It may sound simplistic now, but after you've written a few thousand lines of code, you'll learn to appreciate this type of consistency! Note that I've clicked in the Name property box and now renamed the user form to frmCustomTables.

- Note, too, that the user form itself still has a caption entitled UserForm3. Obviously, we'll want to rename the dialog box to something that better represents its purpose. So I'll click in the Caption property and rename this dialog box to display Custom Tables, since this is what the information captured by this dialog box will be used to create. Although there are several other property items you can customize here, and I do encourage you to experiment with each property, remember that the user interface...the look of the dialog box...will be most efficient if you keep the look consistent to what users are used to seeing. So don't go nuts and create a custom dialog box that looks like it just escaped from a circus! Look at the two dialog boxes displayed below. Which would you prefer to use?

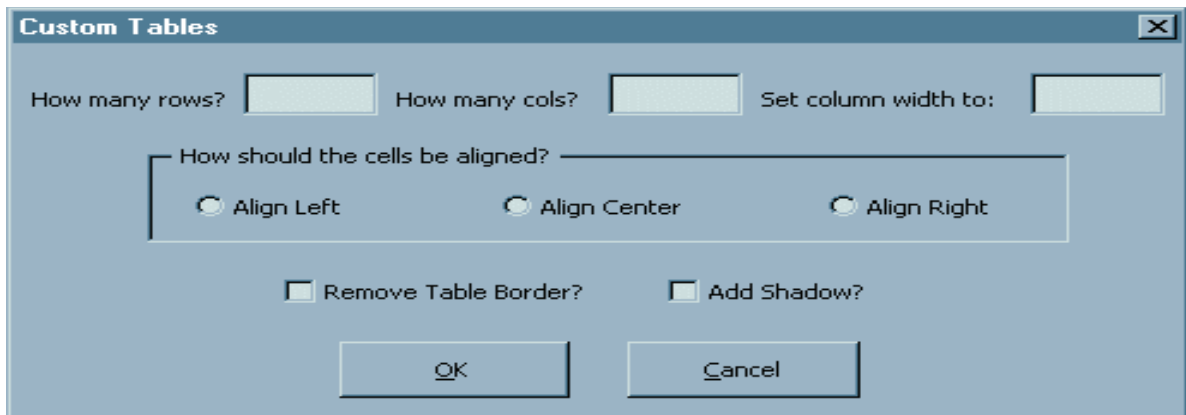


- Adding Controls: We need to add controls to our dialog box so we can ask the necessary questions and acquire the necessary information. We'll add text (labels) and input boxes (text boxes) so the user will know where to type in information, as well as some command buttons to activate the form. And just to demonstrate other controls, I'll also add option buttons and check boxes.
- To add controls to your form, you just click on the control you want and click on the form to add it. A default sized control will be added to your form. From there, you can click on the control and use the handles around the control (the little black squares) to click and drag the control to adjust the size. If you need to add multiple controls of the same type, you can double click the control in the toolbox. This will keep the item selected so you can click multiple times on the form to add several of the same control. This comes in handy when you need to add a bunch of similar controls, such as options buttons.
- Similar to a graphics program, there are also several design tools that you can use to adjust the layout of your controls. I've decided to reorganize my form (above) so that there's more room for more controls. To make sure the controls line up properly, I click on one item and then hold down

the Ctrl button to click and capture more controls. I then click Format/Align/Bottoms to have all the controls adjust and line up.



- Notice that there are many other tools that you can use to help create a professional looking dialog box
- " such as spacing, sizing and centering options. You can also select several items and click Format/Group so you can more easily work with many items as if it were one item. This can make moving or centering items on a form easier. You can also add a Frame to bring attention to certain items as a group. Experiment with the frame's Special Effects property to see its different looks.



7.2.3 Common Dialog Control

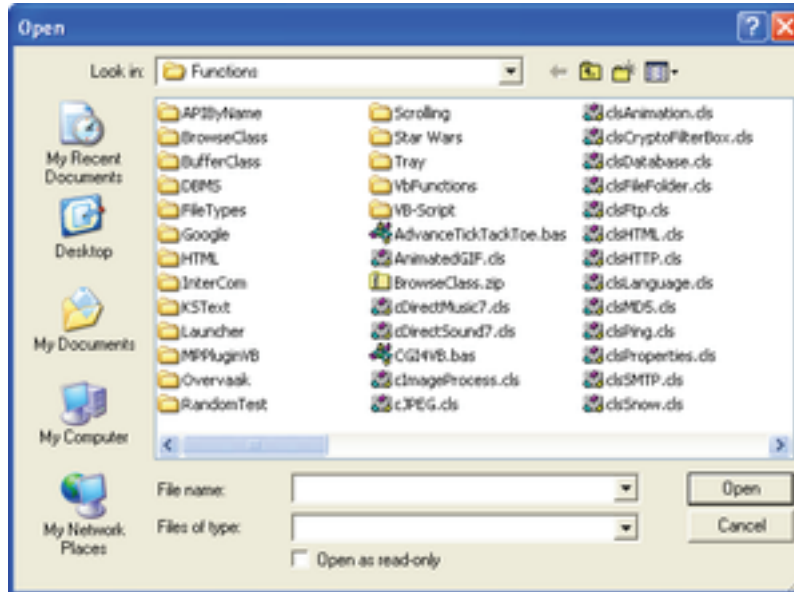
The common dialog box in visual basic is an insertable control that allows users to display a number of common dialog boxes in their program. These include Open and Save As file dialog boxes; the Find and Replace editing dialog boxes; the Print, Print Setup, Print Property Sheet, and Page Setup printing dialog boxes; and the Color and Font dialog boxes.

The easiest way to access these controls is to use a component called Microsoft Common Dialog Control 6.0. To add this component to your project, choose Project - Components, and mark it in the controls list. Subsequently, a new control should appear in your control toolbox. Choose it, and place the control onto your form (note it's only visible in design time).

Open and Save dialogs

ShowOpen shows a dialog that lets the user choose a drive, directory, file name and file extension to (presumably) open a file.

The save dialog is identical in appearance and function, with the exception of the caption. At run time, when the user chooses a file and closes the dialog box, the FileName property is used to get the selected file name.



Before displaying the dialog

Before you can use these dialogs, you must first set the Filter property to allow the user to select a file extension (set default extension using DefaultExt). The filter property is a string with the following structure:

description1|filter1|description2|filter2|

Here, the description is the string that shows up in the list box of all the available filters to choose from. The filter is a file glob expression (otherwise known as a wild card expression) that does the actually filtering, for example, "*.txt", which disallows any file extensions but TXT.

Code example

On Error Resume Next

' Clear errors

Err.Clear

' Use this common dialog control throughout the procedure

With CommonDialog1

' Raises an error when the user press cancel

.CancelError = True

' Set the file extensions to allow

CommonDialog1.Filter = "All Files (*.*)|*.txt|TextFiles (*.txt)|*.bat|Batch Files (*.bat)|*.bat"

```

'Display the open dialog box.
.ShowOpen

' Ignore this if the user has canceled the dialog
If Err <> cdlCancel Then

    ' Open the file here using FileName
    MsgBox "You've opened '" & .FileName & ""

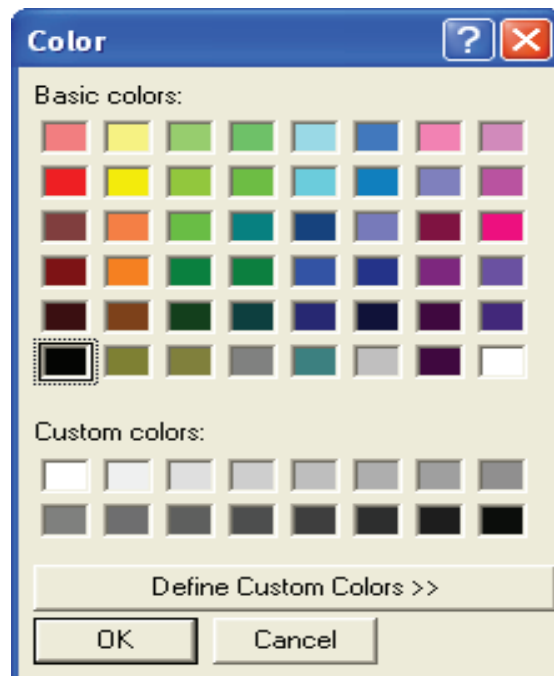
End If

```

End With

Color dialog

The color dialog allows the user to select a color from both a palette as well as through manual choosing using RGB or HSL values. You retrieve the selected color using the Color property.



Code example

```

'Don't raise errors normally
On Error Resume Next

' Clear errors
Err.Clear

' Use this common dialog control throughout the procedure
With CommonDialog1

    ' Raises an error when the user press cancel
    .CancelError = True

    ' Set the Flags property.

```



```
.Flags = cdlCCRGBInit
```

```
' Display the Color dialog box.
```

```
.ShowColor
```

```
' Ignore this if the user has canceled the dialog
```

```
If Err <> cdlCancel Then
```

```
' Set the form's background color to the selected color.
```

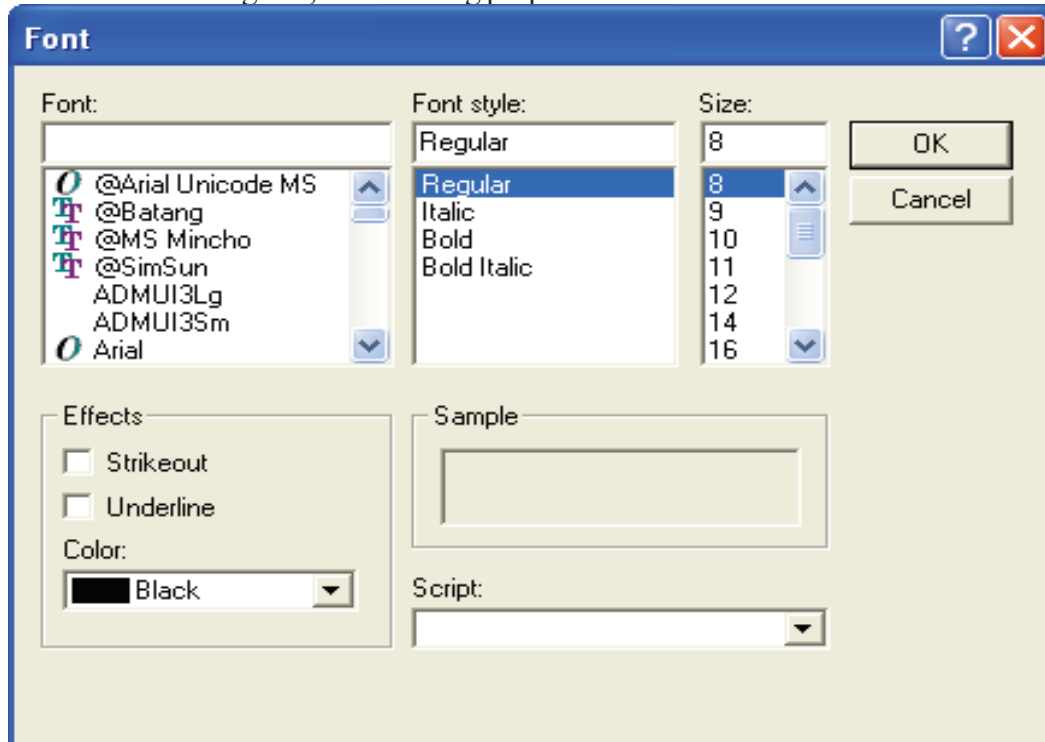
```
Me.BackColor = .Color
```

```
End If
```

```
End With
```

Font dialog

The Font dialog box allows the user to select a font by its size, color, and style. Once the user makes selections in the Font dialog box, the following properties contain information about the user's selections.



Properties

Property:	Determines:
Color	The selected color. To use this property, you must first set the Flags property to cdlCfEffects.
FontBold	Returns whether the bold checkbox was selected.
FontItalic	Returns whether the italic checkbox was selected.
FontStrikethru	Returns whether the strikethrough checkbox was selected.
FontUnderline	Returns whether the underline checkbox was selected.
FontName	Returns the selected font name.
FontSize	Returns the selected font size.

Before displaying the dialog

To display the dialog, you must first set the Flags property either cdlCFScreenFonts or cdlCFPrinterFonts (or cdlCFBoth, if you intend to let the user choose between both screen fonts and printer fonts).

Code example

```
' *** This example require a textbox called "txtText" to execute properly. ***
```

```
On Error Resume Next
```

```
' Clear errors
```

```
Err.Clear
```

```
' Use this common dialog control throughout the procedure
```

```
With CommonDialog1
```

```
    ' Raises an error when the user press cancel
```

```
    .CancelError = True
```

```
    ' Show printer and screen fonts, as well as a font color chooser.
```

```
    .Flags = cdlCFBoth Or cdlCFEffects
```

```
    ' Display the font dialog box.
```

```
    .ShowFont
```

```
End With
```

```
' Ignore this if the user has canceled the dialog
```

```
If Err <> cdlCancel Then
```

```
    ' Set the textbox's font properties according to the users selections.
```

```
    With txtText
```

```
        .Font.Name = CommonDialog1.FontName
```

```
        .Font.Size = CommonDialog1.FontSize
```

```
        .Font.Bold = CommonDialog1.FontBold
```

```
        .Font.Italic = CommonDialog1.FontItalic
```

```
        .Font.Underline = CommonDialog1.FontUnderline
```

```
        .Font.Strikethru = CommonDialog1.FontStrikethru
```

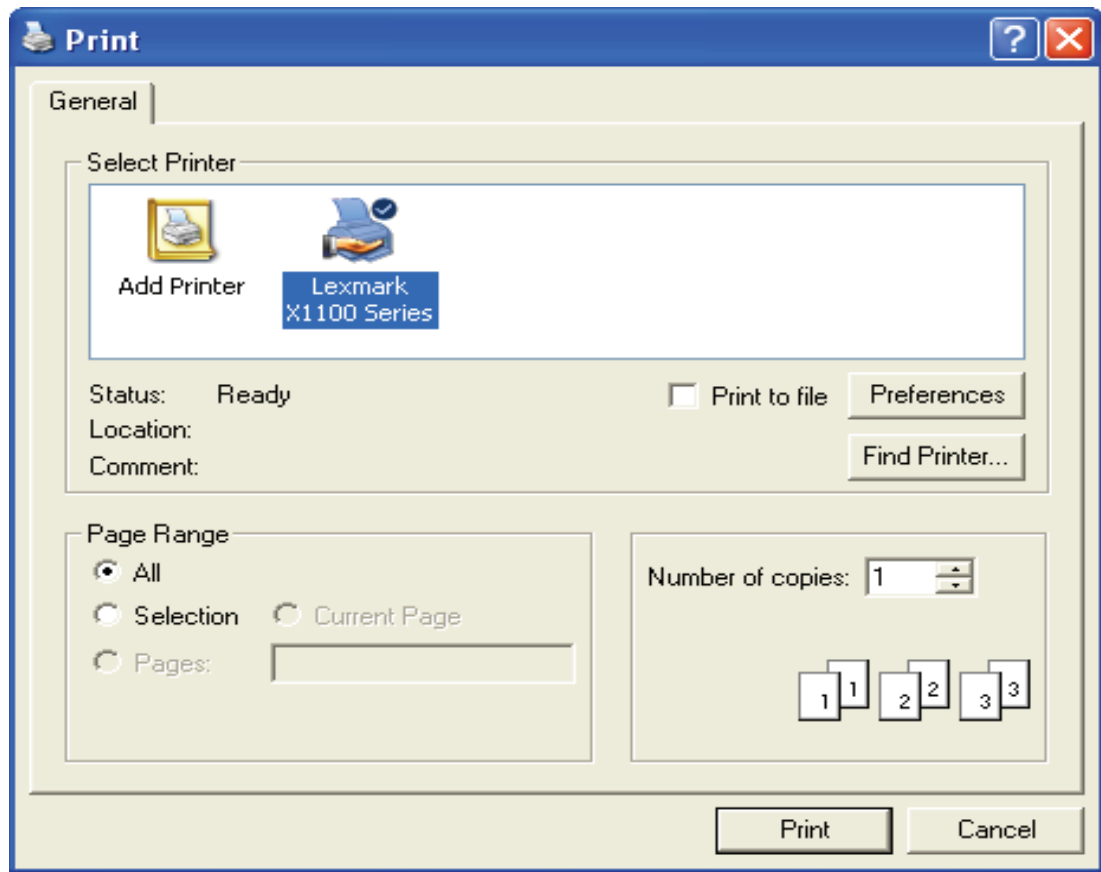
```
        .ForeColor = CommonDialog1.Color
```

```
    End With
```

```
End If
```

Print dialog

The print dialog box allows the user to select how output should be printed. Options and properties accessible to the user includes the amount of copies to make, print quality, the range of pages to print, ect. It also allows the user to choose another printer, as well as configure different settings regarding the current printer.



Properties

Property: Determines:
 Copies The amount of copies to print.
 FromPage The starting page of the print range.
 ToPage The ending page of the print range.
 hDC The device context for the selected printer.
 Orientation The page's orientation (as in portrait or landscape).

Before displaying the dialog

Before showing the dialog, feel free to set the appropriate print dialog properties to set the default values to use.

Code example

On Error Resume Next

' Variables that holds information regarding the print setup.

Dim BeginPage As Long, EndPage As Long, NumCopies As Long, Orientation As Long, TellAs Long

' Clear errors

Err.Clear

' Use this common dialog control throughout the procedure

With CommonDialog1

' Raises an error when the user press cancel

.CancelError = True

```

'Display the printer dialog box.
.ShowPrinter

' Ignore this if the user has canceled the dialog
If Err <> cdlCancel Then

    ' Retrieve the printer settings the user has chosen.
    BeginPage = CommonDialog1.FromPage
    EndPage = CommonDialog1.ToPage
    NumCopies = CommonDialog1.Copies
    Orientation = CommonDialog1.Orientation

    ' Start printing
    For i = 1 To NumCopies
        ' Put code here to send data to your printer.
    Next

End If

End With

```

7.3 Concept of Mouse Events

Visual Basic responds to various mouse events, which are recognized by most of the controls. The main events are MouseDown, MouseUp and MouseMove. MouseDown occurs when the user presses any mouse button and MouseUp occurs when the user releases any mouse button. These events use the arguments button, Shift, X, Y and they contain information about the mouse's condition when the button is clicked.

The first argument is an integer called Button. The value of the argument indicates whether the left, right or middle mouse button was clicked. The second argument is an integer called shift. The value of this argument indicates whether the mouse button was clicked simultaneously with the Shift key, Ctrl key or Alt key. The third and fourth arguments X and Y are the coordinates of the mouse location at the time the mouse button was clicked. As the Form_MouseDown() is executed automatically whenever the mouse button is clicked inside the Form's area the X, Y co-ordinates are referenced to the form.

Positioning a control

MouseDown is the commonly used event and it is combined with the move method to move an Image control to different locations in a Form. The following application illustrates the movement of objects responding to move events. It makes use of two OptionButton Controls, two image controls and a CommandButton. The application is designed in such a way that when an OptionButton is selected, the corresponding image control is placed anywhere in the form whenever it is clicked.

Open a new standard EXE project and save the Form as Move.frm and save the project as Move.vbp. Design the Form as shown below.

Object	Property	Setting
Form	Caption	MouseDown
	Name	frmMouseDown
OptionButton	Caption	Credit card is selected
	Name	optCredit
	Value	True
OptionButton	Caption	Cash is selected
	Name	optCash
Image	Name	imgCredit
	Picture	c:/credit.jpg
Image	Name	imgCash
	Picture	c:/cash.jpg

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

Option Explicit

The following code is entered in the Form_MouseDown() event

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

```
    If optCredit = True Then
```

```
        imgCredit.Move X, Y
```

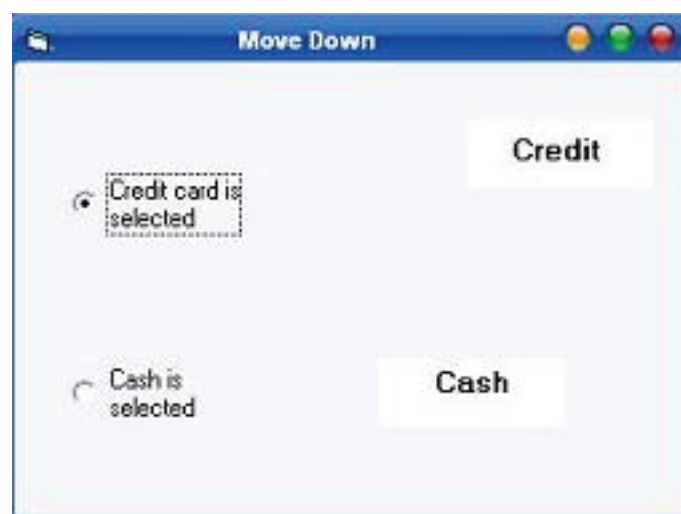
```
    Else
```

```
        imgCash.Move X, Y
```

```
    End If
```

```
End Sub
```

Run the application by keying in F5. You can notice that when the mouse is clicked on the form somewhere, the selected image moves to that clicked location. This is shown in the below figure.



Mouse events

MouseDown and MouseUp event

The MouseDown event fires when the user presses a mouse button over a control or form. Similarly the MouseUp event occurs when the user releases the mouse button over a control or form. Note that if the user moves the mouse between the time the button was pressed and released, the same control (i.e., the control that originally received the MouseDown) will receive the MouseUp event.

During the MouseDown and MouseUp event procedures, you might want to know whether the left or right mouse button was pressed. You might also want to know whether or not one of the auxiliary keys (Shift, Alt, or Ctrl) also was depressed during the mouse event.

Finally, it might be nice to know the relative position of the mouse pointer within the form or control receiving the event.

All of the foregoing information is available in parameters that the MouseUp or MouseDown event procedure receives from the system. The four parameters are:

- **Button As Integer.** This is a value representing which mouse button fired the event. The value of this parameter is either `vbLeftButton`, `vbRightButton`, or `vbMiddleButton`. Again these terms are from the point of view of a right-handed mouse. `vbLeftButton` always refers to the primary button, regardless of whether it's physically the left or right button.
- **Shift As Integer.** This parameter represents an integer that indicates whether an auxiliary key is pressed during the Mouse event. It contains a value of 0 (none), 1 (Shift), 2 (Ctrl), 4 (Alt), or the sum of any combination of those keys. For example, if both the Ctrl and Alt key were pressed, the value of the Shift parameter is 6. You can check for the state of any one of the auxiliary keys with one of the VB constants `vbAltMask`, `vbCtrlMask`, or `vbShiftMask`. The following code illustrates how you could store the state of each auxiliary key in a Boolean variable within the MouseDown or MouseUp event procedure. The bit-wise representation of 1, 2, or 4 in the Shift parameter is 000000001, 000000010, 00000100. By doing a logical AND between the Shift parameter and one of the VB Shift-key constants, you can pick out whether each of the three Shift keys is currently pressed, as illustrated in Listing 3.1.
- **X As Single.** This is the horizontal position of the mouse pointer from the internal left edge of the control or form receiving the event.
- **Y As Single.** This is the vertical position of the mouse pointer from the internal top edge of the control or form receiving the event.

TESTING THE SHIFT MASK IN A MOUSEDOWN OR MOUSEUP EVENT PROCEDURE

```
Dim blnIsAlt As Boolean
Dim blnIsCtrl As Boolean
Dim blnIsShift As Boolean
blnIsAlt = Shift And vbAltMask
blnIsCtrl = Shift And vbCtrlMask
blnIsShift = Shift And vbShiftMask
```

The mouse events can be combined with graphics methods and any number of customized drawing or paint applications can be created. The following application combines MouseMove and MouseDown events, and illustrates a drawing program.

Open a new Standard EXE project and save the Form as Draw.frm and save the Project as Draw.vbp. Name the caption of the as Drawing. Add command button control and name the caption of it as Clear

Enter the following code in the Form_MouseDown () procedure, Form_MouseMove () procedure and cmdClear_Click () procedures respectively.

```
Private Sub cmdClear_Click()  
frmDraw.Cls  
End Sub  
Private Sub Form_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)  
frmDraw.CurrentX = X  
frmDraw.CurrentY = Y  
End Sub  
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)  
If Button = 1 Then  
Line (frmDraw.CurrentX, frmDraw.CurrentY)-(X, Y)  
End If  
End Sub
```

Button value 1 indicates that the left mouse button is clicked. The code written in the MouseDown event changes the CurrentX and CurrentY to the coordinates where the mouse button was just clicked.

Run the application. You can notice that when the mouse is clicked and moved in the Form a line is drawn corresponding to the mouse movement. Following figure illustrates the combined action of MouseDown and MouseMove.



The program uses two graphics related Visual Basic concepts, the Line method and the CurrentX and CurrentY properties. Line method is preferred to draw a line in a Form. The following statement draws a line from the coordinates X = 2500, Y = 2000, X = 5000, Y = 5500

```
Line (2500, 2000) - (5000, 5500)
```

The CurrentX and CurrentY properties are not visible in the properties window of the Form because it cannot be set at the design time. After using the Line method to draw a line in a Form, Visual Basic automatically assigns the coordinate of the line's end point to the CurrentX and CurrentY properties of the Form on which the line is drawn.

MouseMove Event

Visual Basic does not generate a MouseMove event for every pixel the mouse moves over and a limited number of mouse messages are generated per second by the operating environment. The following application illustrates how often the `Form_MouseMove()` event is executed.

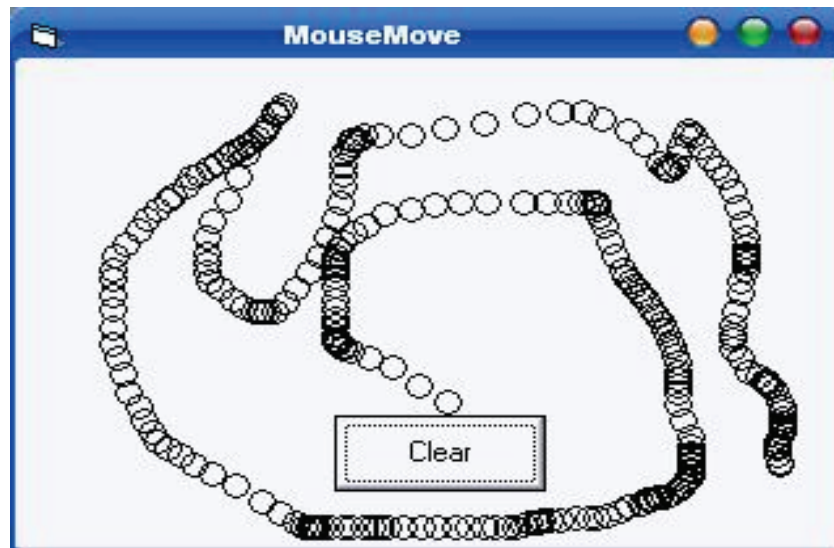
Open a new standard EXE project and save the form as `MouseMove.frm` and save the Project as `MouseMove.vbp`. Place a `CommandButton` control and name the caption as `Clear` and set the name as `cmdClear`.

The following code is entered in the `cmdClear_Click()` and `Form_MouseMove()` events respectively.

```
Private Sub cmdClear_Click()  
    frmMouseMove.Cls  
End Sub  
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)  
    Circle (X, Y), 70  
End Sub
```

The above procedure simply draws small circles at the mouse's current location using the `Circle` method. The parameter `x, y` represent the centre of the circle, and the second parameter represents the radius of the circle.

Save the application and run. You can notice that when the mouse is moved inside the Form, circles are drawn along the path of the mouse movement as shown in below figure. And also you can notice the circles are widely spaced when the mouse is moved quickly. Each small circle is an indication that the `MouseMove` event occurred and the `Form_MouseMove()` procedure was executed.



7.4 Summary

- A dialog box is a form defined with particular properties. Like a form, a dialog box is referred to as a container. Like a form, a dialog box is mostly used to host child controls, insuring the role of dialog between the user and the machine.
- Visual Basic responds to various mouse events, which are recognized by most of the controls. The main events are `MouseDown`, `MouseUp` and `MouseMove`.
- There are two types of dialog boxes: modal and modeless.

- A Modal dialog box is one that the user must first close in order to have access to any other framed window or dialog box of the same application.
- A dialog box is referred to as modeless if the user does not have to close it in order to continue using the application that owns the dialog box
- MouseDown occurs when the user presses any mouse button and MouseUp occurs when the user releases any mouse button.

7.5 Self Assessment Questions

1. What type of dialog boxes available in VB?
2. What methods are used in event handling ?
3. WAP to accept n values in an array and display the values using Input Box?
4. What is the parameters of MouseUp and Mousedown event?
5. Describe the common dialog control?
6. Differentiate b/w Modal dialog box and Modeless dialog Box?

Unit - 8 : MDI Forms and Flex grid Control

Structure of Unit:

- 8.0 Objectives
- 8.1 Introduction
- 8.2 Concept of MDI Forms
 - 8.2.1 Creating and using MDI Form.
 - 8.2.2 Arranging the Child forms.
- 8.3 Concept of Flexgrid Control
 - 8.3.1 Adjusting the size of Controls.
 - 8.3.2 Adding MS-Flexgrid Control.
 - 8.3.3 Changing the Cell Width and Height.
 - 8.3.4 Entering the values in the cells of grid.
 - 8.3.5 Scrollbars in MS-Flexgrid.
- 8.4 Summary
- 8.5 SelfAssessment Questions

8.0 Objectives

After completing this unit, you will be able to:

- Understand the concept of MDI Forms
- Understand the concept of Flexgrid Control
- Know about the arranging of child form
- Learn the Use of MS Flexgrid

8.1 Introduction

The Multiple Document Interface (MDI) was designed to simplify the exchange of information among documents, all under the same roof. With the main application, you can maintain multiple open windows, but not multiple copies of the application. Data exchange is easier when you can view and compare many documents simultaneously.

Flex Grid is a control which facilitates display of tabular data on a Form, along with the facility of Adding a Row, Deleting a Row and Sorting of records etc.

Using Microsoft's FlexGrid control (MSFLXGRD.OCX) you can create utilities to display, filter, edit, validate and update your data. For example, such utilities could include:

1. data entry & validation
2. high level reports
3. ported spreadsheet macro applications retaining cell layout & format

8.2 Concept of MDI Forms

The Multiple Document Interface (MDI) was designed to simplify the exchange of information among documents, all under the same roof. You almost certainly use Windows applications that can open multiple

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

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

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

To create an MDI application, follow these steps:

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

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



Parent and Child Menus

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

Following example illustrates the above explanation.

- * Open a new Project and name the Form as Menu.frm and save the Project as Menu.vbp
- * Design a menu that has the following structure.

◇ MDI Menu Menu caption

- MDI Open opens a new child Form
- MDI Exit terminates the application

* Then design the following menu for the child Form

◇ Child Menu Menu caption

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

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

```
Form1.Show
```

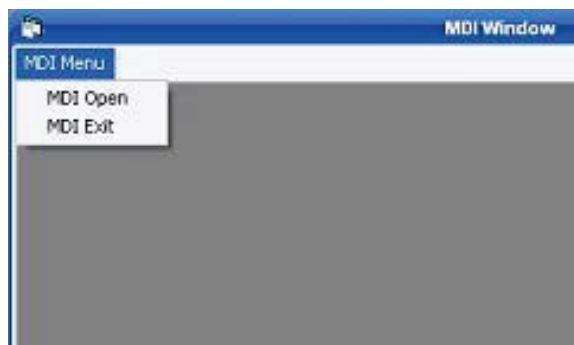
And so double click on MDI Exit and add the following code in the click event

End

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

```
Unload Me
```

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



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

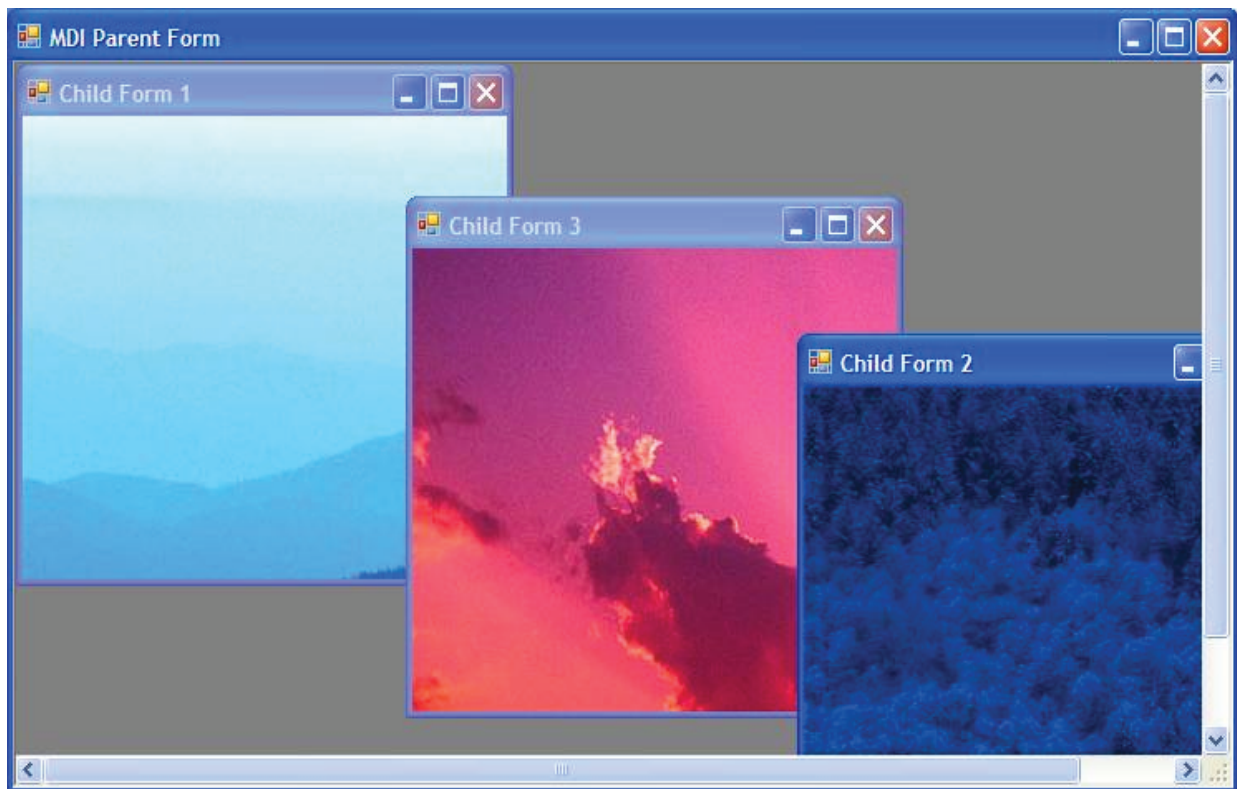
8.2.1 Creating and using MDI Form

The acronym MDI stands for Multi Document Interface. If you have ever used Microsoft Word or Excel the chances are you have used a multi-document interface. Multi-document interfaces consist of a parent form (also called a *container*) which contains other forms. For example, when more than one document is open in some versions of Microsoft Word, each document appears in its own form, contained within the parent form. The forms within the parent can be moved to any position within the parent form, but typically cannot be moved outside the confines of the parent.

The term MDI is actually misleading. A more accurate name would be *multi-form interface* or *multi-window interface*. The reason for this is that there is nothing that restricts the forms to containing just *documents*. In fact, a form in an MDI interface can contain anything that a form in a non-MDI interface can

contain. In fact, recent versions of Microsoft Office (specifically Office 2007) have moved away from the MDI approach. When multiple documents are opened in the Word 2007, for example, each document appears in an independent window. That said the

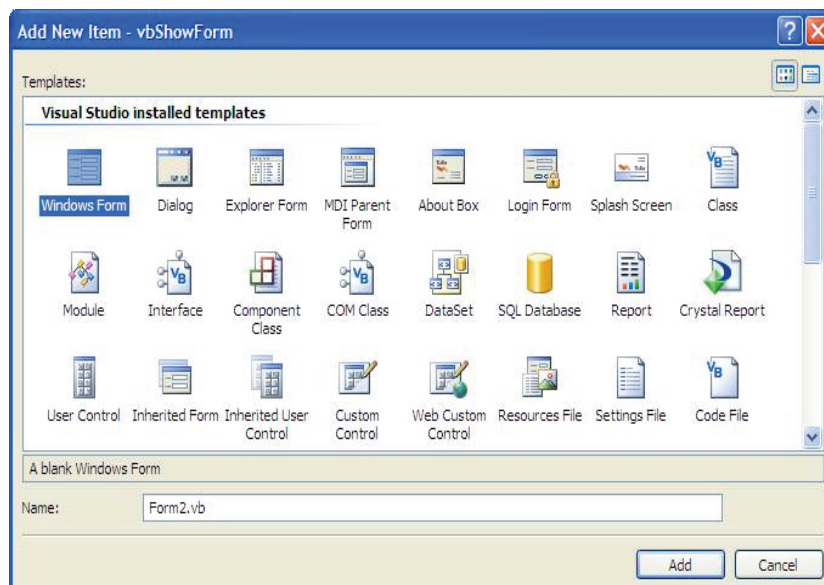
The following [figure shows](#) an application implemented using an MDI. Notice that all the forms are contained in the parent window:



Creating an MDI Parent and Child Forms in Visual Studio

The first step in creating an MDI form is to create the parent form. Begin by launching Visual Studio and creating a new *Windows Application* project named *vbMDI*. The [new project](#) will contain a [single form](#). Change the name of this form in the *Properties* panel to *MDIparent*. Also in the *Properties* panel, change the *IsMdiContainer* property to *True*. You will notice that the form background changes to dark grey, the default for MDI containers. This form is now ready to act as the container for the child forms.

Add two more forms to the project using the Visual Studio *Add New Item* toolbar button:



Edit the properties of the two new forms so that the [title texts](#) are *Child Form 1* and *Child Form2* respectively. Also, use the properties panel to name the forms *MDIchild1* and *MDIchild2*.

The next step is to add the two new forms (*MDIchild1* and *MDIchild2*) to the parent form (*MDIparent*). To do this click on the tab for the parent form in Visual Studio and double click on the form to display the event procedures. We will now write [Visual Basic code](#) to add the two child forms to the container parent form. To do this we will set the *MdiParent* property of each child to reference the *MDIparent* form. Note that because this is the *Load* event of the actual parent form, we refer to it with the keyword *Me* rather than by the form name. Having set the *MdiParent* of each child we then need to display the form using the form *Show()* method:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
```

```
    MDIchild1.MdiParent = Me
```

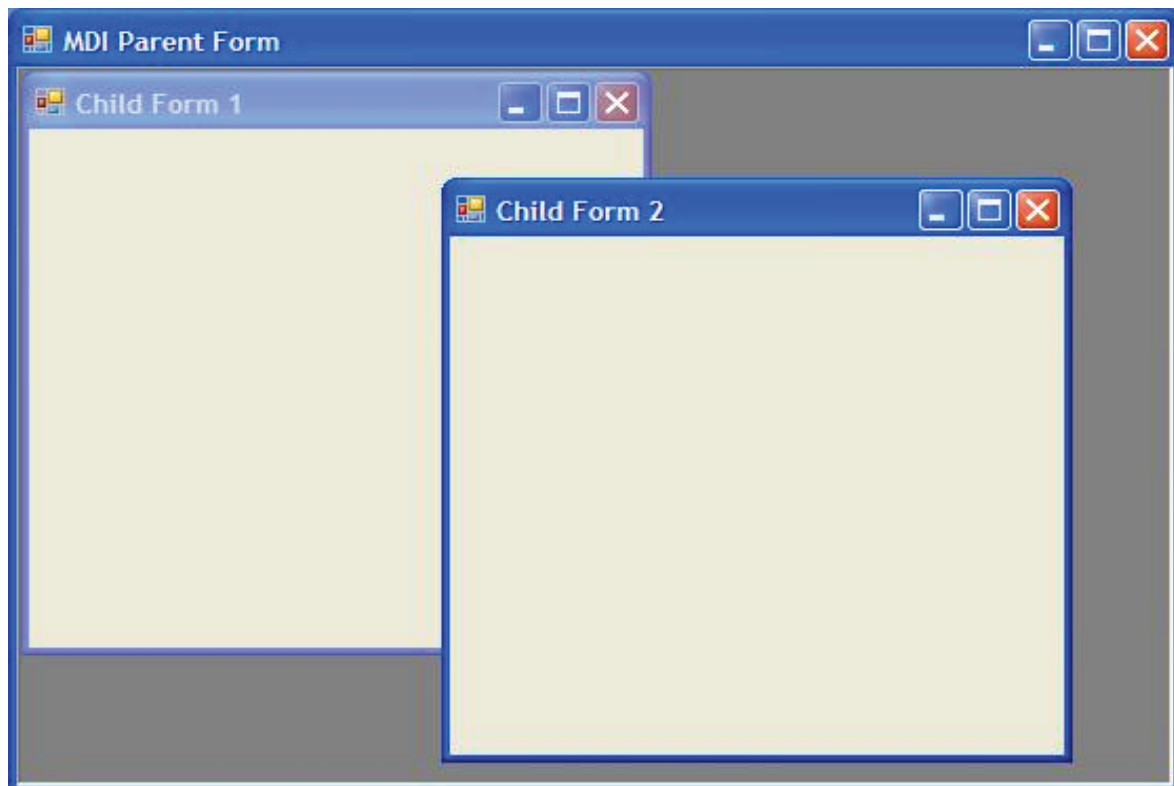
```
    MDIchild1.Show()
```

```
    MDIchild2.MdiParent = Me
```

```
    MDIchild2.Show()
```

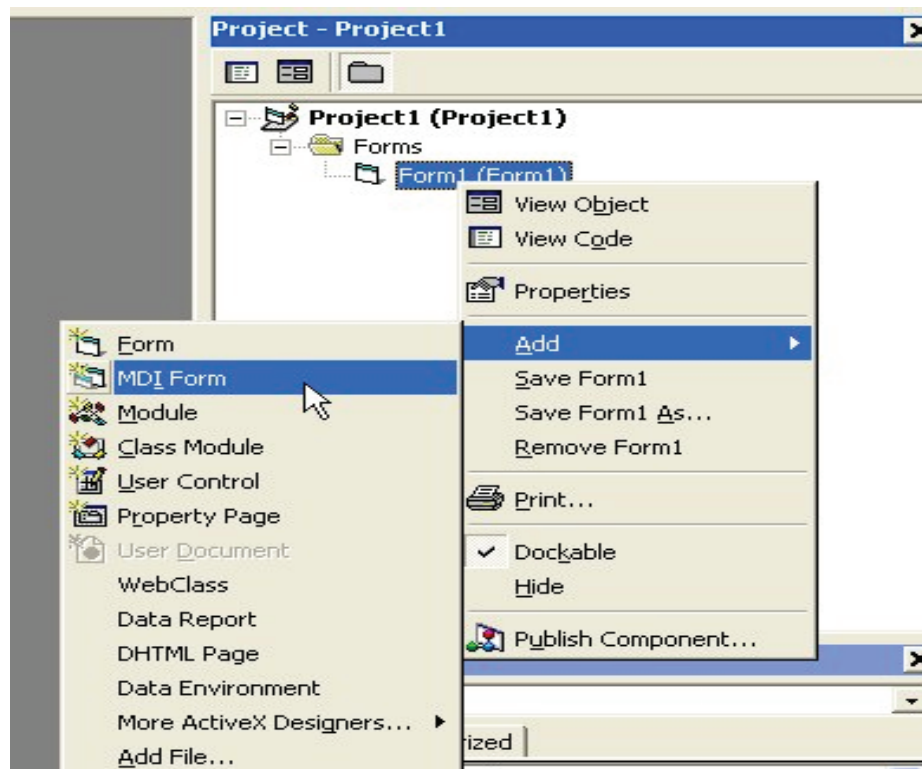
```
End Sub
```

After adding the above code to the parent form's Load event, press **F5** to build and run the example. When the application runs the child forms will both appear in the parent form:

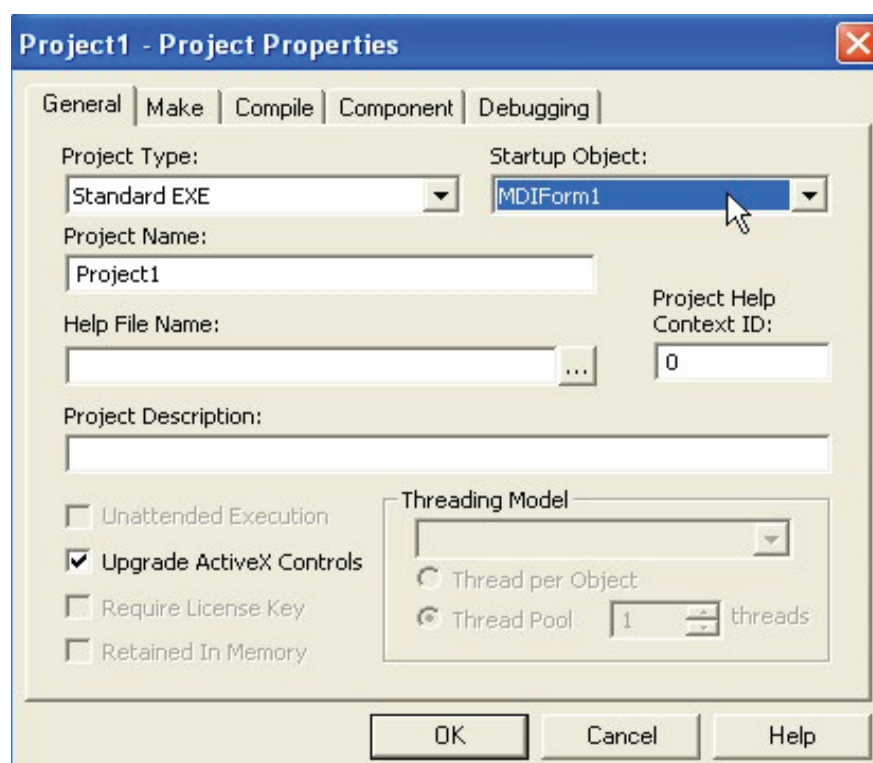


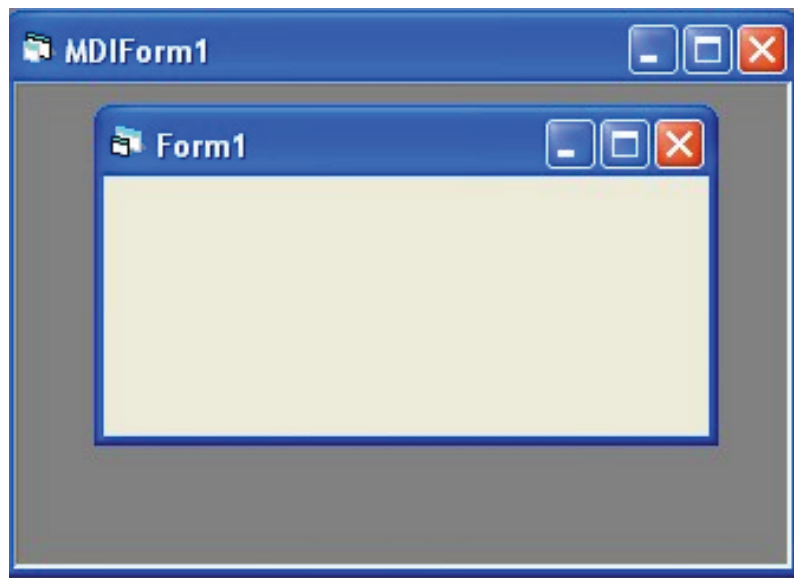
How to use MDI Form in VB

The Form that you have been using so far are single form. In Visual Basic you can use Multiple Document Interface (MDI) Form, a form that can contain multiple forms. Here is how to create MDI Form:



1. Start Visual Basic Standard Exe project.
2. In the Project Window, do Right Click and it show a pop up menu, choose **Add > MDI Form**. In the dialog window, press Open button
3. Go to Form1. Change the **MDI Child Property** of Form1 (and all other forms if exist) into True. This will make Form1 as child form instead of parent
4. In the VB menu, select **Project > Project1 Properties**. In the **General** Tab, StartUp Object, select MDI Form1, then OK. This is to make VB run for the first time by calling the MDI Form.
5. Run the program





8.2.2 Arranging the Child forms.

Use the MDI form's Arrange method passing it a parameter to indicate how you want the child forms arranged.

```
Private Sub mnuWindowArrangeIcons_Click()
    Me.Arrange vbArrangeIcons
End Sub
```

```
Private Sub mnuWindowCascade_Click()
    Me.Arrange vbCascade
End Sub
```

```
Private Sub mnuWindowTileHorizontal_Click()
    Me.Arrange vbTileHorizontal
End Sub
```

```
Private Sub mnuWindowTileVertical_Click()
    Me.Arrange vbTileVertical
End Sub
```

Provide other functions for all child forms by looping through the Forms collection.

```
Private Sub mnuWindowCloseAll_Click()
    Dim frm As Form
```

```
    For Each frm In Forms
        If Not (frm Is Me) Then
            Unload frm
        End If
    Next frm
End Sub
```

```
Private Sub mnuWindowMinimizeAll_Click()
```



```
Dim frmAs Form
```

```
For Each frm In Forms  
    If Not (frm Is Me) Then  
        frm.WindowState = vbMinimized  
    End If  
Next frm  
End Sub
```

```
Private Sub mnuWindowRestoreAll_Click()  
Dim frmAs Form
```

```
For Each frm In Forms  
    If Not (frm Is Me) Then  
        frm.WindowState = vbNormal  
    End If  
Next frm  
End Sub
```

```
Private Sub mnuWindowNewWindow_Click()  
Dim frmAs New Form1
```

```
    frm.Show  
End Sub
```

8.3 Concept of Flexgrid Control

The Grid Control

The grid control lets you display information in a series of rows and columns, including special rows and columns that display row and column headings. The most common example of a program which deals with this type of information display is a spreadsheet.

Visual Basic has two separate controls for working with rows and columns of information:

Dbgrid control for displaying and editing the contents of a database

MsflexGrid control is capable of displaying text or pictures in any of its cells. Moreover, you can use the grid to sort information in the tables and format the information for easier viewing.

The grid control displays information in cells. A cell is a location in the grid control at which a row and a column intersect.

- You can set the current cell in the code.
- The user can set the current cell by clicking a cell or by using the arrow keys.

The Microsoft FlexGrid control provides advanced features for displaying data in a grid. It is similar to the data-bound grid control; however, it does not allow the user to edit data bound to or contained in the control. This enables you to display data to the user while ensuring that the

original data is secure and unchanged. You can add cell-editing features to the Microsoft FlexGrid control by combining it with a text box.

The Microsoft FlexGrid control supports the following features:

- Read-only data binding.
- Ability to pivot data between columns and rows.
- Automatic regrouping of data during column adjustment.
- Adaptation to existing Visual Basic code for the data-bound grid (DBGrid).
- Ability of each cell to contain text, a picture, or both.
- Changing of current cell text in code or at run time.
- Ability to read data automatically when assigned to a data control.
- Word-wrap capability for text within cells.
- Find and replace functionality for complex strings.

8.3.1 Adjusting the size of Controls.

Creating and Sizing Rows and Columns

The **rows** and **cols** properties of the flexgrid determine how many total rows and columns display in the grid.

You can create two kinds of rows or columns in the grid control. The most common is non-fixed.

- A non-fixed row or column scrolls when scroll bars are active in the grid control;
- A fixed row or column does not scroll at any time.

If the number of rows and columns is greater than can fit on-screen, the FlexGrid automatically provides scrollbars. The headers are controlled by the **FixedRows** and **FixedCols** properties. The default colour of cells in a non-fixed row or column is white. Cells in a fixed row or column are gray. You create rows and columns in a grid by setting four properties of the grid control.

Property	description
Rows	Sets or returns the total number of fixed and non-fixed rows in the grid
Cols	Sets or returns the total number of fixed and non-fixed Columns in the grid
FixedRows	Sets or returns the number of fixed rows in the grid
FixedCols	Sets or returns the number of fixed columns in the grid

To control the color displayed, flexgrid control has five major sets of color properties.

Background Foreground

Color Property	Color Property	Color Use
----------------	----------------	-----------

BackColor	ForeColor	Controls the color of any standard cells in the grid. A standard cell is one that is not part of the fixed rows or columns and is not selected.
BackColorFixed	ForeColorFixed	Controls the colors of the “header” cells in the fixed rows and columns.
BackColorSel	ForeColorSel	Controls the colors in cells that are selected
BackColorBkg	N/A	Controls the color of the “empty space” in the flexgrid control that is not occupied by any cells.
CellBackColor	CellForeColor	Controls the color of an individual cells. Can only be used during runtime.

The font information for most cells in the grid is controlled by the **font** property of the grid itself. Different font properties are listed as follows.

CellFontName, CellFontSize, CellFontBold,

CellFontItalic, CellFontUnderline

To change the appearance of an individual cell, you can use the **color** and **font** properties in conjunction with the row and col properties.

For example, the following code selects the cell in row 3, column 4 , and then changes it to a Bold Red font.

```
FgMain.Row =3
```

```
FgMain.col = 4
```

```
FgMain.text = “Bold Font”
```

```
FgMain.CellFontBold = True
```

```
FgMain.CellForeColor = VbRed
```

FlexGrid control gives you control over the lines that make up the grid. These lines are controlled by the following properties.

GridColor :sets the color of the grid lines in all standard cells

GridColorFixed : sets the color of the grid lines in the fixed cells

GridLines : Sets the apperance of the grid lines for the standard cells.

0 – flexGridNone(no grid lines)

1 – flexGridFlat (Flat grid lines)

2 – flexGridInset (inset grid lines)

3 – flexGridRaised (raised grid lines)

GridLinesFixed : sets the appearance of the grid lines in the fixed cells.

GridLineWidth : sets the width of the lines use to make up the grid.

Controlling the flexgrid’s behavior

As you know properties of a control affects not only its appearance, but its behavior as well. Flexgrid has several behavior while running in your program.

AllowBigSelection: Determines whether the user can select a row or column by clicking the header(fixed cell) of the row or column,

AllowUserResizing: Determine whether the user can change the size of the rows or columns in the grid. The property has four possible settings.

0 – flexResizeNone (no resizing allowed, default)

1 – flexResizeColumns (the user can resize columns only)

2 – flexResizeRow (the user resize rows only)

3 – flexResizeBoth (the user can resize rows and columns)

FillStyle :*Determines whether setting the value of a cell property will affect only the current cell or all selected cells.*

0 – flexFillSingle. Allows only the current cell's properties to be set.

1 – flexFillRepeat. Causes setting a cell property(such as CellFontName
or Text) to set the value for all selected cells.

MergeCells : Determines whether adjacent cells with the same contents will be merged to be shown in the grid as single cell.

0 – no merging (default value)

1 – allow merging of anywhere in the grid

2 – allow merging across columns but not across rows

3 – allow merging across rows but not across columns

SelectionMode :Determines how the user can select cells in the grid.

0 – allow user to select cells individually

1 – forces the selections to cover entire rows

2 – forces the selections to cover entire columns

WordWrap: Determines whether the text in a cell will wrap to multiple lines.

Working with the MSFlexGrid Control in Code

Most of properties that you can set at design time can also be set from program code. The most common task that you will perform in working with a grid is setting or retrieving the value of a cell. In the FlexGrid, the contents of all cells are treated as text strings.

There are three properties that you can use to set or retrieve the value of a cell in the FlexGrid: **Text**, **TextArray**, and **TextMatrix**. You can use each of these properties to retrieve or set the values of a single cell.

The following code shows how you would retrieve the value of the cell in the second row and second column of the grid.

FgMain.Row = 1

Fgmain.Col = 1

TxtReturn = FgMain.Text

If a single cell is selected, text property only sets the value of the selected set. If multiple cells are selected, the text property will set the value of the current cell if **FillStyle** property is set to 0, or will set the value of all the selected cells if the **FillStyle** property is set to 1.

You can only retrieve or set the value of the current cell. This means if you want to process multiple cells, you will have to set the Row and Col properties for each cell's value you want to retrieve.

FgMain.Row = 1

FgMain.Col = 1

Fgmain.Text = "One Cell"

Fgmain.Col = 2

FgMain.text = "Another Cell"

TextArray property can be used to retrieve the value of any cell, not just the current cell but for working with multiple cells. The **textarray** property uses a single index to specify the cell to be retrieved. This index is determined by multiplying the desired row number by the number of columns in the grid and then adding the desired column number. The following code shows how to retrieve the value from the cell in the third column of the second row.

InDesiredCol = 2

InDesiredRow = 1

TxtRet = Fgmain.TextArray(InDesiredRow * Fgmain.Col + IndesiredCol)

***TextMatrix** property is used for setting and retrieving the value of a cell. It requires two arguments. The index of the row and the index of the column to be retrieved. Following is used to retrieve the value from the cell in the third column of the second row.*

Txtreturn = Fgmain.TextMatrix(1 , 2)

Adding and Deleting Rows.

There are two ways that you can change the number of rows in the flexgrid. You can change the **rows** property of the grid, or you can use **addItem** and **RemoveItem** methods. Whereas changing Rows property adds or removes items from the bottom of the grid, the flexGrid's methods let you control the insertion and deletion points.

Using **AddItem** method you can specify index value for the method to insert the row somewhere else in the grid. This method works the same way as a list box's AddItem method. The following lines of code show how the AddItem method can be used:

FgMain.AddItem "NemRoman"

FgMain.AddItem "NewRoman", 2

FgMain.AddItem "NewRoman" & Chr(9) & "NewRoman"

To specify the values for multiple cells, create a text string containing the values of all the input cells, in column order. Between each pair of values, insert the Tab character by using the **Chr(9)** function.

To **delete** a specific row from the grid, you use **removeItem** method, which requires you to specify the row number of the row to be removed. The following code shows you how to remove the second row of the grid.

FgMain.RemoveItem 1

To clear the entire grid, you use the clear method, as shown in the following line

MsFlexGrid1.Clear

Sample Application1: Sorting and Merging Data

This program demonstrates how you can sort and merge data in the Microsoft FlexGrid control.

In most cases, you will load data into the control from a database. In this example, however, sample data will be used to populate the columns and rows of the control. The steps to creating this data display are as follows:

- Set the properties of the Microsoft FlexGrid.
- Create data.
- Define routines to calculate an index and to do a sort.
- Define routine to enter the data (from step 2) into the control.
- Allow the control to switch views in terms of data organization.

Setup

The following controls are used in the scenario:

- Microsoft FlexGrid control

Set Properties of Microsoft FlexGrid Control

In this example, the following properties are used to set the number of columns and rows, the font and font size, and to create headings for the columns:

Object	Property	Setting
Microsoft control	FlexGrid	
	Name	Fg1
	Cols	4
	Rows	20
	MergeCells	2 - Restrict
	FormatString	<Region <Product <Employee >Sales FontName Arial

Create Data

Use this routine in the Form_Load event to create an array to store the sample data:

Sub Form_Load ()

Dim I As Integer

' Create array.

For i = Fg1.FixedRows To Fg1.Rows - 1

' Region.

Fg1.TextArray(fgi(i, 0)) = RandomString(0)

' Product.

Fg1.TextArray(fgi(i, 1)) = RandomString(1)

' Employee.

Fg1.TextArray(fgi(i, 2)) = RandomString(2)

**Fg1.TextArray(fgi(i, 3)) = _
Format(Rnd * 10000, "#.00")**

Next

' Set up merging.

Fg1.MergeCol(0) = True

Fg1.MergeCol(1) = True

Fg1.MergeCol(2) = True

' Sort to see the effects.

DoSort

End Sub

Define Routines to Calculate an Index and to Do a Sort

The following two routines are needed to calculate an index to be used with the TextArray property and to sort the data:

Function Fgi (r As Integer, c As Integer) As Integer

Fgi = c + Fg1.Cols * r

End Function

Sub DoSort ()

Fg1.Col = 0

Fg1.ColSel = Fg1.Cols - 1

Fg1.Sort = 1 ' Generic ascending.

End Sub

Define Routine to Enter the Data into Microsoft FlexGrid

Next, define a routine that populates the Microsoft FlexGrid control with sample data:

Function RandomString (kind As Integer)

Dim s As String

Select Case kind

Case 0 ' Region.

Select Case (Rnd * 1000) Mod 5

Case 0: s = "1. Northwest"

Case 1: s = "2. Southwest"

Case 2: s = "3. Midwest"

Case 3: s = "4. East"

Case Else: s = "5. Overseas"

End Select

Case 1 ' Product.

Select Case (Rnd * 1000) Mod 5

Case 0: s = "1. Wahoos"

Case 1: s = "2. Trinkets"

Case 2: s = "3. Foobars"

Case Else: s = "4. Applets"

End Select

Case 2 ' Employee.

Select Case (Rnd * 1000) Mod 4

Case 0: s = "Mary"

Case 1: s = "Sarah"

Case 2: s = "Donna"

Case Else: s = "Paula"

End Select

End Select

RandomString = s

End Function

If you run the project at this point, it should look something like this:

Region	Product	Employee	Sales
1. Northwest	1. Wahoos	Mary	5338.73
	1. Wahoos	Paula	7988.84
	3. Foobars	Donna	5924.56
	4. Applets	Donna	9193.77
2. Southwest	2. Trinkets	Donna	3262.06
	2. Trinkets	Donna	3640.19
	4. Applets	Donna	2613.68
	4. Applets	Mary	157.04
			2895.62
			4013.74

Allow Microsoft FlexGrid to Switch Views in Terms of Data Organization

To allow the user to reorganize the data by dragging columns to a new position, add the following two routines.

This routine uses the Tag property to save the column number when the user presses the mouse button, triggering the MouseDown event:


```
Sub Fg1_MouseDown (Button As Integer, _Shift As Integer, X As Single, Y As Single)
```

```
    Fg1.Tag = ""
```

```
    If Fg1.MouseRow <> 0 Then Exit Sub
```

```
    Fg1.Tag = Str(Fg1.MouseCol)
```

```
    MousePointer = vbSizeWE
```

```
End Sub
```

This routine readjusts the columns and sorts the data when the user releases the mouse button, triggering the MouseUp event:

```
Sub Fg1_MouseUp (Button As Integer, Shift As _  
Integer, X As Single, Y As Single)
```

```
    MousePointer = vbDefault
```

```
    If Fg1.Tag = "" Then Exit Sub
```

```
    Fg1.Redraw = False
```

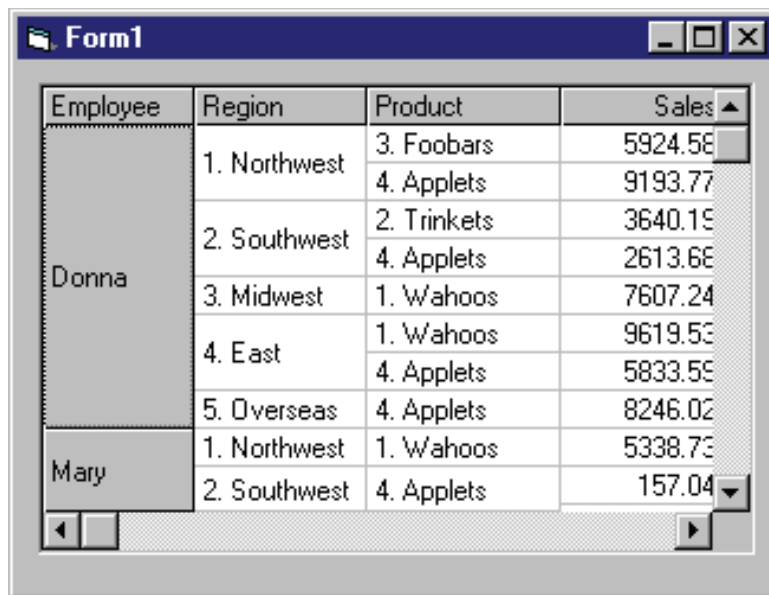
```
    Fg1.ColPosition(Val(Fg1.Tag)) = Fg1.MouseCol
```

```
    DoSort
```

```
    Fg1.Redraw = True
```

```
End Sub
```

At run time, the data is automatically reorganized when the user drags a column to a new position. For example, if the Employee column were dragged to the left, it would then look like this:



Employee	Region	Product	Sales
Donna	1. Northwest	3. Foobars	5924.58
	2. Southwest	4. Applets	9193.77
		2. Trinkets	3640.19
	3. Midwest	4. Applets	2613.68
	4. East	1. Wahoos	7607.24
Mary	5. Overseas	1. Wahoos	9619.53
	1. Northwest	4. Applets	5833.59
		4. Applets	8246.02
	2. Southwest	4. Applets	5338.73
			157.04

Msflex grid Insertion, deletion and modification example

```
Dim lowerbound, upperbound As Long
```

```
Dim i, j, ps As Integer, r As Integer, stnumber As Long
```

```
Dim ONE As String, abc, stname As String
```

```
Option Explicit
```

```
Private Sub Cmdfiletogrid_Click()
```

```
    Dim stfile As Integer, line As String
```

```
    stfile = FreeFile
```

Open "a:\student.dat" For Input As stfile

r = 64

Do While Not EOF(stfile)

Line Input #stfile, line

stnumber = Mid(line, 1, 6)

stname = Mid(line, 7, 20)

fgrd1.TextArray(r) = Str(stnumber)

fgrd1.TextArray(r + 1) = stname

fgrd1.TextArray(r + 2) = "99"

fgrd1.TextArray(r + 3) = "21"

r = r + 4

Loop

End Sub

Private Sub CmdDelete_Click()

'Delete selected row

fgrd1.RemoveItem(fgrd1.Row)

End Sub

Private Sub CmdClear_Click()

' Clear all flexgrid

fgrd1.Clear

End Sub

Private Sub fgrd1_Click()

Dim currow As Integer

currow = fgrd1.Row

Text1.Text = fgrd1.TextMatrix(currow, 0)

Text2.Text = fgrd1.TextMatrix(currow, 1)

Text3.Text = fgrd1.TextMatrix(currow, 2)

Text4.Text = fgrd1.TextMatrix(currow, 3)

End Sub

Private Sub Form_Load()

abc = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

r = 4

Randomize

lowerbound = 960000: upperbound = 969000

For i = 1 To 9

stnumber = Int((upperbound - lowerbound + 1) * Rnd + lowerbound)

fgrd1.TextArray(r) = Str(stnumber)

findname

fgrd1.TextArray(r + 1) = stname

fgrd1.TextArray(r + 2) = "88"

fgrd1.TextArray(r + 3) = "18"

r = r + 4

stnumber = stnumber + 1

```

Next
For i = 10 To 19
fgrd1.TextMatrix(i, 0) = Str(stnumber)
findname
fgrd1.TextMatrix(i, 1) = stname
fgrd1.TextMatrix(i, 2) = "88"
fgrd1.TextMatrix(i, 3) = "18"
stnumber = stnumber + 1
Next
End Sub

```

```

Sub findname()
ONE = ""
stname = ""
For j = 1 To 18
ps = 23 * Rnd
If ps > 0 Then
    ONE = Mid(abc, ps, 1)
    stname = stname + ONE
End If
Next
End Sub

```

Using the FlexGrid

To use the FlexGrid in your application, right click the Toolbox or select the Project menu and choose 'Components'. The Components dialog box appears as shown in Figure 1.

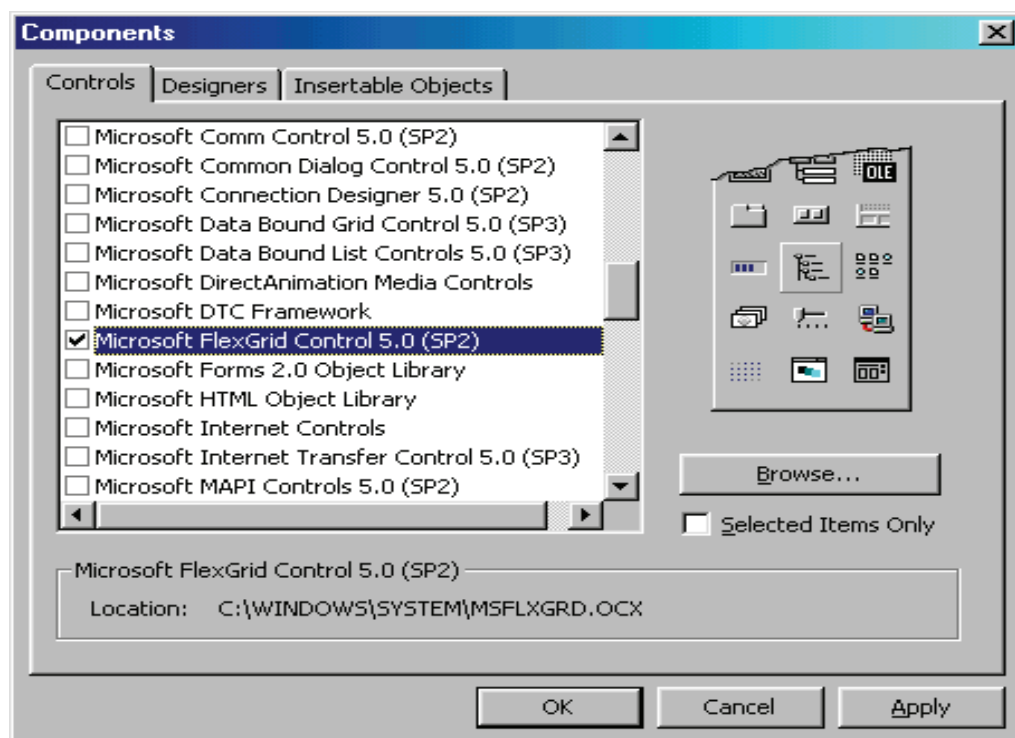


Figure 1: Components Dialog Box

Select Microsoft FlexGrid Control and copy the control onto your form from the Toolbox. The default FlexGrid will appear on the form as shown in Figure 2.

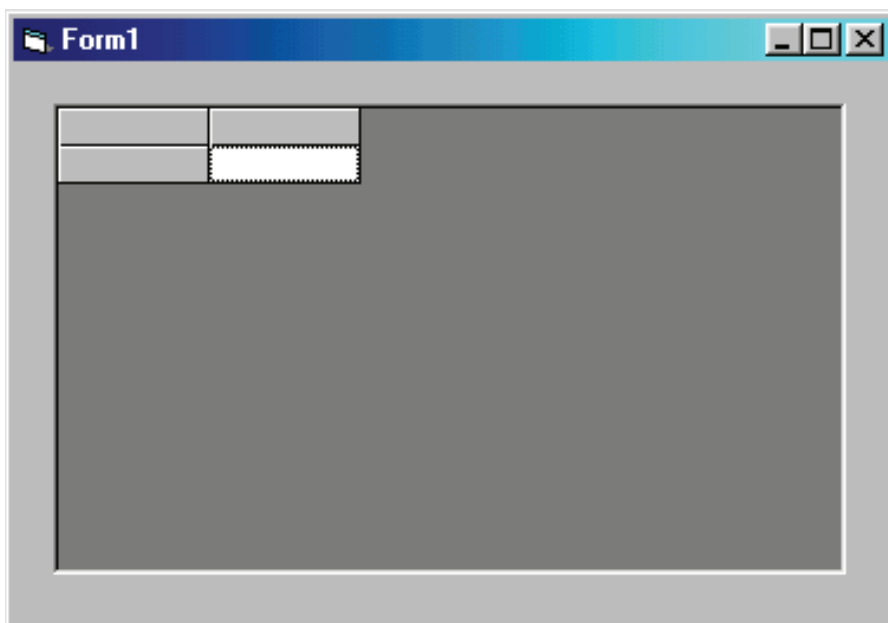


Figure 2: Default FlexGrid Appearance

There are a few simple formatting tips to improve to appearance of the FlexGrid:

First of all, it's good practice to determine the initial number of grid rows and columns using the Row and Col properties respectively. Then set the number of fixed rows and columns (those that hold row and column headings - raised and in grey) with the FixedRow and FixedCol properties respectively.

To ensure the columns are the correct width to fit inside to FlexGrid, first set a variable to the width of the FlexGrid, allowing for a vertical Scrollbar width. Then use this variable to resize to individual column widths, by dividing it by the number of columns.

The height of individual rows can be changed to accommodate multi-line headings. This is done by simply multiplying the appropriate row height by a scaling factor, in this case two, to double the height.

When increasing the size of a row height, to make sure text uses the extra available space, use the WordWrap property the ensure text continues on the the next cell line.

Regarding writing text to the grid there are two main methods. The first one is using the Additem method, this writes text to an entire row in one action and is therefore useful for adding headings. The vbTab constant is used to distinguish adjacent cells. The second method to populate grid cells with text is to directly address individual cells and using the TextMatrix or Text property and set the cell's text.

An example of grid initialisation code would be as follows:

```
Dim lngWidth As Long
Dim intLoopCount As Integer
Const SCROLL_BAR_WIDTH = 320
With MSFlexGrid
    lngWidth = .Width - SCROLL_BAR_WIDTH
    .Cols = 4
```

```

.FixedCols = 1
.Rows = 0
.AddItem vbTab & "Heading Text One" & vbTab & _
    "Heading Text Two" & vbTab & "Heading Text Three" & _
    vbTab & "Heading Text Four"
.Rows = 12
.FixedRows = 1
.WordWrap = True
.RowHeight(0) = .RowHeight(0) * 2

.ColWidth(0) = lngWidth / 4
.ColWidth(1) = lngWidth / 4
.ColWidth(2) = lngWidth / 4
.ColWidth(3) = lngWidth / 4

For intLoopCount = 1 To (.Rows - 1)
    .TextMatrix(intLoopCount, 0) = "Item " & intLoopCount
Next intLoopCount
End With

```

The initialised FlexGrid should appear on the form as shown in Figure 3..

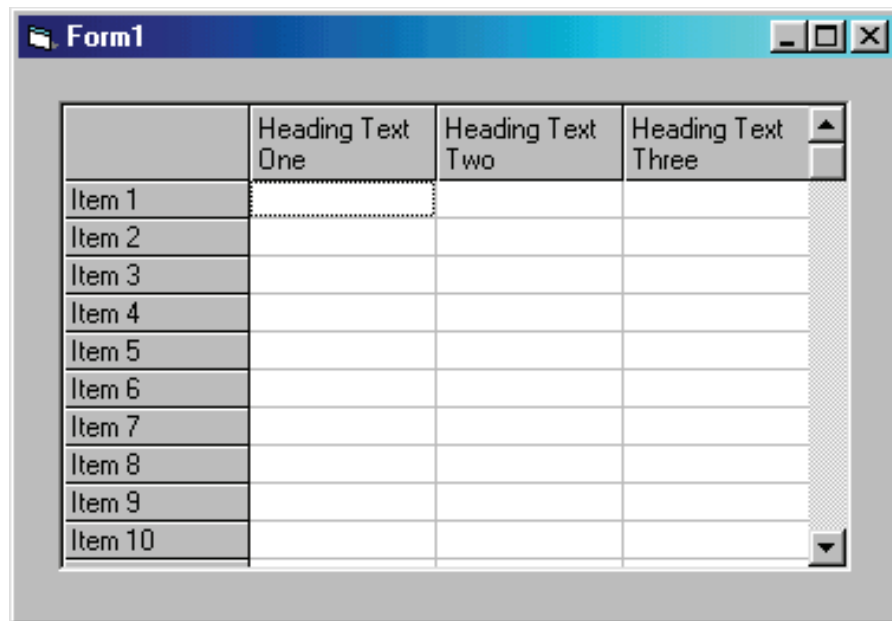


Figure 3: Initialised FlexGrid Appearance

Changing Cell Background & Foreground Colours

The ability to change the individual cell background colour and cell font colour is useful for highlighting particular data states, e.g. values exceeding a specified ceiling, negative values, cells where data cannot be entered etc. To set the background colour of a cell, first reference the cell by setting the Row and Col properties to the appropriate cell, then use the CellBackColor property to set its colour. Similarly use the CellForeColor property to change the colour of the text displayed in a cell. Examples of available colours are found in the FlexGrid's Properties Window under any of the colour properties, especially the palette tab

as this displays a wider range of colours. To set a range of cells to a selected colour each cell must be referenced in turn, unless all the cells in the grid are to have their colour set to the same colour in which case the BackColor and ForeColor property can be used which sets the entire grid's colour.

Populating the FlexGrid

There are two ways to populate Flexgrids: through data-binding and manually.

The data-bound FlexGrid is the easiest but the least flexible approach. Add a Data Control to your form, set its DataBaseName property to an appropriate database and its RecordSource property to a table or to your required SQL query. Then set the FlexGrid's DataSource to the Data Control and when the application is run, the FlexGrid will automatically format the number of columns to the number of fields in your table or returned recordset and the number of rows to the number of records.

However if you wish to display data with greater control, e.g. blank columns and rows separating grouped data, text colour and cell colour dependent upon +ve or -ve values etc., then manually, unbound populating of the FlexGrid is the best option.

Once you have a recordset, data can be directed at any cell within the FlexGrid, which gives the developer much greater control. The following code example takes a recordset which contains two fields one which hold share prices, *shareprice*, and one for share movements, *movement*, and colour codes grid cells depending on the status of the data:

With rsExample

```
.MoveLast  
    'set number of grid rows and columns  
    'to fit recordset  
MSFlexGrid1.Rows = .RecordCount  
MSFlexGrid1.Cols = 3  
.MoveFirst
```

Do

 If !movement < 0 Then

```
        'reference cell to set cell text colour  
        'to Red for shareprice  
MSFlexGrid1.row = .AbsolutePosition + 1  
    MSFlexGrid1.col = 1  
    MSFlexGrid1.CellForeColor = vbRed  
MSFlexGrid1.TextMatrix(.AbsolutePosition + 1, 1) = _  
    !shareprice  
MSFlexGrid1.TextMatrix(.AbsolutePosition + 1, 2) = _  
    !movement
```

 ElseIf !movement = 0 Then

```
        'reference cell to set cell background  
        'colour to Yellow for shareprice  
MSFlexGrid1.row = .AbsolutePosition + 1  
    MSFlexGrid1.col = 1  
    MSFlexGrid1.CellBackColor = vbYellow  
MSFlexGrid1.TextMatrix(.AbsolutePosition + 1, 1) = _
```

```

        !shareprice
MSFlexGrid.TextMatrix(.AbsolutePosition + 1, 2) = _
        !movement
Else
MSFlexGrid1.TextMatrix(.AbsolutePosition + 1, 1) = _
        !shareprice
MSFlexGrid1.TextMatrix(.AbsolutePosition + 1, 2) = _
        !movement
End If
.MoveNext
Loop Until .EOF
End With
Using this approach the FlexGrid can be formatted in any fashion for any recordset.

```

8.3.4 Entering the values in the cells of grid.

Using a Dynamic TextBox to enter Data into the FlexGrid

One disadvantage of the FlexGrid is the absence of a simple method of data entry. Date-bound FlexGrids accept data keyed directly into cells, but these amendments automatically update the recordset which populated the grid and hence the database. So again, data-binding represents a lack of flexibility.

By contrast, unbound FlexGrids do not directly accept data entry, but can accept data through the FlexGrid's KeyPress event or via a Textbox. The Textbox is the favoured method as it lends itself well to validation procedures and is camouflaged well against the FlexGrid. The trick is to set the TextBox's BorderStyle property to '0 -None' and reposition the TextBox over the cell to be edited and resize its width and height to match the cell width and height. This repositioning and resizing can be achieved in one line of code using the TextBoxes Move method in the FlexGrid's Click event or KeyPress event as follows:

```

Private Sub MSFlexGrid1_Click()
    With MSFlexGrid1
        txtDataEntry.Text = .TextMatrix(.row, .col)
        txtDataEntry.Move .CellLeft + .Left, .CellTop + .Top, _
            .CellWidth, .CellHeight
        txtDataEntry.Visible = True
        DoEvents
        txtDataEntry.SetFocus
    End With
End Sub

Private Sub MSFlexGrid1_KeyPress(KeyAscii As Integer)
    Select Case KeyAscii
    Case 45, 46, 47, 48 To 57, 65 To 90, 97 To 122
        txtDataEntry.Text = Chr$(KeyAscii)
        txtDataEntry.SelStart = 1
    End Select
    With MSFlexGrid1
        txtDataEntry.Move .CellLeft + .Left, .CellTop + .Top, _
            .CellWidth, .CellHeight
    End With
End Sub

```

```

        txtDataEntry.Visible = True
        DoEvents
        txtDataEntry.SetFocus
    End With
End Sub

```

The FlexGrid Click event copies the contents of the selected cell to the TextBox ready for editing and snaps the TextBox invisibly to the cell. Each subsequent keypress is then handled by the TextBox KeyPress event which appends to the original cell text.

The FlexGrid KeyPress event operates in the same way as the Click event except no text is copied to the TextBox and new text can then be entered.

```

Private Sub txtDataEntry_KeyPress(KeyAscii As Integer)

    Select Case KeyAscii
        Case 45, 46, 47, 48 To 57, 65 To 90, 97 To 122
            ' These are acceptable keystrokes
        Case Else
            ' These are unacceptable, and should be ignored
            KeyAscii = 0
        End Select
    End Sub

```

The TextBox's KeyPress event can be used to validate data entry as show above.

Once the required text is entered into the TextBox and another cell is selected or return is hit, then the changes are copied back to the FlexGrid cell and a command button is enabled to alert the user the save the new data.

```

Private Sub MSFlexGrid1_LeaveCell()
    Call SaveEdits
End Sub

Private Sub MSFlexGrid1_GotFocus()
    Call SaveEdits
End Sub

Private Sub SaveEdits()
    If Not txtDataEntry.Visible Then Exit Sub
    MSFlexGrid1.TextMatrix(MSFlexGrid1.row, MSFlexGrid1.col) = _
        txtDataEntry.Text
    txtDataEntry.Visible = False

    'enable Save cmdButton to indicate new data to be saved

    cmdSave.Enable = True
End Sub

```

With these six procedures above, data entry and validation will be enabled for the FlexGrid.

The txtDataEntry TextBox used with the FlexGrid should appear as shown in Figure 4.

Date	Currency	Amount
3/6/2000	Pounds	29.50
1/1/2000	Pounds	£4.55
6/8/1999	Dollars	\$9.23
4/1/1997	Francs	F13.70

Figure 4: TextBox used for Data Entry with the FlexGrid

Using Arrays behind FlexGrids

Most of the time, holding data in FlexGrid is more than satisfactory, but occasions arise where a temporary datastore behind the FlexGrid is required. Consider the situation where numeric values are displayed in the FlexGrid, say millions of pounds/dollars. While you would like the grid to *hold* these values right down to pence/cents, you only want to *display* say 3 most significant figures. That is, the grid show £2.02m, but holds £2,019,995.52. For convenience, when the cell is clicked to edit the value, the full value should appear, then return to 3 s.f.'s after amendment, rather like MS Excel.

This is only possible with an array as an intermediate data store between the data source and the FlexGrid. The array should be declared as a datatype appropriate to that of the data intended to be stored within it and dimension to the size of the FlexGrid. Once the data is loaded into the array, the array's contents can be transferred to the FlexGrid using the Format function (or Round function in VB6) to truncate values. Likewise edited values should be retrieved from the array to the txtDataEntry TextBox in the example above, and returned to the array upon completion of editing. A function should be called to dump the array to the FlexGrid upon data retrieval or amendment.

Saving the FlexGrids data should then be a data dump from the array to the database or other data source.

Adding Totals to FlexGrids

Arithmetic operations upon numeric data in FlexGrids can be handled in 2 ways:

- by directly addressing individual grid cells, shown in the code below

```
Dim intRow As Integer
```

```
Dim dblTotal As Double
```

```
With MSFlexGrid1
```

```

For intRow = 1 To 5
    dblTotal = dblTotal + _
    Format(.TextMatrix(intRow, 1), "General Number")
Next intRow

```

```

.TextMatrix(7, 1) = Format(dblTotal, "£#,##0.00")

```

End With

This code sums to values in the first 5 rows of the first column in the FlexGrid and places the total in the row 7, column 1 in a monetary format.

- by addressing array elements, shown in the code below

```

Dim intArrayRow As Integer
Dim dblTotal As Double
Dim arrGridData() as Double

```

```

ReDim arrGridData(10, 10)

```

```

For intArrayRow = 1 To 5
    dblTotal = dblTotal + arrGridData(intArrayRow , 1)
Next intArrayRow

```

```

arrGridData(7, 1) = dblTotal

```

Call subArray2FlxGrd

Again this example sums the first 5 rows and writes them the the seventh row in the first column of the array. The advantage of the array is the data is held as the correct datatype and arrays are faster in terms of read/write access than FlexGrids. Remember to include a call to the procedure to transfer the array data to the FlexGrid.

8.3.5 Scrollbars in MS-Flexgrid.

ScrollBars Property

Syntax

object.**ScrollBars** [=value]

The **ScrollBars** property syntax has these parts:

Part	Description
<i>object</i>	An object expression that evaluates to an object in the Applies To list.
<i>value</i>	An integer or constant that specifies the type of scroll bars, as described in Settings.

Settings

The settings for *value* are:

Constant	Value	Description
flexScrollNone	0	The MSFlexGrid has no scroll bars.
flexScrollHorizontal	1	The MSFlexGrid has a horizontal scroll bar.
flexScrollVertical	2	The MSFlexGrid has a vertical scroll bar.
flexScrollBoth	3	The MSFlexGrid has horizontal and vertical scroll bars. (Default)

Scroll bars appear on an **MSFlexGrid** only if its contents extend beyond its borders and *value* specifies scroll bars. If the **ScrollBars** property is set to **None**, the **MSFlexGrid** will not have scroll bars, regardless of its contents.

8.4 Summary

- The acronym MDI stands for Multiple Document Interface.
- The MDI was designed to simplify the exchange of information among documents, all under the same roof. With the main application, you can maintain multiple open windows, but not multiple copies of the application.
- Flex Grid is a control which facilitates display of tabular data on a Form, along with the facility of Adding a Row, Deleting a Row and Sorting of records etc.
- Using Microsoft's FlexGrid control (MSFLXGRD.OCX) you can create utilities to display, filter, edit, validate and update your data.
- Visual Basic has two separate controls for working with rows and columns of information:

Dbgrid control for displaying and editing the contents of a database

Msflex Grid control is capable of displaying text or pictures in any of its cells. Moreover, you can use the grid to sort information in the tables and format the information for easier viewing

8.5 Self Assessment Questions

1. What do you understand by MDI form in VB?
2. What methods are used in Arranging of Child Form ?
3. What do you understand by Flex Grid control in VB?
4. Describe the MS Flexgrid control?
5. Differentiate b/w Modal dialog box and Modeless dialog Box?
6. What is the syntax of scroll bar in flexgrid?

Unit-9 Graphics and Error Handling in VB

Structure of Unit:

- 9.0 Objectives
- 9.1 Introduction
- 9.2 Concept of Graphics
 - 9.2.1 Fundamentals of Graphics
 - 9.2.2 Graphics Controls
 - 9.2.3 Graphics Methods
- 9.3 Concept of Error Handling
 - 9.3.1 Runtime Errors
 - 9.3.2 Handling runtime Errors by on Error Statements
 - 9.3.3 Err object
 - 9.3.4 Debugging
 - 9.3.5 immediate window
- 9.4 Summary
- 9.5 Self Assessment Questions

9.0 Objectives

After completing this unit, you will be able to:

- Understand the concept of Graphics and Error Handling.
- Understand the graphics control and methods.
- Classify the Error Types.
- Know about the Run time Error.
- Understand the Debugging.
- Understand the immediate window.

9.1 Introduction

Graphics is a very important part of visual basic programming as an attractive interface will be appealing to the users. In the old BASIC, drawing and designing graphics are considered as difficult jobs, as they have to be programmed line by line in a text-based environment. However, in Visual Basic, these jobs have been made easy. Development of an adequate error handling procedure is application dependent. You need to know what type of errors you are looking for and what corrective actions must be taken if these errors are encountered. For example, if a 'divide by zero' is found, you need to decide whether to skip the operation or do something to reset the offending denominator.

Error handling enables the programmer to attempt to recover (i.e., continue executing) from infrequent fatal errors rather than letting them occur and suffering the consequences (such as loss of application data).

9.2 Concept of Graphics

Graphics is a very important part of visual basic programming as an attractive interface will be appealing to the users. In the old BASIC, drawing and designing graphics are considered as difficult jobs, as they have to be programmed line by line in a text-based environment. However, in Visual Basic, these jobs have been made easy. There are four basic controls in VB that you can use to draw graphics on your form: the line control, the shape control, the image box and the picture box

9.2.1 Fundamentals of Graphics

Graphics is a very important part of visual basic programming as an attractive interface will be appealing to the users.

However, in Visual Basic, these jobs have been made easy. There are four basic controls in VB that you can use to draw graphics on your form: the line control, the shape control, the image box and the picture box

You can place graphics on three controls:

1. Form
2. Picture Box
3. Image Box

9.2.2 Graphics Controls

The line and Shape controls

To draw a straight line, just click on the line control and then use your mouse to draw the line on the form. After drawing the line, you can then change its color, width and style using the `BorderColor`, `BorderWidth` and `BorderStyle` properties.

Similarly, to draw a shape, just click on the shape control and draw the shape on the form. The default shape is a rectangle, with the shape property set at 0. You can change the shape to square, oval, circle and rounded rectangle by changing the shape property's value to 1, 2, 3 4, and 5 respectively. In addition, you can change its background color using the `BackColor` property, its border style using the `BorderStyle` property, its border color using the `BorderColor` property as well its border width using the `BorderWidth` property.

Example 9.1

The program in this example allows the user to change the shape by selecting a particular shape from a list of options from a list box, as well as changing its color through a common dialog box.

The objects to be inserted in the form are a list box, a command button, a shape control and a common dialog box. The common dialog box can be inserted by clicking on 'project' on the menu and then select the Microsoft Common Dialog Control 6.0 by clicking the check box. After that, the Microsoft Common Dialog Control 6.0 will appear in the toolbox; and you can drag it into the form. The list of items can be added to the list box through the `AddItem` method. The procedure for the common dialog box to present the standard colors is as follows:

```
CommonDialog1.Flags = &H1&
```

```
CommonDialog1.ShowColor
```

Shape1.BackColor = CommonDialog1.Color

The last line will change the background color of the shape by clicking on a particular color on the common dialog box as shown in the Figure below:

The Interface.	The Code
<div data-bbox="293 300 1044 772"> </div> <div data-bbox="438 783 881 1472"> </div>	<pre> Private Sub Form_Load() List1.AddItem "Rectangle" List1.AddItem "Square" List1.AddItem "Oval" List1.AddItem "Circle" List1.AddItem "Rounded Rectangle" List1.AddItem "Rounded Square" End Sub Private Sub List1_Click() Select Case List1.ListIndex Case 0 Shape1.Shape = 0 Case 1 Shape1.Shape = 1 Case 2 Shape1.Shape = 2 Case 3 Shape1.Shape = 3 Case 4 Shape1.Shape = 4 Case 5 Shape1.Shape = 5 End Select End Sub Private Sub Command1_Click() CommonDialog1.Flags = &H1& CommonDialog1.ShowColor Shape1.BackColor = CommonDialog1.Color End Sub </pre>

The Image Box and the Picture Box

Using the line and shape controls to draw graphics will only enable you to create a simple design. In order to improve the look of the interface, you need to put in images and pictures of your own. Fortunately, there are two very powerful graphics tools you can use in Visual Basic which are the image box and the picture box.

To load a picture or image into an image box or a picture box, you can click on the picture property in the properties window and a dialog box will appear which will prompt the user to select a certain picture file. You can also load a picture at runtime by using the LoadPicture () method.

The syntax is

Image1.Picture= LoadPicture("C:\path name\picture file name") or

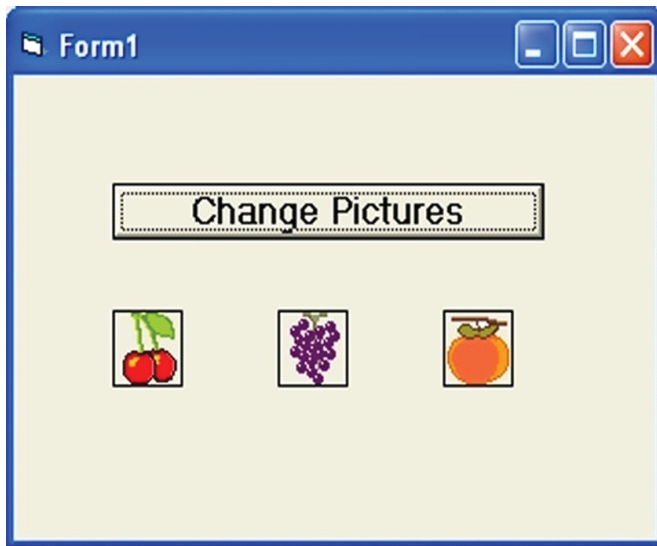
picture1.Picture= LoadPicture("C:\path name\picture name")

For example, the following statement will load the grape.gif picture into the image box.

Image1.Picture= LoadPicture("C:\My Folder\VB program\Images\grape.gif")

Example 9.2

In this example, each time you click on the 'change pictures' button as shown in Figure 19.2, you will be able to see three images loaded into the image boxes. This program uses the Rnd function to generate random integers and then uses the LoadPicture method to load different pictures into the image boxes using the If...Then...Statements based on the random numbers generated. The output is shown in Figure 19.2 below



```
Dim a, b, c As Integer
Private Sub Command1_Click ()
Randomize Timer
a = 3 + Int(Rnd * 3)
b = 3 + Int(Rnd * 3)
c = 3 + Int(Rnd * 3)
```

```
If a = 3 Then
Image1(0).Picture = LoadPicture("C:\My
Folder\VB program\Images\grape.gif")
End If
If a = 4 Then
Image1(0).Picture = LoadPicture("C:\My
Folder\VB program\Images\cherry.gif")
End If
If a = 5 Then
Image1(0).Picture = LoadPicture("C:\My
Folder\VB program\Images\orange.gif")
End If
If b = 3 Then
Image1(1).Picture = LoadPicture("C:\My
Folder\VB program\Images\grape.gif")
End If
If b = 4 Then
Image1(1).Picture = LoadPicture("C:\My
Folder\VB program\Images\cherry.gif")
End If
If b = 5 Then
Image1(1).Picture = LoadPicture("C:\My
Folder\VB program\Images\orange.gif")
End If
If c = 3 Then
Image1(2).Picture = LoadPicture("C:\My
Folder\VB program\Images\grape.gif")
End If
If c = 4 Then
Image1(2).Picture = LoadPicture("C:\My
Folder\VB program\Images\cherry.gif")
End If
If c = 5 Then
Image1(2).Picture = LoadPicture("C:\My
Folder\VB program\Images\orange.gif")
End If
End Sub
```

9.2.3 Graphics Methods

PSet, Line and Circle Drawing Methods

Other than using the line and shape controls to draw graphics on the form, you can also use the Pset, Line and Circle methods to draw graphics on the form.

(a) The Pset Method

The Pset method draw a dot on the screen, it takes the format

Pset (x , y), color

(x,y) is the coordinates of the point and color is its color. To specify the color, you can use the color codes or the standard VB color constant such as VbRed, VbBlue, VbGreen and etc. For example, Pset(100,200), VbRed will display a red dot at the (100,200) coordinates.

The Pset method can also be used to draw a straight line on the form. The procedure is

For x= a to b

Pset(x,x)

Next x

This procedure will draw a line starting from the point (a,a) and to the point (b,b). For example, the following procedure will draw a magenta line from the point (0,0) to the point (1000,1000).

For x= 0 to 100

Pset(x,x) , vbMagenta

Next x

(b) The Line Method

Although the Pset method can be used to draw a straight line on the form, it is a little slow. It is better to use the Line method if you want to draw a straight line faster. The format of the Line command is shown below. It draws a line from the point (x1, y1) to the point (x2, y2) and the color constant will determine the color of the line.

Line (x1, y1)-(x2, y2), color

For example, the following command will draw a red line from the point (0, 0) to the point (1000, 2000).

Line (0, 0)-(1000, 2000), VbRed

The Line method can also be used to draw a rectangle. The format is

Line (x1-y1)-(x2, y2), color, B

The four corners of the rectangle are (x1-y1), (x2-y1), (x1-y2) and (x2, y2)

Another variation of the Line method is to fill the rectangle with a certain color. The format is

Line (x1, y1)-(x2, y2), color, BF

If you wish to draw the graphics in a picture box, you can use the following formats

Picture1.Line (x1, y1)-(x2, y2), color

Picture1.Line (x1-y1)-(x2, y2), color, B

Picture1.Line (x1-y1)-(x2, y2), color, BF

Picture1.Circle (x1, y1), radius, color

(c) The Circle Method

The circle method takes the following format

Circle (x1, y1), radius, color

That draws a circle centered at (x1, y1), with a certain radius and a certain border color. For example, the procedure

Circle (400, 400), 500, VbRed

draws a circle centered at (400, 400) with a radius of 500 twips and a red border.

9.3 Concept of Error Handling

Development of an adequate error handling procedure is application dependent. You need to know what type of errors you are looking for and what corrective actions must be taken if these errors are encountered. For example, if a 'divide by zero' is found, you need to decide whether to skip the operation or do something to reset the offending denominator.

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

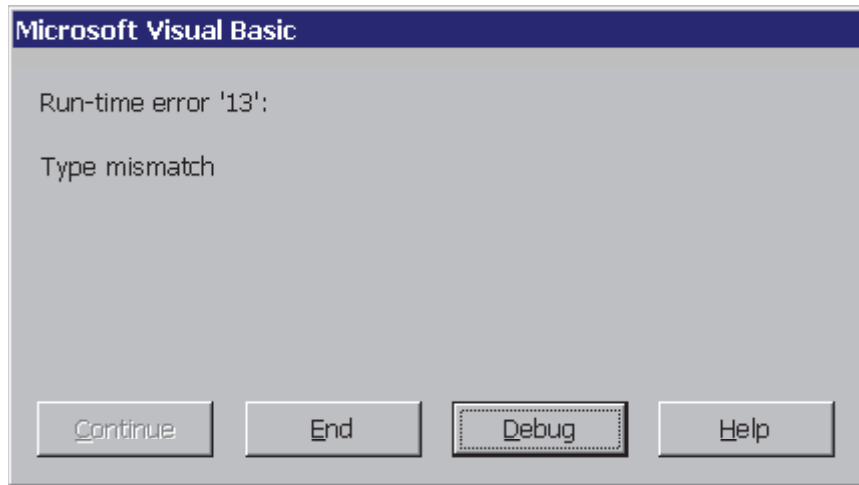
These errors can be grouped into three categories:

1. Syntax errors
 2. Run-time errors
 3. Logic errors
- Syntax errors occur when you mistype a command or leave out an expected phrase or argument. Visual Basic detects these errors as they occur and even provides help in correcting them. You cannot run a Visual Basic program until all syntax errors have been corrected.
 - Run-time errors are usually beyond your program's control. Examples include: when a variable takes on an unexpected value (divide by zero), when a drive door is left open, or when a file is not found. Visual Basic allows you to trap such errors and make attempts to correct them.
 - logical errors are the most difficult to find. With logic errors, the program will usually run, but will produce incorrect or unexpected results. The Visual Basic debugger is an aid in detecting logical errors.

9.3.1 Runtime Errors

Run-time errors are usually beyond your program's control. Examples include: when a variable takes on an unexpected value (divide by zero), when a drive door is left open, or when a file is not found. Visual Basic allows you to trap such errors and make attempts to correct them.

These occur while your application is running, when Visual Basic finds a statement that is impossible to carry out, such as multiplying the contents of TextBox1 by 10 when textbox1 contains 'Hello'. If this error occurs when running Visual Basic, a dialog is displayed like the one below. You are given the error number along with its message. You have the option to End the current program, Debug the program which pauses the code, or to get help on that error



Visual Basic Error Dialog Box

If however an error occurs once your application has been made into an exe, the error number and message are given. But once the OK button has been pressed, the program closes.

9.3.2 Handling runtime Errors by on Error Statements

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

- Error trapping is enabled with the On Error statement:

On Error GoTo errlabel

Yes, this uses the dreaded GoTo statement! Any time a run-time error occurs following this line, program control is transferred to the line labeled errlabel. Recall a labeled line is simply a line with the label followed by a colon (:).

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

```
Sub SubExample()
```

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

```
On Error GoTo HandleErrors
```

```
[Procedure code]
```

```
Exit Sub
```

```
HandleErrors:
```

```
Error handling code]
```

```
End Sub
```

Once you have set up the variable declarations, constant definitions, and any other procedure preliminaries, the On Error statement is executed to enable error trapping. Your normal procedure code follows this statement. The error handling code goes at the end of the procedure, following the HandleErrors statement

label. This is the code that is executed if an error is encountered anywhere in the Sub procedure. Note you must exit (with Exit Sub) from the code before reaching the HandleErrors line to avoid inadvertent execution of the error handling code.

- Since the error handling code is in the same procedure where an error occurs, all variables in that procedure are available for possible corrective action. If at some time in your procedure, you want to turn off error trapping, that is done with the following statement:

On Error GoTo 0

- Once a run-time error occurs, we would like to know what the error is and attempt to fix it. This is done in the error handling code.
- Visual Basic offers help in identifying run-time errors. The Err object returns, in its Number property (Err.Number), the number associated with the current error condition. (The Err function has other useful properties that we won't cover here - consult on-line help for further information.) The Error() function takes this error number as its argument and returns a string description of the error. Consult on-line help for Visual Basic run-time error numbers and their descriptions.
- Once an error has been trapped and some action taken, control must be returned to your application. That control is returned via the Resume statement.

Use On Error

A Visual Basic program uses the On Error statement to register error handling code. This statement can take one of three forms:

- On Error GoTo 0
- On Error Resume Next
- On Error GoTo line

These forms tell Visual Basic what it should do when the program encounters an error. The three forms are described in the following sections.

On Error GoTo 0

On Error GoTo 0 is relatively straightforward. It simply cancels any currently installed error handler assigned by a previous On Error GoTo line or On Error Resume Next. If the program encounters an error after this statement executes, it crashes.

On Error Resume Next

On Error Resume Next makes the program ignore errors. When it encounters an error, the program continues execution after the statement that caused the error. When a program uses On Error Resume Next, it should check the Err object after every operation that might cause an error. If the value Err.Number is nonzero, the operation caused an error and the program can take special action. The program should check Err.Number immediately after the statement in question. Certain other actions reset the Err object and remove the previous error information.

Many programs use On Error Resume Next when they present a common dialog to the user. The CommonDialog control's CancelError property indicates whether the control should raise an error if the user cancels the dialog. The following code fragment shows how a program can use CancelError to decide whether to continue an action such as loading a file.

```

' Generate an error if the user cancels.
dlgOpenFile.CancelError = True

' Ignore errors for now.
On Error Resume Next

' Present the dialog.
dlgOpenFile.ShowOpen

' See if there was an error.
If Err.Number = cdlCancel Then
    ' The user canceled. Do nothing.
    Exit Sub
ElseIf Err.Number <> 0 Then
    ' Unknown error. Take more action.
    :
End If

' Resume normal error handling.
On Error GoTo 0

```

On Error GoTo Line

The On Error GoTo line statement registers a new error handler. If the program encounters an error, it passes control to the error handler beginning at the indicated line number or label. The error handler can then take appropriate action.

The following code shows a simple error handler that catches unexpected errors and describes them to the user.

```

Private Sub DoSomething()
    ' Install the error handler.
    On Error GoTo UnexpectedError

    ' Do stuff.
    :

    ' Do not pass through into the error handler code.
    Exit Sub

UnexpectedError:
    ' Describe the error to the user.
    MsgBox "Unexpected error" & _
        Str$(Err.Number) & _
        " in subroutine DoSomething." & _
        vbCrLf & _
        Err.Description
    Exit Sub
End Sub

```

9.3.3 Err object

To help out programmers, VB6 now provides a built-in object called “Err”. On occurrence of an error, VB fills the properties of the Err object with information that uniquely identifies the error and with information that you can use in your code to figure out what actions to take as a result of the error.

The properties of the **Err** object are set by the generator of an error—Visual Basic, an object, or the programmer.

When a run-time error occurs, the properties of the **Err** object are filled with information that uniquely identifies the error and that you can use to handle the error. To generate a run-time error in your code, use the **Raise** method.

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

Use the **Raise** method rather than the **Error** statement to generate run-time errors for system errors and class modules. Your decision about whether to use the **Raise** method in other code depends on the richness of the information you want to return.

The **Err** object is an intrinsic object with global scope. Therefore, you do not need to create an instance of it in your code.

[Example](#)

```
[* Internal error: Invalid file format. | In-line.WMF *]
```

This example uses the properties of the **Err** object in constructing an error-message dialog box. Notice that if you use the **Clear** method first, when you generate a Visual Basic error with the **Raise** method, Visual Basic’s default values become the properties of the **Err** object.

```
Dim Msg As String
```

```
‘ If an error occurs, construct an error message.
```

```
On Error Resume Next ‘ Defer error handling.
```

```
Err.Clear()
```

```
Err.Raise(6) ‘ Generate an “Overflow” error.
```

```
‘ Check for error, then show message.
```

```
If Err.Number <> 0 Then
```

```
Msg = “Error # “ & Str(Err.Number) & “ was generated by “ _  
    & Err.Source & ControlChars.CrLf & Err.Description
```

```
MsgBox(Msg, MsgBoxStyle.Information, “Error”)
```

```
End If
```

Methods of the Err Object

The Err object has two methods that you can invoke in your applications. These methods are also invoked automatically in Visual Basic applications as described in the following sections.

Clear Method

The Clear method of the Err object reinitializes all the Err properties. You can use the Clear method at any time to explicitly reset the Err object. Visual Basic also invokes Clear automatically in the following three situations:

- When either a Resume or Resume Next statement is encountered.
- At an Exit Sub, Exit Function, or Exit Property statement.
- Each time an On Error statement is executed.

Raise Method

The Raise method of the Err object is used to generate errors within code. You can use Raise to create your own runtime errors that will be used elsewhere in your application. The Raise method can also be used to pass error information from a class module to another application that uses objects of that class.

The syntax for the Raise method is

Err.Raise Number, [Source], [Description], [Helpfile], [Helpcontext]

The arguments of the Raise method correspond to the properties of the Err object. They are as follows:

- **Number.** This is a required argument. It is a long integer that contains the error number. Remember that Visual Basic errors fall between 0 and 65,535, inclusive, and VB reserves error numbers 0-512 for itself. If you are defining any of your own errors, use numbers within the range 513-65,535.
- **Source.** An optional argument identifying where an error occurred. Source is a string property that can contain any information that will help point to the exact location of the problem. It may contain the class module name, form name, and procedure. The standard is to set the Source to project.class.
- **Description.** An optional argument describing the error that has occurred. If the description is not set, Visual Basic examines the Number argument to determine whether the error number is recognized (between 0 and 65,535). If Number does map to a Visual Basic error, the Description property is set automatically. If Number does not correspond to a Visual Basic error, the Description is set to Application-Defined or Object-Defined Error.
- **HelpFile.** Identifies a help file and a path to the file. This optional argument sets the Err.HelpFile property, which can be used with the HelpContext property, to provide help to the user. If HelpFile is not specified, the path and filename for the Visual Basic Help file is used.
- **HelpContext.** Used with the HelpFile argument, the optional HelpContext argument identifies a topic within the HelpFile. If the HelpContext is not specified, Visual Basic uses the help topic of the Visual Basic Help file corresponding to the Number argument (if a topic is available).

The Raise method is usually used within class modules.

If a routine cannot handle an error itself, it should raise a new error that makes sense within its context. For example, the following routine attempts to read a data file. If the file is not found, the FileOpenError error handler raises the myappErrNoInputFile error. This gives the calling subroutine more information than Visual Basic's initial file not found error. The error Visual Basic generates indicates that some file was not found. The new error explains that an input data file was not found. The Err.Description field even includes the name of the file that was not found.

```

' Define application error constants.
Private Const myappErrNoInputFile = vbObjectError + 1000
:
' Define Visual Basic error constants.
Private Const vbErrFileNotFound = 53
:
Private Sub ReadInputData(ByVal file_name As String)
Dim file_number As Integer

    ' Open the file.
    file_number = FreeFile
    On Error GoTo FileOpenError
    Open file_name For Input As file_number

    ' Process the file.
    On Error GoTo FileReadError
    :
    ' Process the file here.
    :
    ' Close the file.
    Close file_number
    Exit Sub

FileOpenError:
    ' There was an error opening the file.
    If Err.Number = vbErrFileNotFound Then
        ' It's a file not found error. Convert it
        ' to myappErrNoInputFile.
        Err.Raise myappErrNoInputFile, _
            "MyApp.ReadInputData", _
            "Could not open input file """" & _
                file_name & """"."
    Else
        ' It's some other error. Reraise it so some
        ' other routine can catch it.
        Err.Raise Err.Number, _
            Err.Source, _
            Err.Description, _
            Err.HelpFile, _
            Err.HelpContext
    End If
    Exit Sub

FileReadError:
    ' There was an error reading the file.
    :
    Exit Sub
End Sub

```

A program could invoke this subroutine using code similar to the following. The error handler uses the information stored in the Err object by the Raise method to present a message to the user.

```
On Error GoTo DataInputError
ReadInputData "c:\mydata.dat"
Exit Sub
```

DataInputError:

```
' There was an error loading the data.
MsgBox "Error" & Str$(Err.Number) & _
    " loading the input data." & vbCrLf & _
    Err.Description
```

Routines that present messages to users normally format the error information as shown in the previous code. To make that formatting as simple as possible, routines should not format the error description in the Raise statement. For example, the following code formats an error's description.

```
Err.Raise myappErrNoInputFile, _
    "MyApp.ReadInputData", _
    "Error" & Str$(myappErrNoInputFile) & _
    " opening the input file."
```

When this error occurs, the error handler that catches the error will probably display a message like this one:

Error -2147220504 loading the input data.

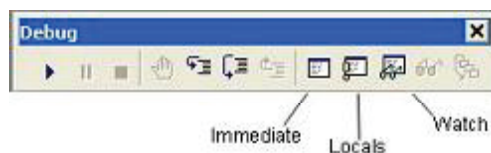
Error -2147220504 opening the input file.

Leave the formatting to the routine that actually records the error or presents the message to the user.

9.3.4 Debugging

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

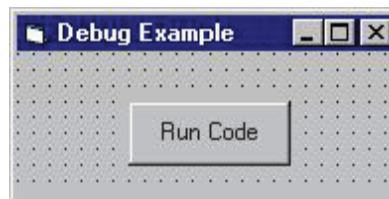
- Debugging a code is an art, not a science. There are no prescribed processes that you can follow to eliminate all logic errors in your program. The usual approach is to eliminate them as they are discovered.
- What we'll do here is present the debugging tools available in the Visual Basic environment (several of which appear as buttons on the toolbar) and describe their use with an example. You, as the program designer, should select the debugging approach and tools you feel most comfortable with.
- The interface between your application and the debugging tools is via three different debug windows: the Immediate Window, the Locals Window, and the Watch Window. These windows can be accessed from the View menu (the Immediate Window can be accessed by pressing Ctrl+G). Or, they can be selected from the Debug Toolbar (accessed using the Toolbars option under the View menu):



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

Example - Debugging

1. Unlike other examples, we'll do this one as a group. It will be used to demonstrate use of the debugging tools.
2. The example simply has a form with a single command button. The button is used to execute some code. We won't be real careful about proper naming conventions and such in this example.



3. The code attached to this button's Click event is a simple loop that evaluates a function at several values.

```
Private Sub Command1_Click()
```

```
Dim X As Integer, Y As Integer
```

```
X = 0
```

```
Do
```

```
Y = Fcn(X)
```

```
X = X + 1
```

```
Loop While X <= 20
```

```
End Sub
```

This code begins with an X value of 0 and computes the Y value using the general integer function Fcn. It then increments X by 1 and repeats the Loop. It continues looping While X is less than or equal to 20.

The function Fcn is computed using:

```
Function Fcn(X As Integer) As Integer
```

```
Fcn = CInt(0.1 * X ^ 2)
```

```
End Function
```

Admittedly, this code doesn't do much, especially without any output, but it makes a good example for looking at debugger use. Set up the application and get ready to try debugging.

9.3.5 immediate window

The **Immediate** window is used at design time to debug and evaluate expressions, execute statements, print variable values, and so forth. It allows you to enter expressions to be evaluated or executed by the development language during debugging. To display the **Immediate** window, open a project for editing, then choose **Windows** from the **Debug** menu and select **Immediate**.

You can use this window to issue individual Visual Studio commands. The available commands include `EvaluateStatement`, which can be used to assign values to variables. The **Immediate** window also supports IntelliSense.

While testing a program for correctness (usually called debugging) it is often advantageous to see interim results in a calculation, or to check on values being passed to a subroutine. Setting breakpoints to stop the program at specific lines can help, as can sending printed values to a second window. VB provides a dockable window for text called the **Immediate Window**. It is available under the View menu or you can press CTRL+G to bring it into view. It is called the Immediate Window because it can be used to execute BASIC commands immediately. For example:

```
Print Atn(1) * 4
```

When you type that line into the Immediate Window and press Enter, VB interprets that line as a VB command and executes the command. In this case it prints the number 3.14159265358979 on the line following the command. (For expediency, the question mark can be substituted for the Print command in most circumstances.)

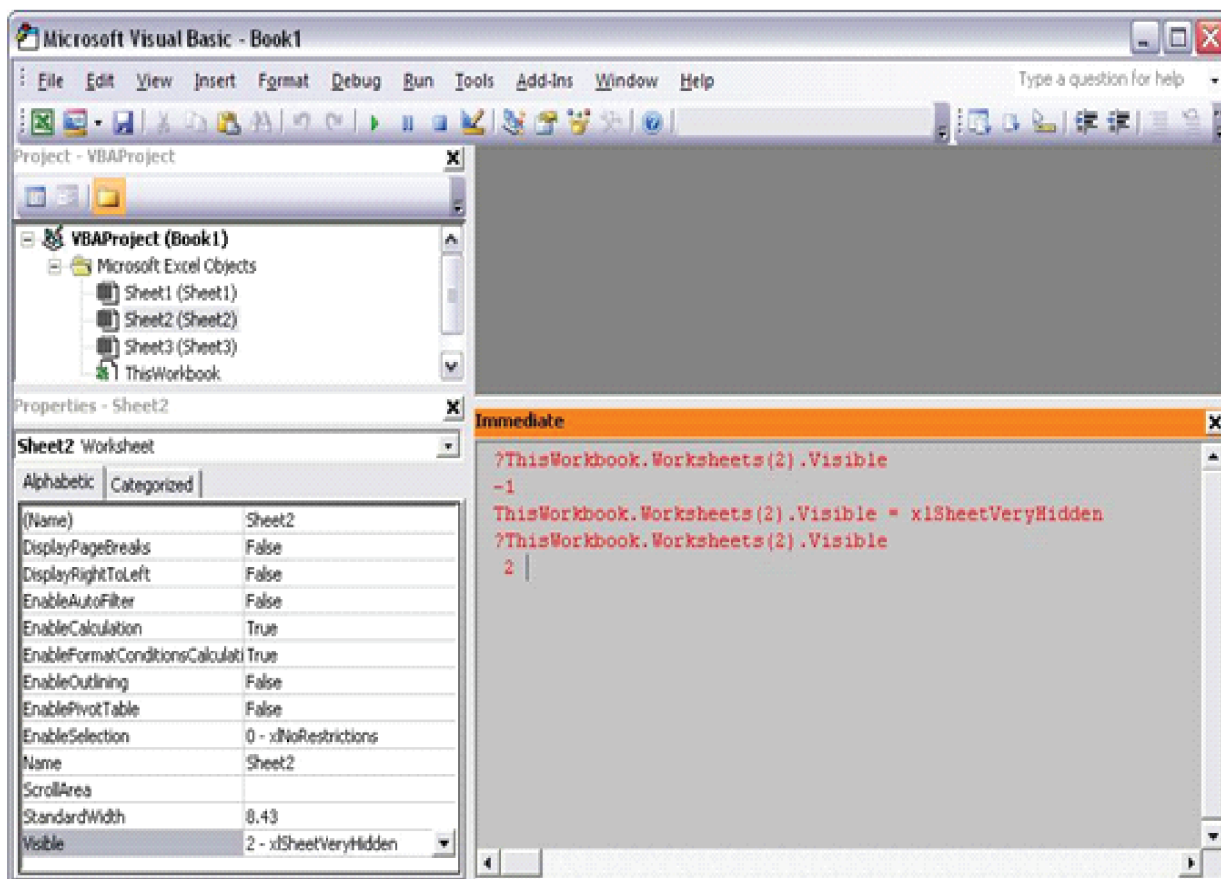
You can send text to that same window from a running program using the Print method of the Debug object:

```
Private Sub Text1_KeyPress(KeyAscii As Integer)
    Debug.Print Chr(KeyAscii), KeyAscii
```

```
End Sub
```

In this case, with a Textbox called Text1 on some form, every letter a user enters into the textbox will be echoed to the Immediate Window, as well as its numerical ASCII value.

Because text is sent to the Immediate Window using the Debug object, many people refer to the Immediate Window as the Debug Window. You may find it listed either way in this literature.



Displaying the Values of Variables

This window can be particularly useful while debugging an application. For example, to check the value of a variable `myVar`, you can use the [Print Command](#):

```
>Debug.Print varA
```

The question mark (?) is an alias for `Debug.Print`, so this command can also be written:

```
>? varA
```

Both versions of this command will return the value of the variable `varA`.

Note

To issue a Visual Studio command in the **Immediate** window, you must preface the command with a greater than sign (>). To enter multiple commands, switch to the **Command** window.

Design Time Expression Evaluation

You can use the **Immediate** window to execute a function or subroutine at design time.

To execute a function at design time

1. Copy the following code into a Visual Basic console application:
2. Module Module1
- 3.
4. Sub Main()
5. MyFunction(5)
6. End Sub
- 7.
8. Function MyFunction(ByVal input as Integer) As Integer
9. Return input * 2
10. End Function
- 11.
12. End Module
13. On the **Debug** menu, click **Windows**, and then click **Immediate**.
14. Type `?MyFunction(2)` in the **Immediate** window and press Enter.

The **Immediate** window will run `MyFunction` and display 4.

If the function or subroutine contains a breakpoint, Visual Studio will break execution at the appropriate point. You can then use the debugger windows to examine your program state.

You cannot use design time expression evaluation in project types that require starting up an execution environment, including Visual Studio Tools for Office projects, Web projects, Smart Device projects, and SQL projects.

Design Time Expression Evaluation in Multi-Project Solutions

When establishing the context for design time expression evaluation, Visual Studio references the currently selected project in Solution Explorer. If no project is selected in Solution Explorer, Visual Studio attempts to evaluate the function against the startup project. If the function cannot be evaluated in the current context, you will receive an error message. If you are attempting to evaluate a function in a project that is not the startup project for the solution and you receive an error, try selecting the project in Solution Explorer and attempt the evaluation again.

Entering Commands

You must enter the greater than sign (>) when issuing Visual Studio commands in the **Immediate** window. Use the UPARROW and DOWNARROW keys to scroll through previously issued commands.

Task	Solution	Example
Evaluate an expression.	Preface the expression with a question mark (?).	? a+b
Temporarily enter Command mode while in Immediate mode (to execute a single command).	Enter the command, prefacing it with a greater than sign (>).	>alias
Switch to the Command window.	Enter cmd into the window, prefacing it with a greater than sign (>).	>cmd
Switch back to the Immediate window.	Enter immed into the window without the greater than sign (>).	immed

Mark Mode

When you click on any previous line in the **Immediate** window, you shift automatically into Mark mode. This allows you to select, edit, and copy the text of previous commands as you would in any text editor, and paste them into the current line.

The Equals (=) Sign

The window used to enter the `EvaluateStatement` command determines whether an equals sign (=) is interpreted as a comparison operator or as an assignment operator.

In the **Immediate** window, an equals sign (=) is interpreted as an assignment operator. So, for example, the command

```
>Debug.EvaluateStatement(varA=varB)
```

will assign to variable `varA` the value of variable `varB`.

In the **Command** window, by contrast, an equals sign (=) is interpreted as a comparison operator. You cannot use assignment operations in the **Command** window. So, for example, if the values of variables `varA` and `varB` are different, then the command

```
>Debug.EvaluateStatement(varA=varB)
```

will return a value of `False`.

First-Chance Exception Notifications

In some settings configurations, first-chance exception notifications are displayed in the **Immediate** window.

To toggle first-chance exception notifications in the Immediate window

1. On the **View** menu, click **Other Windows**, and click **Output**.
2. Right-click on the text area of the **Output** window, and select or deselect **Exception Messages**.

9.4 Summary

This chapter is intended to provide a basic grounding in Graphics drawing in Visual Basic. The area of graphics drawing is vast and it is impossible to cover everything in a single chapter. Hopefully enough has been covered here to give you the confidence to experiment and learn more.

Handling run-time errors is something all applications must do if they are to be robust and reliable.

The key points for error handling are:

- There are two steps to handling run-time errors:
 1. Trap the error by enabling an error handler using the On Error statement.
 2. Handle the error by examining the properties of the Err object and writing code to deal with the problem.
- Error handlers can be dedicated blocks of code enabled by using On Error Goto **label** or can be inline handlers enabled by using On Error Resume Next.
- You can raise your own errors by calling the Raise method of the Err object.
- Do your best to handle run-time errors rather than just inform the user of the problem, but if you can't do anything but display a message, make it as informative as possible.
- Avoid terminating the application if at all possible.

The Err object was introduced in Visual Basic 4.0. For backward compatibility, VB continues to support the Err and Error statements and functions. Any new code should be using the Err object and legacy code should be converted to use the Err object.

9.5 Self Assessment Questions

1. What type of Error occurs in VB?
2. What methods are used to handle errors?
3. What are on Error and resume statements?
4. what do you understand by Err Object?

Unit - 10 : Connectivity with database using ADODC Control

Structure of Unit:

- 10.0 Objective
- 10.1 Introduction
- 10.2 Bounded Control
- 10.3 Un Bounded Control
- 10.4 ADODC Steps
 - 10.4.1 Record set
 - 10.4.2 Record Pointer
 - 10.4.3 Methods
 - 10.4.4 Properties
 - 10.4.5 Accessing and Navigation
- 10.5 Dynaset and Snapshot
- 10.6 Summary
- 10.7 Self Assessment Questions

10.0 Objective

The main objective of this unit is that enable the environment to create an application / software to solve the user's problem.

10.1 Introduction

In this unit we provide the details about the environment of an application. An application design using two different sections and every section have their independent role in our application.

Front - End :

The designing part of our application which will display at user's screen for the manipulation of data known as Front End.

Back - End :

The background area of our application that can provide permission to store data in permanent memory with various features for the further transaction on data known as **Back End**.

The unit have all the set of tool with their properties and method that can set connection between front and back end.

10.2 Bounded Control

All the set of components that are utilize by binding their properties and methods with any of mediator known as Bounded Controls. These are ADO and RDO.

10.3 Un Bounded Control

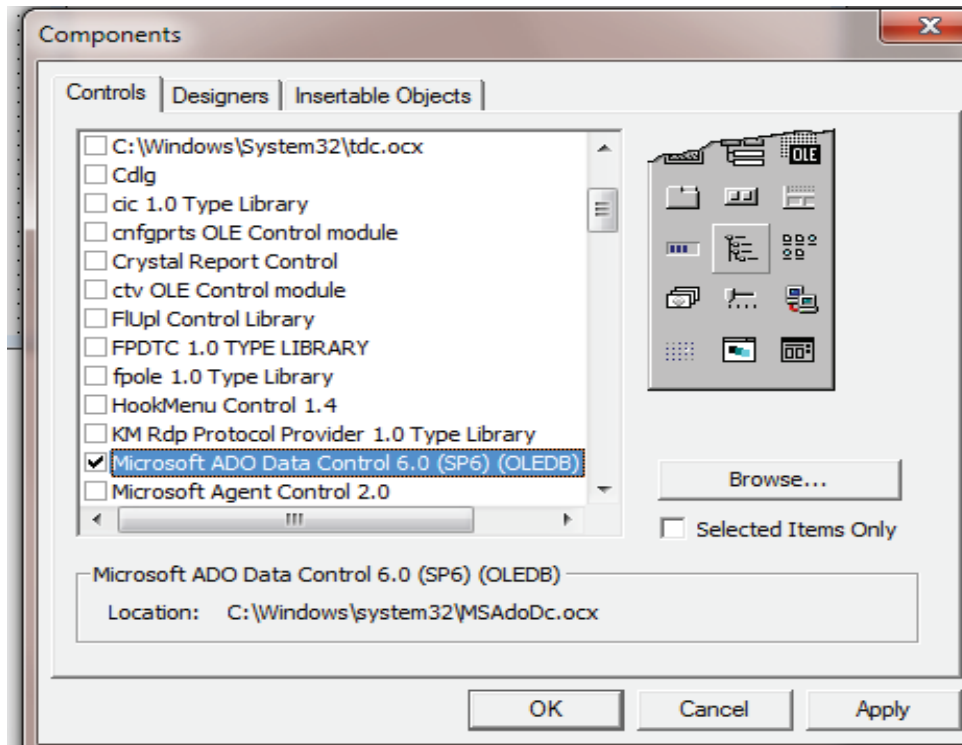
All the set of components that are utilize directly without binding their properties and methods with any of mediator known as Un Bounded Controls. These are Data Grid and Flex Grid control.

10.4 Steps to Activate the ADODC

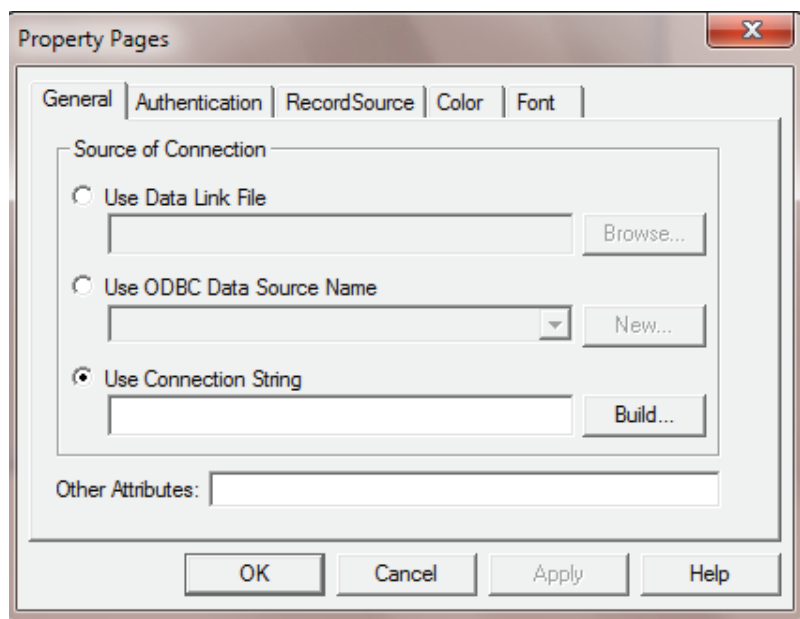
To implement the ADODC we have to use some set of steps in our project. These are as follows-

Design the form according to user's requirement.

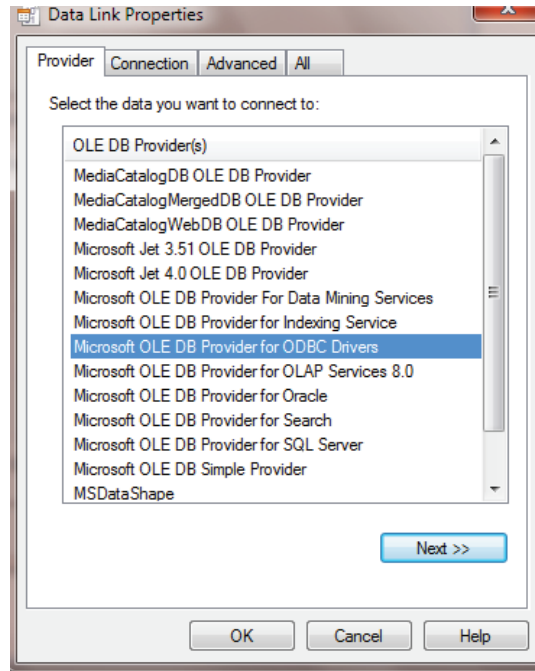
1. Activate the ADODC tools using the **Components** option from **Project Menu**.



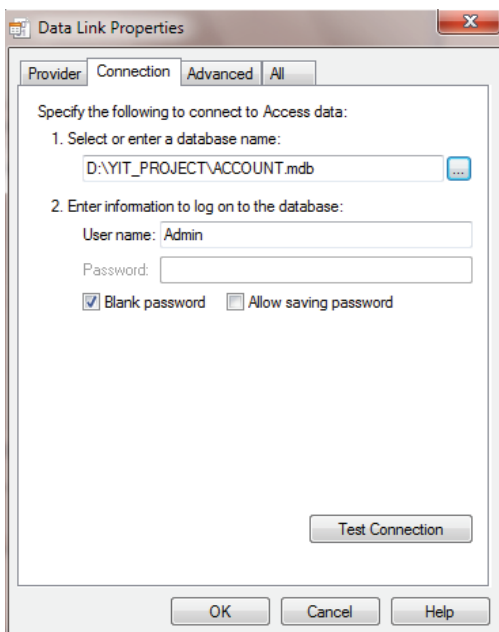
2. Now there is an object in the tool box. Place it on our form and set it's properties.
3. First of all **Right click** on it and select the **Properties Option**.
4. Now we a dialog box or wizard to follow steps.
5. In the Dialog Box set the connection string by click at **Build** button.



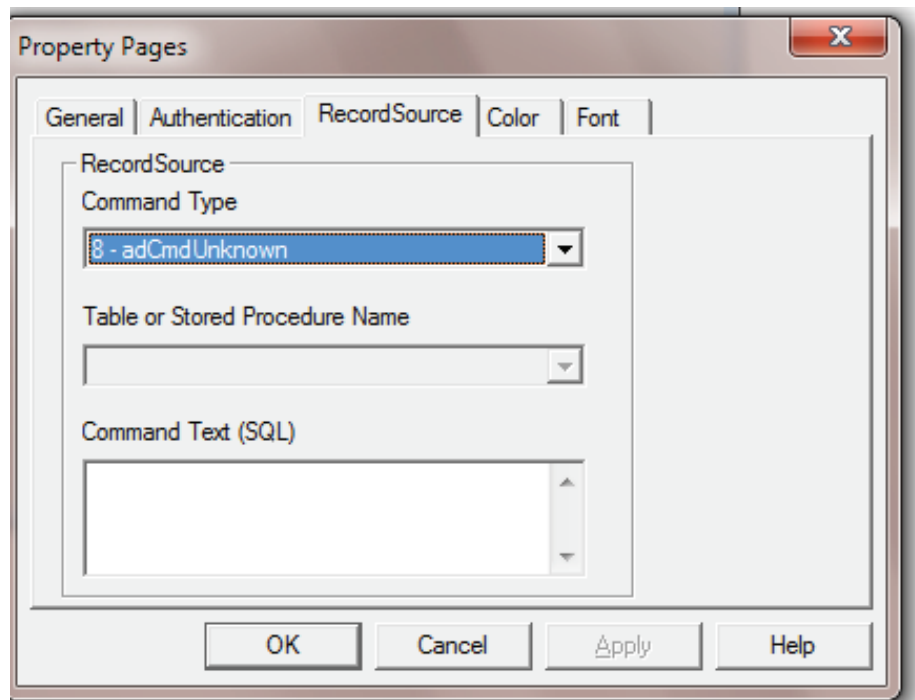
6. After click at **Build** button we have another dialog box where we have to select the provider. The provider is a communicator between database and our project. We have different kinds of provider according to different database platform.



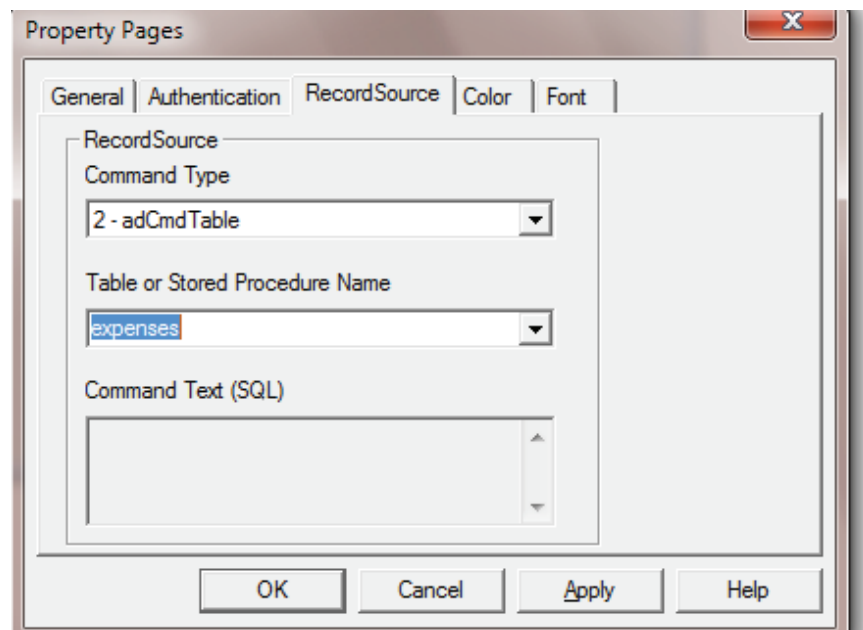
7. Select the **Provider** according to the database software then click at next button where we have another dialog box that accept the name of database with address.



8. Click at **Test Connection** button to confirm the connectivity than click at **OK** button to again transfer at ADODC dialog box to set the **Record Source**.



9. Select the **CMD TABLE** option from **Command Type** and select a table from list of tables then click at OK button to finalize the ADODC.



10. Now we have to connect the ADODC with our tools like text box for that these are two properties to set connection.
 - a. **Data Source** - In this property set the name of ADODC as a data source.
 - b. **Data Field** - Select the field from the list of fields available in the property.

10.4.1 Record Set

Visual Basic uses an object called RecordSet to hold your table. To declare such an object and to open the table, do this:

Dim rsMyRS As RecordSet

Set rsMyRS = dbMyDB.OpenRecordSet("MyTable", dbOpen Dynaset)

What happened there? Well, I declared a RecordSet object and used the Database object's OpenRecordSet method to open a table of type Dynaset. You can open a RecordSet in several modes. VB's online help file explains the different modes and what they are for. Dynaset gives you a RecordSet that you can add, delete and modify records in.

10.4.2 Record Pointer

The ADODC provide a cursor in the dynamic database which is available in the adodc at run time known as Record Pointer. The pointer helps to perform any type of task on our database. It is used to select the record for the further processing.

10.4.3 Method

ADODC provide set of method to work with it in our application. Every method have some unique feature to fulfill user's problem.

Move Next

The method is utilize to move the record pointer at next record at run time.

Syntax : Adodc1.Recordset.MoveNext

Move Previous

The method is utilize to move the record pointer at Previous record at run time.

Syntax : Adodc1.Recordset.MovePrevious

Move First

The method is utilize to move the record pointer at First record at run time.

Syntax : Adodc1.Recordset.MoveFirst

Move Last

The method is utilize to move the record pointer at Last record at run time.

Syntax : Adodc1.Recordset.MoveLast

Fields()

The method is utilize to retrieve the data from the fields of database. The method perform using index of fields. The index started from 0 to N-1 in the database.

Syntax : Adodc1.Recordset.Fields(index)

AddNew

To insert a new blank record at the last of database adodc provide AddNew method.

Syntax : Adodc1.Recordset.AddNew

Save

To save a record from form to the database at run time we have save method.

Syntax : Adodc1.Recordset.Save

Delete

To remove a record from database permanently adodc provide Delete method. For that we have to set the position of cursor.

Syntax : Adodc1.Recordset.Delete

State

To confirm the status of the record set before process it we have to use state method. It have option to set the state using adstateclose and adstateopen keywords.

Syntax : Adodc1.Recordset.state=adstateclose

Adodc1.Recordset.state=adstateopen

Close

To close the connection or empty the recordset we have close method.

Syntax : Adodc1.Recordset.close

10.4.4 Accessing records

Now that we have opened a table (referred to as RecordSet from now on) we want to access the records in it. The RecordSet object allows us to move in it by using the methods MoveFirst, MoveNext, MovePrevious, MoveLast (among others). I will use some of these to fill up a list box with the records of our RecordSet.

To get this example to work, make a database (with Access) called "MyDatabase.mdb" with the table "MyTable" in it. This table should have the fields "ID" of type "Counter" that you set to be the primary key, the field "Name" of type Text and a field "P hone" of type Text. Add some records to it. Put a list box on a form and call it "lstRecords".

Dim dbMyDB As Database

Dim rsMyRS As RecordSet

Private Sub Form_Load()

Set dbMyDB = OpenDatabase("MyDatabase.mdb")

Set rsMyRS = dbMyDB.OpenRecordSet("MyTable", dbOpenDynaset)

If Not rsMyRS.EOF Then rsMyRS.MoveFirst

Do While Not rsMyRS.EOF

lstRecords.AddItem rsMyRS!Name

lstRecords.ItemData(lstRecords.NewIndex) = rsMyRS!ID

rsMyRS.MoveNext

Loop

End Sub

This will make the list box fill up with your records when the form loads. I have introduced some new concepts with this example. We have all ready covered the first part where we open the table. The line that says *If Not rsMyRS.EOF Then rsMyRS.MoveFirst* tells the program to move to the first record in case there are any records at all. The **EOF** is a Boolean property that is true if the current record is the last. It is also true if there are no records in the RecordSet.

Then we make the program add the "Name" field of all records to the list box by adding the current records field "Name" and moving to the next record. You ask for a field of a RecordSet by putting a ! between the name of the RecordSet object and the name of the field. The while loop checks to see if there are more records to add.

10.5 Dynaset and Snapshot

The three types of recordsets are Tables, Dynasets, and Snapshots. All recordsets have records (rows) and fields (columns). The Professional Edition of Visual Basic lets you create object variables of type Dynaset, Snapshot, and Table. The Standard Edition supports Dynaset object variables but not Snapshot or Table object variables.

A table is a fundamental part of a Database and contains data about a particular subject. A Table object is a logical representation of a physical table.

To make a Snapshot or Dynaset, use the CreateSnapshot or CreateDynaset method on a Database or any recordset. A Snapshot is a static, read-only picture of a set of records that you can use to find data or generate reports. The records in a Snapshot cannot be updated (or modified), whereas records in a Dynaset can be updated.

The move methods (MoveFirst, MoveLast, MoveNext, and MovePrevious) apply to all three types of recordsets (Dynasets, Snapshots, and Tables).

The find methods (FindFirst, FindLast, FindNext, and FindPrevious) apply to Dynaset objects and Snapshot objects, but not to Table objects. The Seek method applies only to Table objects.

For intensive searches, you may want to use both Table and Dynaset objects on the same base table. You can use the Seek method on the Table objects and the find methods on any open Dynasets.

Visual Basic data controls always use Dynasets. Data controls don't use Snapshot objects or Table objects.

Dynaset objects are a set of record pointers to those records which existed in the base table in the Database at the time the Dynaset was created. Your Dynaset also adds pointers to any new records which you add to the Dynaset and deletes pointers of deleted records.

If you add a record to a base table, the record does not immediately appear in any currently existing Dynaset based on that table. You would need to re-create the Dynaset to see a new record that was added to the base table after the Dynaset was created. However, if you add a new record to a Dynaset, the record appears immediately in both the Dynaset and the base table. Deleting a record is reflected in a similar way.

10.6 Summary

We have two different type of database tools ADODC which is static tool that have pre define rules and syntax and ADODB that provide permission to integrate our query with it. The ADODC and ADODB both are bounded control, they perform when ever we have to connect it with other controls. Adodc and Adodb have set of properties and method to handle operation belongs to user's problem.

10.7 Self Assessment Questions

1. Define the set of steps to control the ADODC.
2. How can we navigate the records? Define the procedure to add a record.
3. Define the record set and their properties.
4. Define the differences and similarities between shopshot set and dynaset.

Unit - 11 : Connectivity with database and Database operation

Structure of Unit:

- 11.0 Objective
- 11.1 Introduction
- 11.2 ADODB
 - 11.2.1 Implementation
 - 11.2.2 Operations using Connection string
- 11.3 ODBC and DSN
- 11.4 RDO
 - 11.4.1 Steps
 - 11.4.2 Properties
 - 11.4.3 Methods
- 11.5 Search, Update And Delete Operation
- 11.6 Summary
- 11.7 SelfAssessment Questions

11.0 Objective

The main objective of this unit is that enable our application to run on standalone and network. This concept also helpful to create distributed database application where all the data of our application is at centralized location.

In this unit we have the connection string and various operation on database.

11.1 Introduction

In this unit we provide the details about the environment of an network environment and their corresponding tool that enable it in our application. In this unit we can perform connection using the dynamic database control known as ADODB. The Unit also explain the Remote Data Control and ODBC.

11.2 ADODB

The run time object of ADODC perform as a ABODB. That means it is run time object of adodc which allow us to perform dynamic database connectivity. Adodb is not a physical tool it is a runtime dynamic object which is design and then destroy at run time. To use it we have to perform set of code using these set of steps.

11.2.1 Implementation

- * We have to activate the ADODC tool using **Component Option of Project Menu.**
- * Place the ADODC tool at our form then write set of code.
- * First of all we have to create two run time object of ADODB.
- * **Connection Object** - Used to holds the connection string for the database connectivity.
- * **RecordSet Object** - Used to store the data in the tabular format at run time to perform various task or manipulation.

Syntax :

```
Dim cn as new ADODB.Connection
```

```
Dim rs as new ADODB.Recordset
```

- * Now we have to open the connection string using the OPEN() of adodb. The method required two arguments the name of provider and data source. The connection open at load event of form.

Syntax :

```
Private Sub Form1_Load()
```

```
cn.Open"Provider=Microsoft.Jet.OLEDB.4.0;DataSource=D:\ACC.mdb"
```

```
End Sub
```

- * After open the connection we have to retrieve data from database in the object of record set. To perform this task there is also OPEN() method with 4 arguments. These are-
- * Open method perform by accepting query by which we can get our required data.
- * After defining the query in the open method we have to define the source of database for that we can use connection object.
- * Now we have to set the mode of record set using adOpenDynamic.
- * At last we have to set the cursor mode to navigate it using adLockPessimistic.

11.2.2 Operation Using Connection String and Object with SQL Query

```
Private Sub Form1_Load()
```

```
cn.Open"Provider=Microsoft.Jet.OLEDB.4.0;DataSource=D:\ACC.mdb"
```

```
rs.open"select* From expenses", cn, adOpenDynamic, adLockPessimistic
```

```
End Sub
```

- * To retrieve the data from the object of Record Set use fields() method.

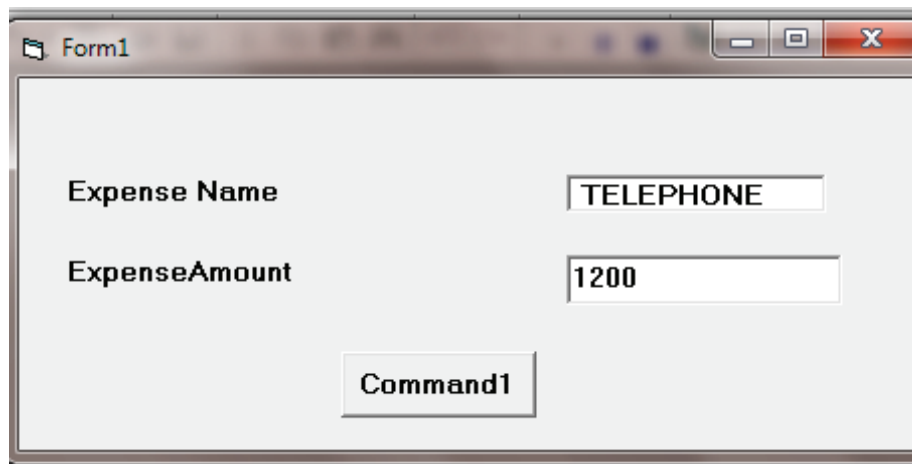
Syntax :

```
Private Sub Command1_Click()
```

```
Text1.text = rs.fields(0)
```

```
Text2.text = rs.fields(2)
```

```
End Sub
```

The image shows a screenshot of a Windows application window titled "Form1". Inside the window, there are two text boxes. The first text box is labeled "Expense Name" and contains the text "TELEPHONE". The second text box is labeled "ExpenseAmount" and contains the text "1200". Below these two text boxes, there is a button labeled "Command1". The window has a standard Windows title bar with minimize, maximize, and close buttons.

- * **To retrieve the data from next record we have to use.**

Syntax :

```
Private Sub Command2_Click()
```

```
rs.movenext
```

```
if not rs.EOF
```

```
Text1.text = rs.fields(0)
```

```
Text2.text = rs.fields(2)
```

```
Else
```

```
MsgBox("Record Not Found")
```

```
Command2.enabled=false
```

```
Endif
```

```
End Sub
```

- * **To retrieve the data from previous record we have to use.**

Syntax :

```
Private Sub Command3_Click()
```

```
rs.moveprevious
```

```
if not rs.BOF
```

```
Text1.text = rs.fields(0)
```

```
Text2.text = rs.fields(2)
```

```
Else
```

```
MsgBox("Record Not Found")
```

```
Command3.enabled=false
```

```
Endif
```

```
End Sub
```

- * **To Search the record from database using Select Query -**

Syntax :

```
Private Sub Command4_Click()
```

```
If rs.state=adstateopen then
```

```
rs.close
```

```
endif
```

```
rs.open"select* From expenses where ename='"+text1.text+"", cn, adOpenDynamic,  
adLockPessemistic
```



```

if rs.count>0 then
    text2.text=rs.fields(1)
else
    MsgBox("Record Not Found")
endif
End Sub

```

Note :-

- * Before perform searching we have to close the record set then reopen it with new requirements.
- * To find out the record we have to use select query with where condition.
- * To Insert the record in the database using Insert Query-

Syntax :

```

Private Sub Command5_Click()
If rs.state=adstateopen then
rs.close
endif
rs.open"insert into expenses values('"+text1.text+"','"+text2.text+"")", cn,
adOpenDynamic, adLockPessemistic
MsgBox("One Record Saved")
End Sub

```

- * **To Update the existing record in the database using Update Query -**

Syntax :

```

Private Sub Command6_Click()
If rs.state=adstateopen then
rs.close
endif
rs.open"update expenses set eamont='"+text2.text+" where ename='"+text1.text+"'",
cn, adOpenDynamic, adLockPessemistic
if rs.count>0 then
    MsgBox("One Record Updated")
Else
    MsgBox("Record Not Found")
endif
End Sub

```

endif

End Sub

EOF()

It stands for End of File and utilize to confirm that the current record is end of file or not. If record pointer is at last then return TRUE otherwise FALSE.

BOF()

It stands for Beginning of File and utilize to confirm that the current record is Beginning of file or not. If record pointer is at First position then return TRUE otherwise FALSE.

Syntax : Adodc1.Recordset.EOF

11.3 ODBC and DSN

It is Open Database Connectivity which have permission to set connection between our form and database from Oracle and SQL. It also enable us to set the connectivity using other database packages like Access, Fox- pro, etc.

To implementation of RDO we have to utilize the concept of DSN.

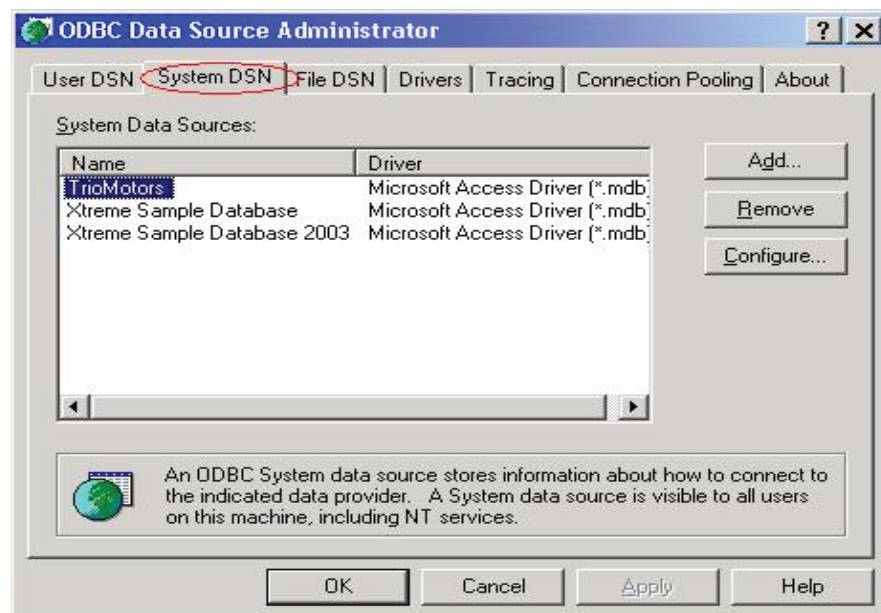
DSN (Data Source Name)

It stands for Data Source Name which is an integrated module outside the VB. to assign a short cut or alias for the database and provider by removing the connection string. To design the DSN we have to use set of steps. These are as follows :

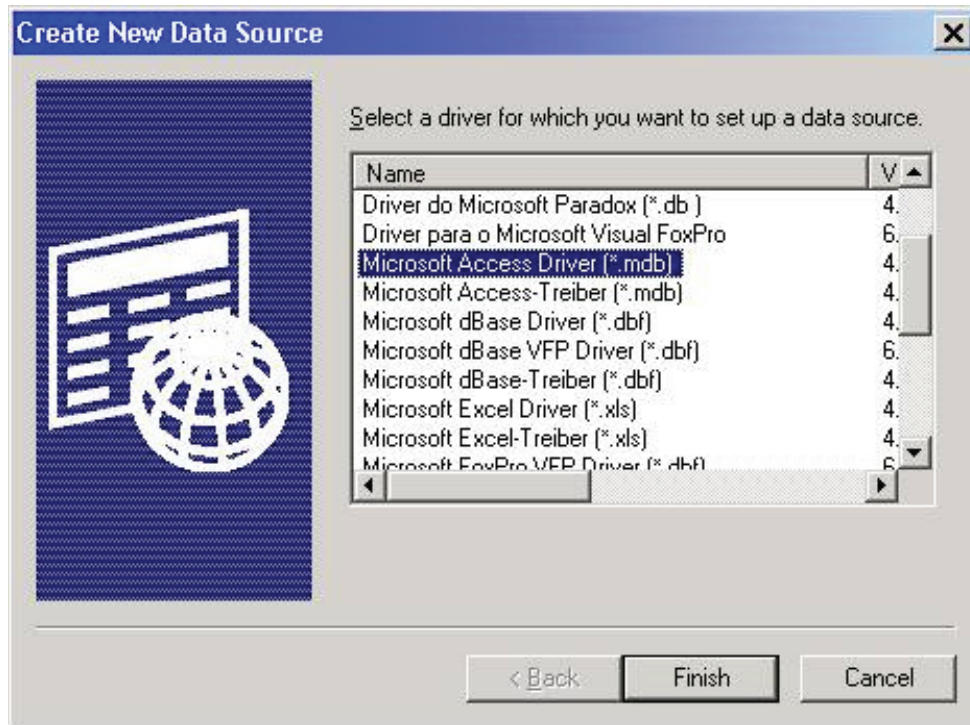
Setting Up an ODBC Data Source

Follow the steps below to set up an ODBC Data Source (this process is also called "setting up a DSN", where "DSN" stands for "Data Source Name"). These steps assume Windows 2000 for the operating system. On other versions of Windows, some steps may vary slightly.

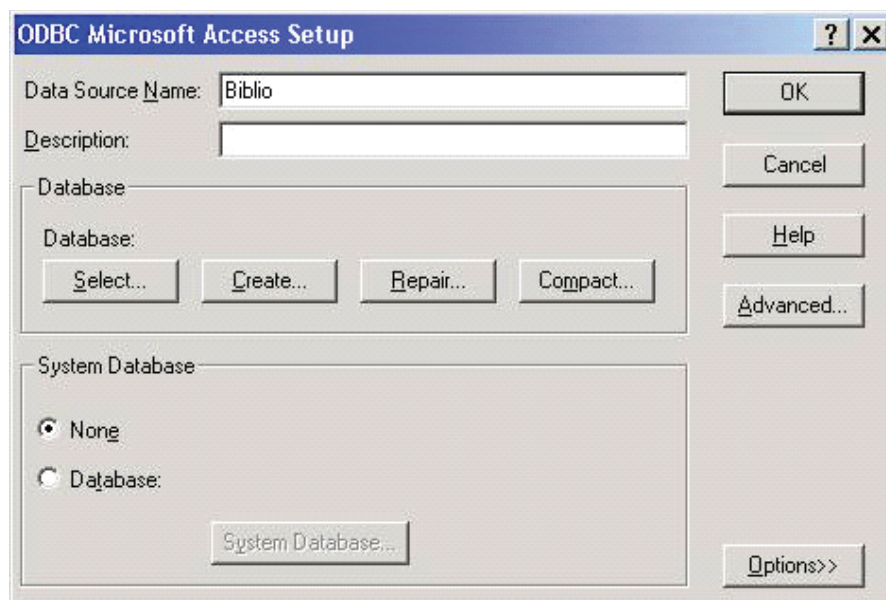
▫ Via Windows **Control Panel**, double-click on **Administrative Tools**, then **Data Sources (ODBC)**. The **ODBC Data Source Administrator** screen is displayed, as shown below. Click on the **System DSN** tab.



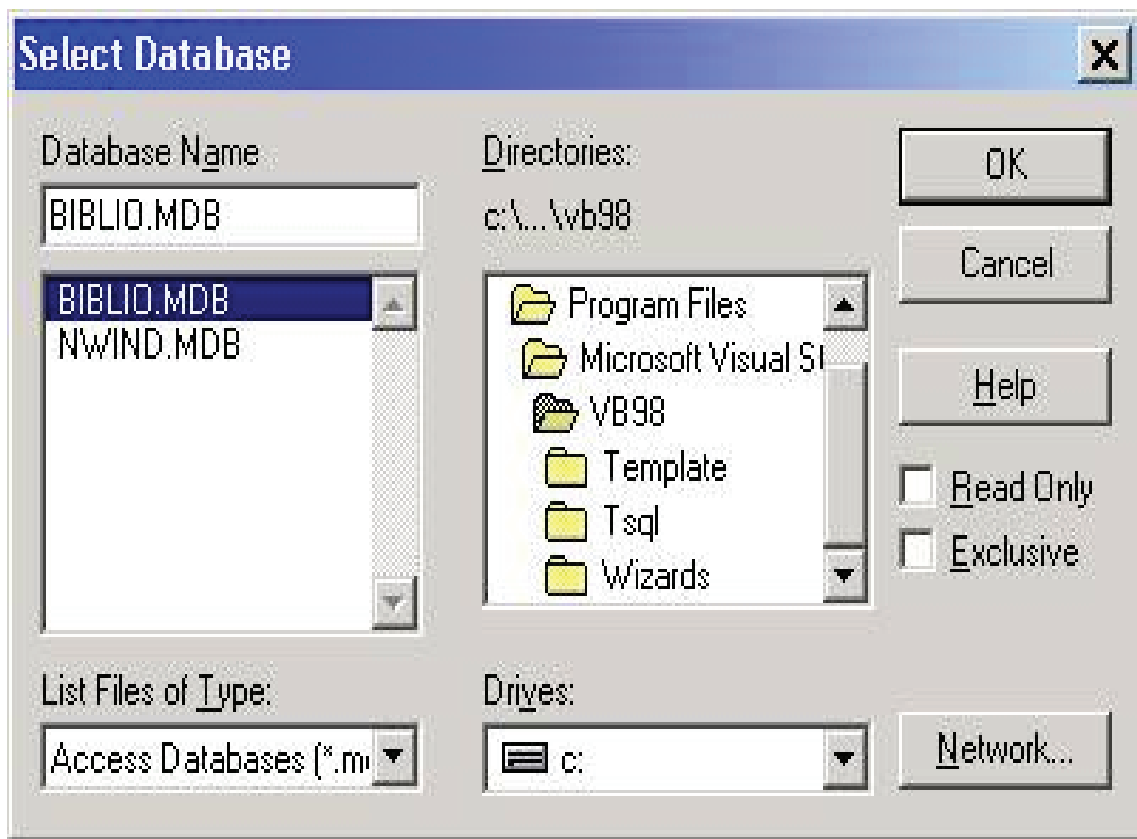
- Click the Add button. The **Create New Data Source** dialog box will appear. Select **Microsoft Access Driver (*.mdb)** from the list and click the **Finish** button.



- The **ODBC Microsoft Access Setup** dialog box will appear. For **Data Source Name**, type **Biblio**. If desired, you can type an entry for **Description**, but this is not required.

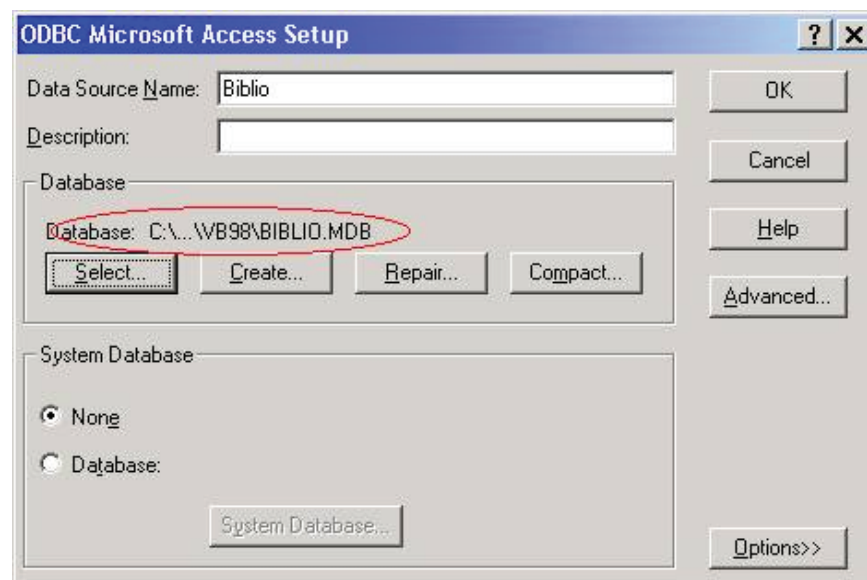


- Click the **Select** button. The **Select Database** dialog box appears. On a default installation of VB6 or Visual Studio 6, the **BIBLIO.MDB** sample database should reside in the folder **C:\Program Files\Microsoft Visual Studio\VB98**. Navigate to that folder, select **BIBLIO.MDB** from the file list, and click **OK**.

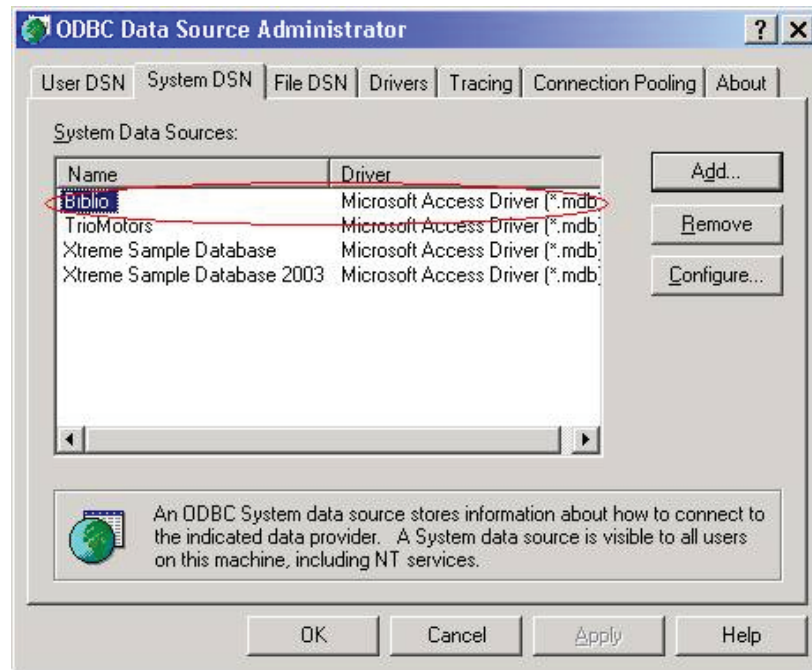


Note: If VB was installed in a different location on your system, navigate to the appropriate folder. If you do not have the **BIBLIO.MDB** sample database file on your system at all, you can download it here. In that case, copy the file to the folder of your choice, and navigate to that folder to select the database for this step.

- When you are returned to the **ODBC Microsoft Access Setup** screen, the database you selected should be reflected as shown below. Click **OK** to dismiss this screen.



- When you are returned to the **ODBC Data Source Administrator** screen, the new **DSN** should appear as shown below. Click **OK** to dismiss this screen.



At this point, the Biblio database is ready to be used with RDO in the sample application.

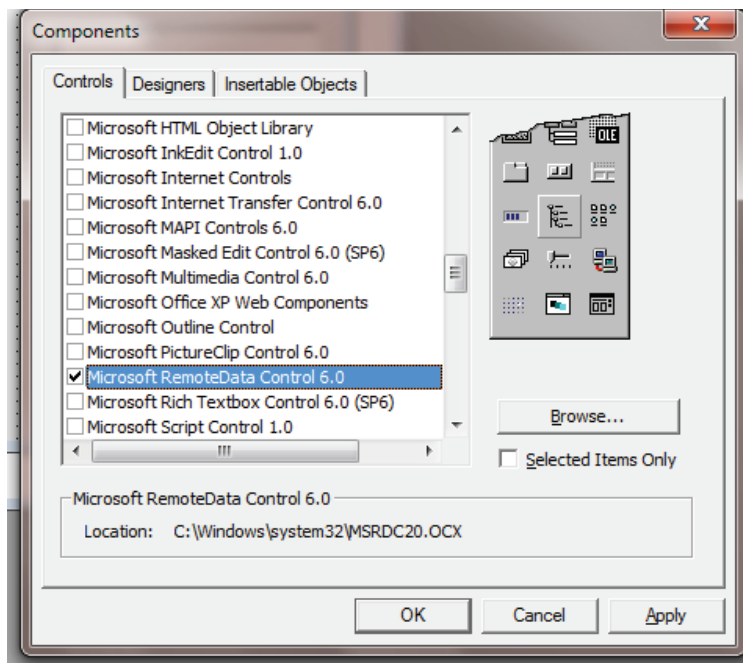
11.4 RDO (Remote Data Object)

The RDO stands for Remote Data Object. It is a bounded control to retrieve the data from the database. The RDO perform using a mediator connector known as ODBC.

11.4.1 Steps

After the designing of DSN now we have to set the connectivity with RDO tool using set of procedure.

1. Activate the RDO tool using components option of project menu by using the Microsoft **Remote Data Control 6.0** service.



2. Now have a new object at our form which we have to place at our form.
3. The RDO tool have some properties which we have to set for the connection.

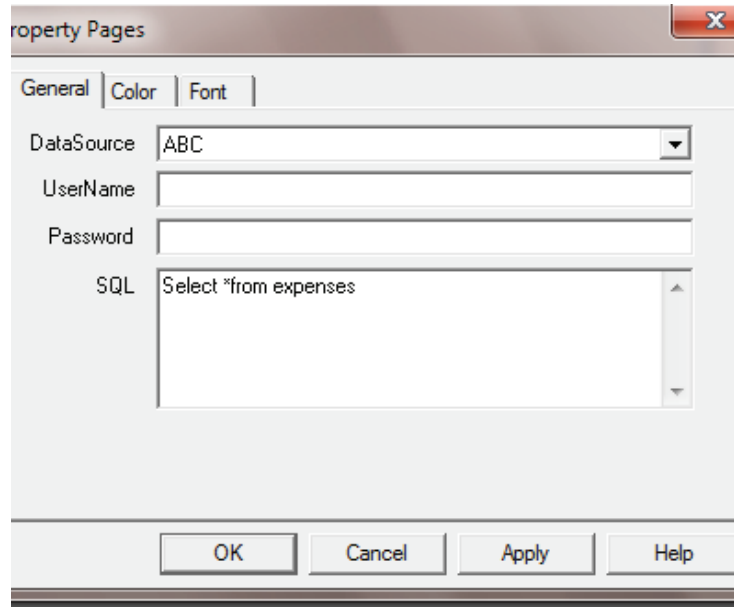
11.4.2 Properties

1. Data Source Name-

In this property we have to select the name of DSN which is available in the list of available DSN which we create by control panel.

2. SQL-

Now right click at the tool and select the property option. We have dialog box where we have to define the query which will execute to retrieve our data with user name and password, if required.



4. After decide the properties of RDO we have to set the properties of Text Box.

1. **Data Source** - Select the RDO tools.
2. **Data Fields** - Select the appropriate field from the list of fields.

11.5 Search, Update And Delete Operation

Searching the RecordSet

You might have wondered why the value of the field "ID" in the list box's ItemData property. This is because that we would know the primary key for all the records in order to search for a record.

Put a text box somewhere on the form and call it "txtPhone". Then copy the following code to the project.

```
Private Sub lstRecords_Click()  
rsMyRS.FindFirst "ID=" & Str(lstRecords.ItemData(lstRecords.ListIndex))  
txtPhone.Text = rsMyRS!Phone  
End Sub
```

This will display the phone number of the selected person when clicking in the list box. It uses the FindFirst method of the RecordSet object. This takes a string parameter that is like what is after WHERE in a SQL expression. You state the field that you want to search in (here "ID"), then the evaluation criteria (here "=") and last the value to search for (here the ItemData of the selected item in the list box).

So what we did was to search for the record with the "ID" field value that was the same as the ItemData property of the selected item in the list box. Then we show the value of the "Phone" field in the text box.

Updating the Database

You will probably want to be able to update some value of some field when doing database programming. This is done with Edit and Update. We will try to change the value of the "Phone" field by editing the text in the text box and clicking a button.

Put a command button on the form and name it "cmdUpdate". Then copy the following code to the project.

```
Private Sub cmdUpdate_Click()  
rsMyRS.Edit  
rsMyRS!Phone = txtPhone.Text  
rsMyRS.Update  
End Sub
```

Could it be that simple? Yes. This changes the phonenumber of our selected person. Or to put it technically: This changes the value of the "Phone" field of our current record. Imagine the current record being a set of boxes, with a field in each box. The Edit method takes the lid off all of the boxes and Update puts them back on. When we write rsMyRS!Phone = txtPhone.Text we replace the content of the "Phone" box with the content in the text box.

Deleting and Adding records

Deleting

Deleting records couldn't be simpler. To delete the current record you just invoke the Delete method of the RecordSet object. We will put this feature in our little project. Make one more command button named "cmdDelete" and the following code will do the work of deleting our currently selected person.

```
Private Sub cmdDelete_Click()  
rsMyRS.Delete  
lstRecords.RemoveItem lstRecords.ListIndex  
End Sub
```

I won't even bother to explain that in greater detail =). The first statement deletes the record and the second removes the list box entry.

Adding

Adding records is much like updateing, except you use AddNew instead of Edit. Let's add one more command button to our application. Let's call it...errh...let me see...yea! "cmdNew" =). Here is the code that adds a new record.

```
Private Sub cmdNew_Click()  
rsMyRS.AddNew  
rsMyRS!Name = "A New Person"  
lstRecords.AddItem rsMyRS!Name
```

lstRecords.ItemData(lstRecords.NewIndex) = rsMyRS!ID

rsMyRS!Phone = "Person's Phone Number"

rsMyRS.Update

End Sub

11.6 Summary

The unit have the set of control and method that can helps to retrieve the data from database and also provide representation that will use the sql query with database operations. When we create the object of record set than it accept the query belong to our requirement and provide the results. The sql query also accept by RDO control directly to retriive our data from database. The Adodb object also enable use to use the query for insert, delete and search to handle various task with writing typical set of code.

11.7 Self Assessment Questions

1. Write down the procedure to insert a record in the database.
2. What do you mean by ODBC? Define the steps to set the RDO connection.
3. What is the meaning of DSN? How can we access a record using the connection string.

Unit - 12 : Object Linking and Embedding (OLE)

Structure of Unit:

- 12.0 Objective
- 12.1 Introduction
- 12.2 Linking Vs. Embedding
- 12.3 Autosizing an OLE Control
- 12.4 OLE Container Control
- 12.5 Pop-Up Menus At Design Time
- 12.6 OLE Control At Runtime
- 12.7 New OLE Controls At Runtime
- 12.8 Example of OLE using Excel Sheet
- 12.9 Summary
- 12.10 Self Assessment Questions

12.0 Objective

After completing of this chapter one can understand about OLE, and linking vs. Embedding also the chapter Summarizing, Autosizing OLE container, design and Run Time OLE.

12.1 Introduction

Using OLE you can give the users of your program direct access to OLE server programs like Microsoft Word or Excel. In fact, you can integrate all kinds of programs together using OLE, giving your program the power of database, spreadsheet, word processor, and even graphics programs all wrapped into one.

Visual Basic lets you do this with the OLE control. This control can display OLE objects, and those objects appear as mini-versions of the programs connected to them. For example, if you display an Excel spreadsheet in an OLE control, the control displays what looks like a small version of Excel right there in your program. The program that creates the object displayed in

the OLE control is an OLE server, and your program, which displays the OLE object, is called an OLE container. In fact, the proper name for the OLE control is the OLE container control.

You can use the OLE object in the OLE control just as you would in the program that created it; for example, you can work with an Excel spreadsheet in an OLE control just as if it was open in Excel itself. How does that work?

There are two primary ways of working with the OLE objects in an OLE control: opening them and editing them in place. When you open them, the OLE server application is launched in its own window and the OLE object appears in that application. When you want to save your changes to the OLE object in the OLE control, you use the server's Update item in the File menu.

When you edit an OLE object in place, the server application is not launched in its own window; instead, the object becomes active in the OLE control itself and may be edited directly. The OLE container program's menu system is taken over by the OLE server-and you may be startled to see Microsoft Word's or Excel's menu system in your program's menu bar. To close an OLE object that is open for in-place editing, you click the form outside the object. As an example, the Microsoft Excel spreadsheet in Figure 12.1 is open for in-place editing.

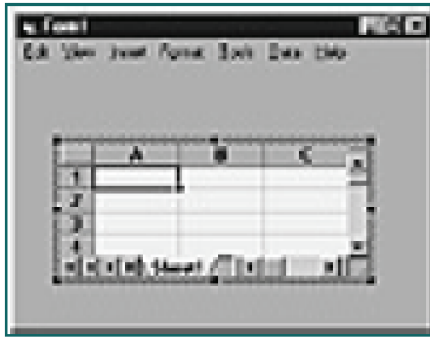


Figure 12.1 Opening an Excel spreadsheet in an OLE control for in-place editing.

When the OLE object in an OLE control is closed, it appears in its inactive state, as shown in Figure 12.2.

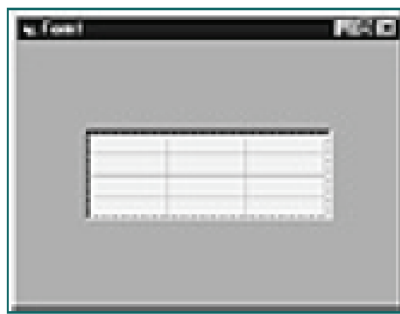


Figure 12.2 An inactive OLE object.

OLE actions are called verbs; for example, opening an OLE object is accomplished with the `VbOLEOpen` verb, and editing it in place is accomplished with the `VbOLEInPlaceActivate` verb. We'll see how to handle OLE verbs in this chapter when we use the OLE control's `DoVerb` method.

What other methods does the OLE control support? Those methods and what they do appear in Table 12.1. You can also use the OLE control's `Action` method to invoke the methods in Table 12.1, and the values for this property also appear in Table 12.1. When you use the `Action` property, the control often uses other properties of the control, such as the `SourceDoc` property, to find the data it needs to perform the requested operation. Note, however, that the `Action` property is considered obsolete, and we'll use the OLE methods instead.

Table 12.1 OLE methods.

Method	Action	Value	Meaning
CreateEmbed	0		Creates embedded object
CreateLink	1		Creates linked object from the contents of a file
Copy	4		Copies the object to the system Clipboard
Paste	5		Copies data from the system Clipboard to an OLE container control
Update	6		Retrieves the current data from the application that supplied the object and displays that data as a picture in the OLE

		container control
DoVerb	7	Opens an object for an operation, such as editing
Close	9	Closes an object and terminates the connection to the application that provided the object
Delete	10	Deletes the specified object and frees the memory associated with it
SaveToFile	11	Saves an object to a data file
ReadFromFile	12	Loads an object that was saved to a data file
InsertObjDlg	14	Displays the Insert Object dialog box
PasteSpecialDlg	15	Displays the Paste Special dialog box
FetchVerbs	17	Updates the list of verbs an object supports
SaveToOle1File	18	Saves an object to the OLE version 1 file format

12.2 Linking Vs. Embedding

What's the difference between linking and embedding OLE objects in the OLE control? The main difference has to do with where the object's data (such as the data in a spreadsheet) is stored. Data associated with a linked object is manipulated by the OLE server application that created it and is stored outside an OLE container control. Data associated with an embedded object is contained in an OLE container control, and that data can be saved with your Visual Basic application. When you embed an object into an OLE control, that control stores the name

of the application that supplied the object, along with its data. The `OLEType` property of the control is set to `Embedded (1)`. When you link an object to an OLE control, that control stores the name of the application that supplied the object and a reference to the data. The `OLEType` property of the control is set to `Linked (0)`.

Note that because an embedded object's data is stored in your program, you're responsible for storing and reading that data if it changes. To do that, we'll use the `SaveToFile` and `ReadFromFile` methods in this chapter. You can also specify the types of objects that an OLE control can take by setting its `OLETypeAllowed` property to `Linked (0)`, `Embedded (1)`, or `Either (2)`.

To embed objects into an OLE control at runtime, you usually use the `Insert Object` object of the container program's `Object` menu. To link objects to an OLE control, you usually copy the object in the server application and use the `Paste special` menu object in the OLE container.

12.3 Autosizing an OLE Control

After you've added an OLE control to a form, you can specify how you want that control to handle OLE object insertions: for example, should the control resize itself when an OLE object is inserted in it or not? You can specify how the control should handle object insertions with the `SizeMode` property. Here are the possible values for the `SizeMode` property:

- * `vbOLESizeClip-0` (the default); `Clip`. The object is displayed in actual size. If the object is larger than the OLE control, its image is clipped by the control's borders.

- * `vbOLESizeStretch-1`; `Stretch`. The object's image is sized to fill the OLE control. (Note that the image may not maintain the original proportions of the object.)

- * `vbOLESizeAutoSize-2`; Autosize. The OLE control is resized to display the entire object.
- * `vbOLESizeZoom-3`; Zoom. The object is resized to fill the OLE container control as much as possible while still maintaining the original proportions of the object.

12.4 OLE Container Control

There are two ways of displaying an OLE object in an OLE control: displaying that object's content and displaying it as an icon. Throughout this chapter, we'll use the content display, which is the default, but you can also display it in icon form using the OLE control's `DisplayType` property. Here are the possible settings for `DisplayType`:

* `vbOLEDisplayContent-0` (the default); Content. When the OLE container control contains an object, the object's data is displayed in the control.

* `vbOLEDisplayIcon-1`; Icon. When the OLE container control contains an object, the object's icon is displayed in the control. For example, we've set the `DisplayType` property of the OLE control in Figure 12.3 to `vbOLEDisplayIcon` so it displays the icon of its OLE object, which in this case is an Excel object. Note that you cannot open an iconic OLE object for in-place editing, but you can open it for editing in the OLE server program (in other words, when you double-click the object, the OLE server application opens in its own window).



Figure 12.3 Displaying an OLE object in iconic form.

12.5 Pop-Up Menus At Design Time

When you add an OLE control to a form, the Insert Object dialog box appears, and you can insert an object into the control at that time. However, if you don't want to insert an object right away, you can use the control's pop-up menu later. To open the OLE control's pop-up menu, just right-click the control at design time. Here are the primary items in the menu that appears:

- o Insert Object-Open the Insert Object dialog box.
- o Paste Special-Link to Clipboard object.
- o Delete-Delete embedded object.
- o Create Link-Create a link to the document in the SourceDoc property.
- o Create Embedded Object-Creates an embedded object using the Class or SourceDoc property.

Note that you can link to or embed a document in the OLE control if you've set the control's `SourceDoc` property. You can also create a new embedded object if you've set the control's `Class` property; classes specify what code components (formerly called OLE automation servers) are available.

12.6 OLE Control At Runtime

The Testing Department is on the phone again. It's fine that you've been able to embed an OLE object in an OLE control at design time, but don't you think it would be great if you could let users embed their own OLE objects? Maybe, you say. To let users insert their own OLE objects into an OLE control, you can display the same Insert Object dialog box that appears at design time when you created the OLE control, and you use the control's `InsertObjDlg` method to do that (this method takes no parameters). Let's see an example. Add an OLE control, `OLE1`, to a form now, and click Cancel when the Insert Object dialog box appears. Set the OLE control's `SizeMode` property to `VbOLESizeAutoSize` (note that when you set an OLE control's `SizeMode` property to `VbOLESizeAutoSize`, the OLE server sets the size of the OLE object, so the size of the OLE control, `OLE1`, will probably change when the object is inserted). Finally, add an Insert menu to the form with one item in it: Insert Object. When users click the Insert Object menu item, they want to insert an OLE object in `OLE1`, and we can let them create and embed a new object, embed an existing file, or embed a link to an existing file with the `InsertObjDlg` method. We do so like this:

```
Private Sub InsertObject_Click()  
OLE1.InsertObjDlg
```

```
...
```

```
End Sub
```

When we execute this method, `InsertObjDlg`, the program displays the Insert Object dialog box, which appears in Figure 12.4. Using this dialog box, the user can embed new objects, existing files, or link to existing files. In fact, that's all the code we need-when the `InsertObjDlg` method finishes, it inserts the new OLE object in the OLE control.

We can do one more thing here-we can check to make sure the OLE insertion operation was completed successfully. To do that, we'll use the OLE control's `OLEType` property to check if there's an OLE object in `OLE1`. This property can take the values `vbOLELinked`, `vbOLEEmbedded`, or `vbOLENone`. Here, we check that property, and if it's set to `vbOLENone`, we indicate to the user that there was an error:

```
Private Sub InsertObject_Click()  
OLE1.InsertObjDlg  
If OLE1.OLEType = vbOLENone Then  
MsgBox "OLE operation failed."  
End If  
End Sub
```

Using this code allows us to insert OLE objects like the Excel spreadsheet you see in our OLE control in Figure 12.4. Here, when the object is first inserted, it's opened for in-place editing by default.

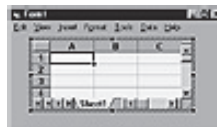


Figure 12.4 Inserting an OLE object into a program.

There's a problem here, however. Now that we've inserted a new OLE object and activated it, how do we deactivate it? There are no Exit items in the Excel menu system that's taken over our program's menu system. Usually you deactivate an OLE object by clicking the form around the OLE control, and we'll see how that works in the next topic.

The code for this example so far, `insertole.frm`

```

VERSION 6.00
Begin VB.Form Form1
Caption = "Form1"
ClientHeight = 2115
ClientLeft = 165
ClientTop = 735
ClientWidth = 4680
LinkTopic = "Form1"
ScaleHeight = 2115
ScaleWidth = 4680
StartupPosition = 3 'Windows Default
Begin VB.OLE OLE1
Height = 1095
Left = 840
SizeMode = 2 'AutoSize
TabIndex = 0
Top = 360
Width = 3015
End
Begin VB.Menu File
Caption = "File"
End
Begin VB.Menu Insert
Caption = "Insert"
Begin VB.Menu InsertObject
Caption = "Insert object"
End
Begin VB.Menu PasteSpecial
Caption = "Paste special"
End
End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Private Sub Form_Click()
OLE1.AppIsRunning = False
End Sub
Private Sub InsertObject_Click()
OLE1.InsertObjDlg
If OLE1.OLEType = vbOLENone Then
MsgBox "OLE operation failed."
End If
End Sub

```

```

Private Sub PasteSpecial_Click()
If OLE1.PasteOK Then
OLE1.PasteSpecialDlg
End If
If OLE1.OLEType = vbOLENone Then
MsgBox "OLE operation failed."
End If
End Sub

```

12.7 New OLE Controls At Runtime

The olemultiple example we've developed in the previous two topics is designed to handle multiple OLE objects, but so far we only can handle two such objects because there are only two OLE controls in the program. OLE container programs, however, may be expected to handle more than two OLE objects, so we'll take a look at how to load additional OLE controls as required at runtime.

Let's see an example. We'll modify the olemultiple example now to let the user load in multiple OLE controls as required. To do that, add a new menu item, Create New OLE Control, to that program's File menu. When the user selects this item, we start by incrementing the total number of OLE controls, which we've stored in the variable `intTotalOLEControls`:

```

Private Sub CreateNewOLEControl_Click()
intTotalOLEControls = intTotalOLEControls + 1

```

...

Next, we load a new OLE control, adding it to our array of OLE controls, `OLEControls`:

```

Private Sub CreateNewOLEControl_Click()
intTotalOLEControls = intTotalOLEControls + 1
Load OLEControls(intTotalOLEControls - 1)

```

...

Now we position the new OLE control at upper left in the program, make it visible, and let the user insert an OLE object into it with the `InsertObjDlg` method:

```

Private Sub CreateNewOLEControl_Click()
intTotalOLEControls = intTotalOLEControls + 1
Load OLEControls(intTotalOLEControls - 1)
OLEControls(intTotalOLEControls - 1).Move 0, 0
OLEControls(intTotalOLEControls - 1).Visible = True
OLEControls(intTotalOLEControls - 1).InsertObjDlg

```

...

Finally, we can check if the object insertion operation was completed successfully and inform the user if not:

```

Private Sub CreateNewOLEControl_Click()
intTotalOLEControls = intTotalOLEControls + 1
Load OLEControls(intTotalOLEControls - 1)
OLEControls(intTotalOLEControls - 1).Move 0, 0
OLEControls(intTotalOLEControls - 1).Visible = True
OLEControls(intTotalOLEControls - 1).InsertObjDlg

```

```

If OLEControls(intTotalOLEControls - 1).OLEType = None Then
MsgBox "OLE operation failed."
End If
End Sub

```

And that's it-now we let the user add OLE controls as needed, using the Create New OLE Control menu item, as you can see in Figure 12.5. As you can also see in Figure 12.5, however, the placement of our new OLE object is less than optimal. Ideally, of course, the user can specify where in a container program the new OLE object should go, and we'll take a look at that in the next topic, where we let the user drag OLE controls in a form.



Figure 12.5 Loading a new OLE control at runtime.

The code for this example, olemultiple.frm version 2, appears in Listing 12.2

Listing 12.2 olemultiple.frm version 2

```

VERSION 6.00
Begin VB.Form Form1
Caption = "Form1"
ClientHeight = 3015
ClientLeft = 165
ClientTop = 735
ClientWidth = 4680
LinkTopic = "Form1"
ScaleHeight = 3015
ScaleWidth = 4680
StartupPosition = 3 'Windows Default
Begin VB.OLE OLEControls
Height = 1215
Index = 1
Left = 840
SizeMode = 2 'AutoSize
TabIndex = 1
Top = 1680
Width = 3015
End
Begin VB.OLE OLEControls

```



```

Height = 1215
Index = 0
Left = 840
SizeMode = 2 'AutoSize
TabIndex = 0
Top = 240
Width = 3015
End
Begin VB.Menu File
Caption = "File"
Begin VB.Menu ActivateObject
Caption = "Activate object"
End
Begin VB.Menu CreateNewOLEControl
Caption = "Create new OLE control"
End
End
Begin VB.Menu Insert
Caption = "Insert"
Begin VB.Menu InsertObject
Caption = "Insert object"
End
End
Begin VB.Menu PasteSpecial
Caption = "Paste special"
End
End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Dim intTotalOLEControls As Integer
Dim intXOffset, intYOffset As Integer
Private Sub ActivateObject_Click()

```

```

IfTypeOfActiveControl Is OLE Then
ActiveControl.DoVerb 0
End If
End Sub

Private Sub CreateNewOLEControl_Click()
intTotalOLEControls = intTotalOLEControls + 1
Load OLEControls(intTotalOLEControls - 1)
OLEControls(intTotalOLEControls - 1).Move 0, 0
OLEControls(intTotalOLEControls - 1).Visible = True
OLEControls(intTotalOLEControls - 1).InsertObjDlg
If OLEControls(intTotalOLEControls - 1).OLEType = None Then
MsgBox "OLE operation failed."
End If
End Sub

Private Sub Form_Load()
intTotalOLEControls = 2
End Sub

Private Sub InsertObject_Click()
IfTypeOfActiveControl Is OLE Then
ActiveControl.InsertObjDlg
If ActiveControl.OLEType = None Then
MsgBox "OLE operation failed."
End If
End If
End

```

12.8 Example of OLE using Excel Sheet

Classes and components provide a granular way to reuse code. A class of applications referred to as application servers represents a way for a client application to reuse a whole server application, which is often a significant improvement over reusing some classes or components.

Microsoft has a whole suite of applications that are available for this purpose. Collectively these applications are referred to as Microsoft Office XP. Generally included in Office are Microsoft's Word, Excel, and PowerPoint, and developer versions include Access and FrontPage. All of these Office applications have the ability to be programmed with Visual Basic for Applications and an extensive object model exposed using OLE Automation, or just Automation.

Note: Microsoft Office XP's MS-Excel was used for the examples in this article.

To demonstrate how Automation works in Office XP I will be providing an Automation controller example that shows you how to control Microsoft Excel. In this scenario Excel is the code that we are reusing, and

in this context Excel is referred to as an Automation server. The sample client application-the code that we are writing-plays the role of Automation controller.

The key to successfully implementing an Automation controller for any of the Office XP applications is to name what it is that you want to accomplish and then find the entities in the OLE Automation interface for that Office application and program to that interface. Each Office application has its own object model. For example, you can look up the MS-Excel object model in the Visual Basic for Applications editor's help, as illustrated in figure 12.6 .

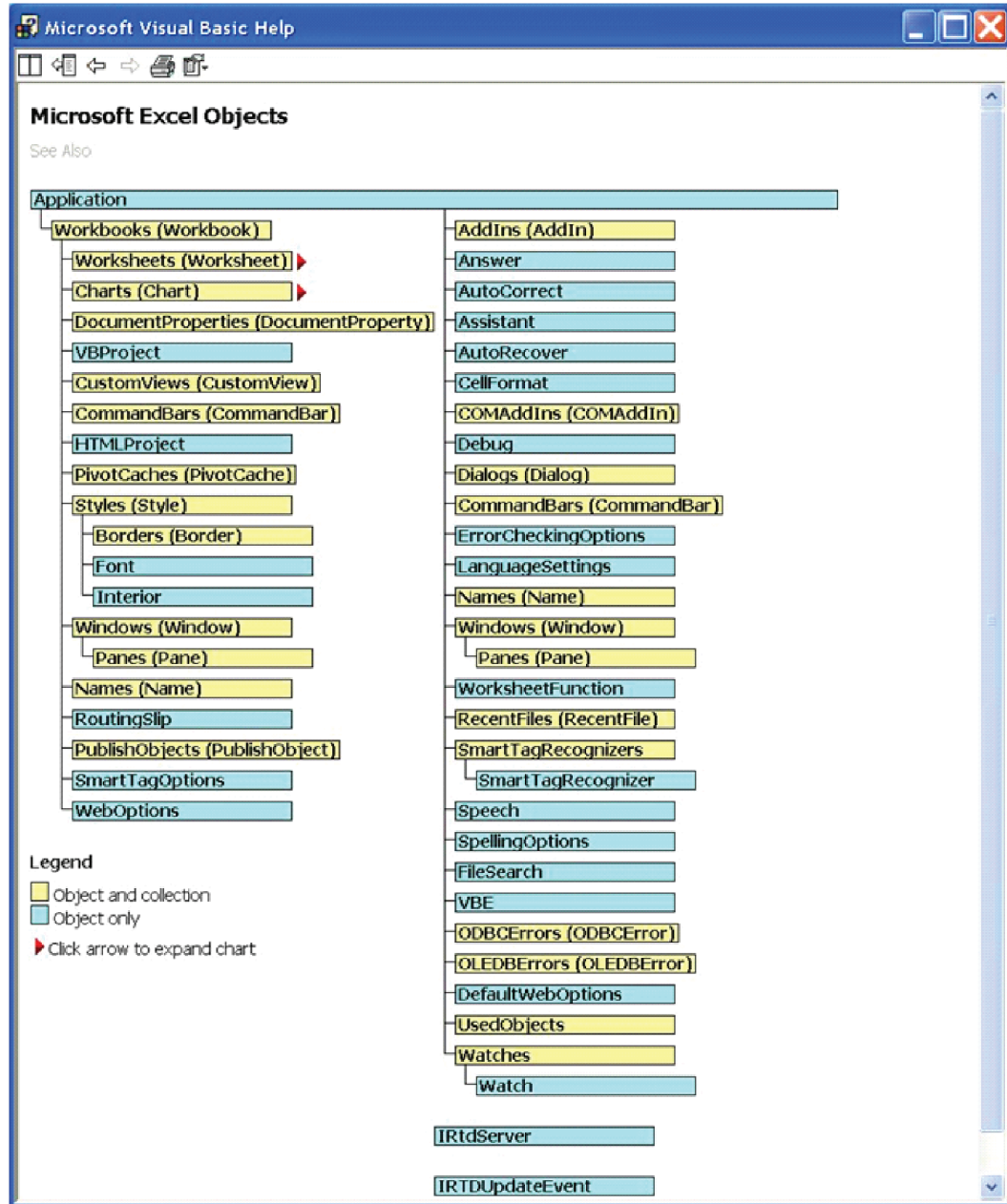


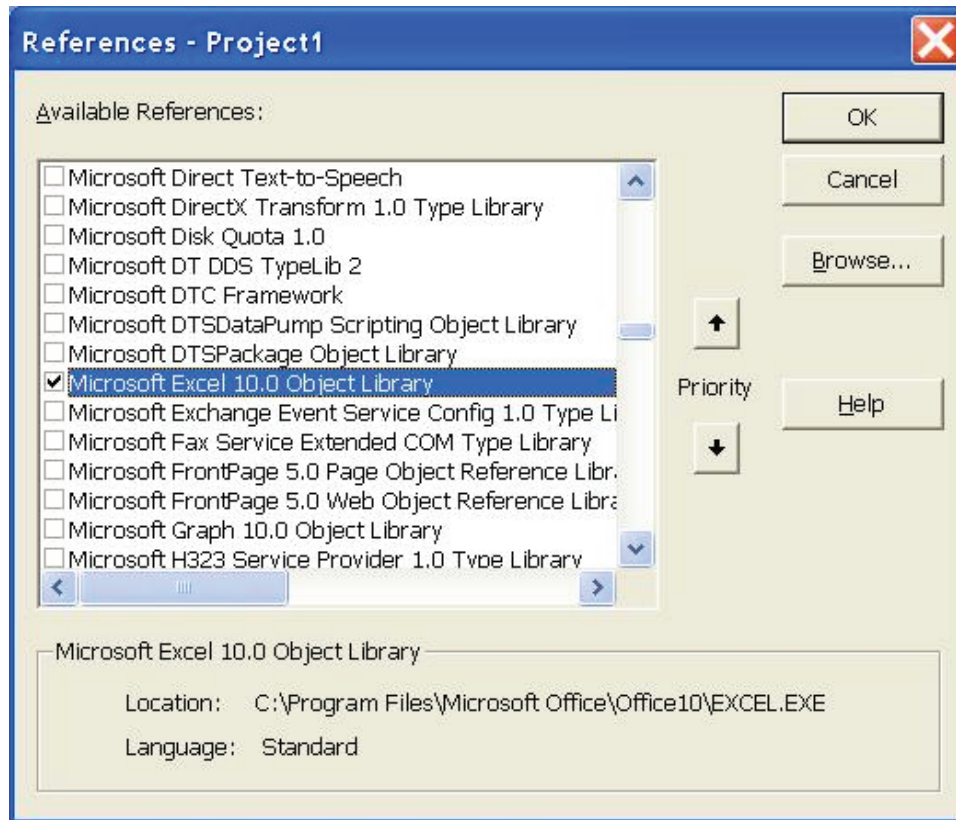
Figure 12.6 : Excel's object model

For our purposes we will create an Automation controller that enables us to start an instance of Excel and dump some random data into the Worksheet. Clearly, this serves only as a basic introduction. However, once you understand the rudiments everything else is a matter of figuring out what part of the object model contains the data and capabilities that you need to get the job done.

Adding a Reference to Excel

Our example will implement an Automation controller in VB6 for MS-Excel. The first thing we need to do after we have identified our objective and created a project is to add a reference to the Excel Object library.

To accomplish this select Project | References in Visual Basic 6 and check the Microsoft Excel 10.0 Object Library as shown in figure 2.



Add a reference to the Microsoft Excel 10.0 Object Library

It is worth noting that Automation has been around for quite a while in MS-Office. Thus if you are using an earlier version of Office then the version number of the Object Library will change. If you don't have Excel installed then code that is very similar to the example code in this article can be used to experiment with Automation in one of the other Microsoft Office applications, like Word.

Creating an Instance of Excel

After you have added a reference to the Excel object library you can easily control Excel from your client application. The key is to declare a variable of type `Excel.Application` and create a new instance of that type. You can perform this step when your client application is loaded-as shown in listing 1-or upon demand at some later time.

Note: If you are an experienced programmer then you know that you do not have to add a reference to Excel to create an Automation controller. You can declare a variable of type `Object` and use the `CreateObject("Excel.Application")` method call. This is referred to as a late binding and results in weaker code than does introducing the library and using specific types.

Listing 1: The `Form_Load` event handler creates an instance of the `Excel.Application` object.

```
Private Excel As Excel.Application
Private Sub Form_Load()
    Set Excel = New Excel.Application
    Set Workbook = Excel.Workbooks.Add
End Sub
```

From the listing you can see that the reference to an instance of Excel is declared outside of the method in the Form's scope (or class scope as an alternative). A new instance of the object is created and assigned to the variable named Excel. I also elected to create a new Workbook object in the Load event. You could create these objects at any time. After Load runs we can verify that Excel is running by opening the Windows Task Manager-see figure 3-and look for the Excel.exe executable in the task manager.

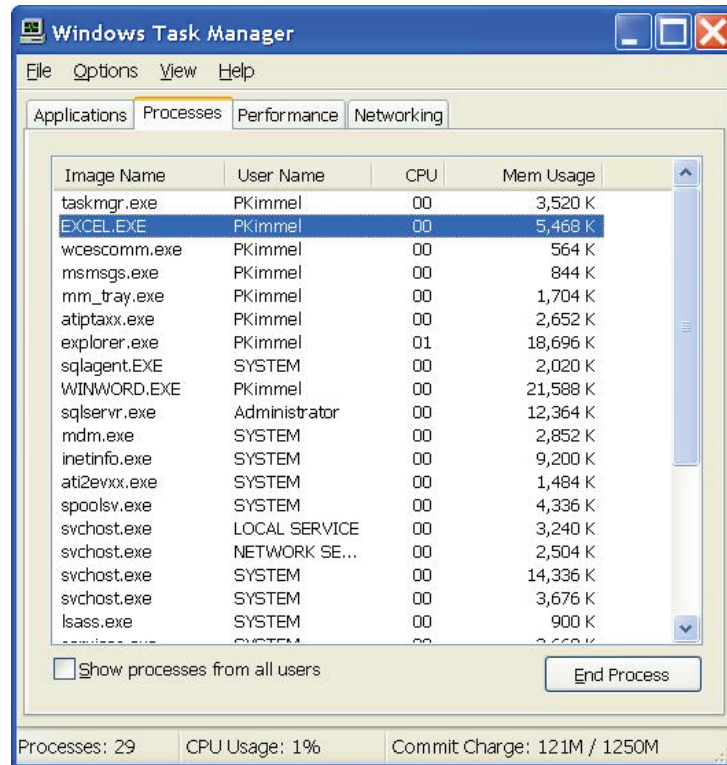


Figure 3: Microsoft Excel running as an application server, shown in the windows XP Task Manager

By default when you start Excel as an Automation server it is not visible. I added some code that demonstrates how to show or conceal the running Automation server (see listing 2).

Listing 2: One example that demonstrates how to manage revealing and concealing the running instance of Excel created in listing 1.

```
Private Sub MenuShow_Click()
    If(MenuShow.Caption = "&Show") Then
        MenuShow.Caption = "&Hide"
        Workbook.Activate
        Excel.Visible = True
    Else
        MenuShow.Caption = "&Show"
        Excel.Visible = False
    End If
End Sub
```

In our example the Visible state of the instance of Excel we created is associated closely with the value of a menu caption. If the caption is "&Show" then we toggle the Visible state to True, Active the Workbook, and

toggle the menu caption to a suitable value, indicating that when we click the menu the next time the instance of Excel will be hidden. Any similar strategy for showing and hiding the server instance is sufficient. The key is to remember to examine or modify the Visible state of the Application object for this purpose.

Using the Active Workbook and Creating a Worksheet

You can interact with Excel in as many ways as are exposed by the object model. MS-Excel is workbook and worksheet oriented. A file is referred to as a workbook and a single spreadsheet is referred to as a worksheet. For our purposes we need one workbook and one worksheet. Listing 1 demonstrated how to add a new Workbook and the next listing (listing 3) demonstrates how to add data to the active Worksheet.

Listing 3: Add data to the active Worksheet in Excel.

```
Private Row As Integer
Private Sub MenuAdd_Click()
    Dim Worksheet As Worksheet
    Set Worksheet = Workbook.ActiveSheet
    Row = Row + 1
    Worksheet.Rows.Cells(Row, 1) = Text1.Text
End Sub
```

code that adds data to our Worksheet by declaring a temporary variable. A private field named Row is used to store an internal counter and the text in Text1 is added to column 1 and whatever the current value of Row is.

The Automation controller is a very simplistic application that simply dumps data into a Worksheet into the first column. However, at this point you have all of the basic rudiments that get you to data at the cell level. The final step is to demonstrate how we can clean up our work area.

Cleaning up Your Work Area

For our purposes the Worksheet represents a temporary work area. The data put into the Worksheet is not persistent; that is, it is not intended to outlive the current run of the Automation controller application. Listing 4 demonstrates how we can close the active Workbook, discarding the temporary data and shut down the instance of Excel.

Listing 4: Discarding changes to the active Workbook and shutting down the Automation server.

```
Private Sub Form_Unload(Cancel As Integer)
    Call Workbook.Close(False)
    Excel.Quit
    Set Excel = Nothing
End Sub
```

12.9 Summary

The most difficult challenge when implementing an Automation controller application is to learn what the Automation server's object model has to offer. After you learn how to start and stop instances of the server all you need to do is know the name of the objects that contain the data you are interested in. The objects in the object model will be different for each server. Fortunately, the applications in Office XP represent well-understood solution domains and as a result are reasonably intuitive to understand.

It makes sense that there are objects in the object model that match objects you use when using Excel as a client application. For example, it is reasonable to expect that there is a representation of a Workbook, Worksheet, columns, rows, and cells. However, there is much more. If you can identify the entity you are looking for in the Excel client then it will be much easier to find in the Excel Automation object model. The same relative contextual relationships should be supported too. For example, Workbooks should have something similar to a collection of Worksheets, and Worksheets should have collections of columns, rows, and cells.

12.10 Self Assessment Questions

- 1 What do you understand by the following
 - (i) OLE
 - (ii) Linked object
 - (iii) Embedded object
- 2 What does an OLE container control do?
- 3 Define OLE autosize.
- 4 Discuss some basic properties of OLE container control.
5. Explain with example using Excel sheet for OLE.

Unit - 13 : Object Orientated Programming in VB

Structure of Unit:

- 13.0 Objective
- 13.1 Introduction
- 13.2 Classes in Visual Basic
- 13.3 Instantiating an Object
- 13.4 Your First Class Module
- 13.5 The Class Initialize event
- 13.6 Object instantiation
- 13.7 Object termination
- 13.8 The Class Terminate Event
- 13.9 Creating An Object From A Class
- 13.10 Summary
- 13.11 SelfAssessment Questions

13.0 Objective

This unit gives overview of Object Oriented Programing concept in Visual Basic. it also gives clear idea about Object, Class, Inheritance, Polymorphism.

13.1 Introduction

VB6 is not a full OOP in the sense that it does not have inheritance capabilities although it can make use of some benefits of inheritance. However, VB2010 is a fully functional Object Oriented Programming Language, just like other OOP such as C++ and Java. It is different from the earlier versions of VB because it focuses more on the data itself while the previous versions focus more on the actions. Previous versions of VB are known as procedural or functional programming language. Some other procedural programming languages are C, Pascal and Fortran.

A class is a portion of the program (a source code file, in Visual Basic) that defines the properties, methods, and events-in a word, behavior-of one or more objects that will be created during execution. An object is an entity created at run time, which requires memory and possibly other system resources, and is then destroyed when it's no longer needed or when the application ends. In a sense, classes are design time-only entities, while objects are run time-only entities.

Users will never see a class; rather, they'll probably see and interact with objects created from your classes, such as invoices, customer data, or circles on the screen.

Language while VB6 may have OOP capabilities, it is not fully object oriented. In order to qualify as a fully object oriented programming language, it must have three core technologies namely encapsulation, inheritance and polymorphism. These three terms are explained below:

Encapsulation

refers to the creation of self-contained modules that bind processing functions to the data. These user-defined data types are called classes. Each class contains data as well as a set of methods which manipulate the data. The data components of a class are called instance variables and one instance of a class is an object. For example, in a library system, a class could be member, and John and Sharon could be two instances (two objects) of the library class.

Inheritances

Classes are created according to hierarchies, and inheritance allows the structure and methods in one class to be passed down the hierarchy. That means less programming is required when adding functions to complex systems. If a step is added at the bottom of a hierarchy, then only the processing and data associated with that unique step needs to be added. Everything else about that step is inherited. The ability to reuse existing objects is considered a major advantage of object technology.

Polymorphism

Object-oriented programming allows procedures about objects to be created whose exact type is not known until runtime. For example, a screen cursor may change its shape from an arrow to a line depending on the program mode. The routine to move the cursor on screen in response to mouse movement would be written for "cursor," and polymorphism allows that cursor to take on whatever shape is required at runtime. It also allows new shapes to be easily integrated.

VB6 allows users to write programs that break down into modules. These modules will represent the real-world objects and are known as classes or types. An object can be created out of a class and it is known as an instance of the class. A class can also comprise subclass. For example, apple tree is a subclass of the plant class and the apple in your backyard is an instance of the apple tree class. Another example is student class is a subclass of the human class while your son John is an instance of the student class.

13.2 Classes in Visual Basic

In Visual Basic 6.0, a class is defined in a class module. A single class module is stored in a special type of file that has a .cls file extension. In Visual Basic 2005, a class is defined in a Class statement that specifies the name and the members of a class. Class statements are stored in source files. The entire source file can be viewed as plain text. Multiple Class statements as well as other type statements can be stored in single source file. Visual Basic does not require the name of the source file to match a Class or type defined in the source file.

Global Classes

In Visual Basic 6.0, the value of the Instancing property of a class determines whether a class is private - that is, for use only within one component, or available for other applications to use. It also determines how other applications create instances of the class and how other applications call the class member. In Visual Basic 2005, the Instancing property is no longer supported.

You can apply the Public keyword to class statements in an assembly to expose classes in that assembly to other assemblies.

You can reference an external assembly such as a class library to enable code in your application to use the Public classes from that class library.

You can use the Imports keyword to import namespace names from referenced projects and assemblies. The Imports keyword can also import namespace names defined within the same project as the file in which the statement appears.

You can apply the Shared keyword to field, property, and method members of classes and structures to implement shared members. Shared members are properties, procedures, and fields that are shared by all instances of a class or structure. The shared members of a class can be accessed without instantiating the class.

Classes are added to the project by either right-clicking on the Project Name in the Project explorer and selecting Add and Class Module or from the Project menu and selecting Add Class Module. Unlike modules, classes are generally identified by a Capital C rather than a three-letter mnemonic at the start of the name. The following is an example of a Class Clock called "CClock". The class has the data members mHour, mMinute, and mSecond as attributes, and the operations (things it can do) setTime and displayTime.

Option Explicit

' Attributes of the class

Private mHour As Integer

Private mMinute As Integer

Private mSecond As Integer

' Operations for the class

Public Sub setTime(ByVal h As Integer, ByVal m As Integer, ByVal s As Integer)

 ' Ensure the time is a valid time

 If h >= 0 And h < 24 Then

 mHour = h

 Else

 mHour = 0

 End If

 If m >= 0 And m < 60 Then

 mMinute = m

 Else

 mMinute = 0

 End If

 If s >= 0 And s < 60 Then

 mSecond = s

 Else

 mSecond = 0

 End If

End Sub

Public Function displayTime() As String

 displayTime = Format(mHour, "00") & ":" & _

 Format(mMinute, "00") & ":" & _

 Format(mSecond, "00")

End Function

A class consists of data members as well as methods. In VB2010, the program structure to define a Human class can be written as follows:

Public Class Human

' Data Members

Private Name As String

Private Birthdate As String

Private Gender As String

Private Age As Integer

' Methods

```

Overridable Sub ShowInfo( )
MessageBox.Show(Name)
MessageBox.Show(Birthdate)
MessageBox.Show(Gender)
MessageBox.Show(Age)
End Sub
End Class

```

After you have created the human class, you can create a subclass that inherits the attributes or data from the human class. For example, you can create a students class that is a subclass of the human class. Under the student class, you don't have to define any data fields that are already defined under the human class, you only have to define the data fields that are different from an instance of the human class. For example, you may want to include StudentID and Address in the student class. The program code for the StudentClass is as follows:

```

Public Class Students
Inherits Human
Public StudentID as String
Public Address As String
Overrides Sub ShowInfo( )
MessageBox.Show(Name)
MessageBox.Show(StudentID)
MessageBox.Show(Birthdate)
MessageBox.Show(Gender)
MessageBox.Show(Age)
MessageBox.Show(Address)
End Sub

```

13.3 Instantiating an Object

An object is instantiated with the New keyword. This can either be done when the variable is declared, or using the Set statement. The following declares a variable that is assigned an object reference. The following declares a variable and is instantiated using the Set Statement.

```

Dim t As CClock
Set t = New CClock
Me.Print "Initial time is: " & t.displayTime()
t.setTime 19, 30, 15
Me.Print "After calling SetTime: " & t.displayTime()
Set t = Nothing

```

Aggregation

Classes may themselves contain other classes. This type of arrangement is known as Aggregation or Composition and promotes reuse, one of the major features of the Object Oriented approach. An example of Aggregation is used in the Properties example.

Generalisation

Generalisation is a means of abstraction. More specific classes are Inherited from more general classes to define data and behaviour.

Interface Inheritance

Visual Basic 6 does not support Inheritance, but does support Interface Inheritance. Interface methods are usually empty shells with a code body to provide a Super class from which other classes may inherit the interface. Interface classes are prefixed with a capital I (eg. IShape). VB.Net does support Inheritance, as well as function overloading. In the following example, an Interface Class called IShape defines the method calculateArea that may be used by shapes such as circle, rectangle, triangle, etc. The actual implementation of the calculateArea would be different for each shape, and therefore cannot be implemented. When a method behaves differently in derived classes, but essentially means the same thing, it is said to be Polymorphic.

Two classes are defined, CCircle and CRectangle which implement the Interface Class IShape. To state that they implement the interface class, the statement Implements IShape is added at the top of both classes. The classes then have to provide a definition for the abstract method IShape_calculateArea.

The IShape Interface Class:

Option Explicit

Public Function calculateArea() As Double

' Left empty

End Function

Class_Initialize

In Visual Basic 6.0, a class Initialize event can be handled by a Class_Initialize method to execute code that needs to be executed at the moment an object is created. For example, the values of class data variables can be initialized. In Visual Basic 2005, The Initialize event and the Class_Initialize handler are not supported. To provide class initialization, add one or more constructor methods to the classes and structures you define.

Me Keywords:- The Me keyword provides a way to refer to the specific instance of a class or structure in which the code is currently executing.

Class_Initialize:- Visual Basic 6.0 provides support for the constructor concept through the Class_Initialize method. This method is private and allows no parameters. This method is called automatically when a new instance of the class is created. A call to the New keyword calls the Class_Initialize method, if it exist. Suppose you want to specify the yearly growth of a tree with a default of five years. The class code would look something like this:

‘ Visual Basic 6.0

Private mvarYearlyGrowth As Integer

Public Property Get YearlyGrowth() As Integer

YearlyGrowth = mvarYearlyGrowth

End Property

Public Property Let YearlyGrowth(ByVal newValue As Integer)

mvarYearlyGrowth = newValue

End Property

Private Sub Class_Initialize()

mvarHeight = 5

End Sub

13.4 Your First Class Module

Creating a class in Visual Basic is straightforward: just issue an Add Class Module command from the Project menu. A new code editor window appears on an empty listing. By default, the first class module is named Class1, so the very first thing you should do is change this into a more appropriate name. In this first example, it show how to encapsulate personal data related to a person, so this first class CPerson is assumed.

Microsoft suggests that you use the cls prefix for class module names, but I don't comply simply because I feel it makes my code less readable. I often prefer to use the shorter C prefix for classes (and I for interfaces), and sometimes I use no prefix at all, especially when objects are grouped in hierarchies. Of course, this is a matter of personal preference, and I don't insist that my system is more rational than any other.

The first version of our class includes only a few properties. These properties are exposed as Public members of the class module itself, as you can see in this code and also in Figure 13.1 on the following page:

* In the declaration section of the CPerson class module

```
Public FirstName As String
```

```
Public LastName As String
```

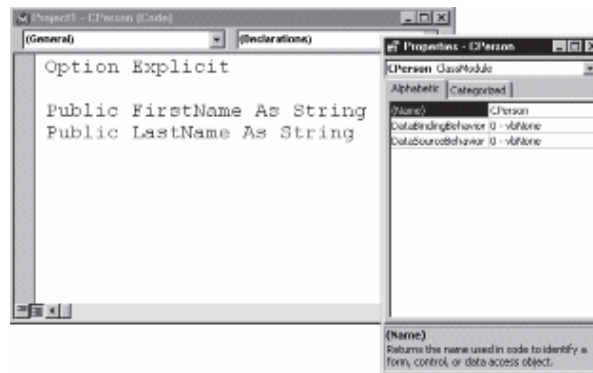


Figure 13.1 Creating a class module, giving it a name in the Properties window, and adding some Public variables in the code editor window.

This is a very simple class, but it's a good starting point for experimenting with some interesting concepts, without being distracted by details. Once you have created a class module, you can declare an object variable that refers to an instance of that class:

* In a form module

```
Private Sub cmdCreatePerson_Click()  
Dim pers As CPerson ' Declare.  
Set pers = New CPerson ' Create.  
pers.FirstName = "John" ' Assign properties.  
pers.LastName = "Smith"  
Print pers.FirstName & " " & pers.LastName ' Check that it works.  
End Sub
```

13.5 The Class Initialize event

As you start building classes, you'll soon notice how often you want to assign a well-defined value to a property at the time of the creation of the object itself, without having to specify it in the caller code. For

example, if you're dealing with an Employee object you can reasonably expect that in most cases its Citizenship property is "American" (or whatever nationality applies where you live). Similarly, in most cases the AddressFrom property in a hypothetical Invoice object will probably match the address of the company you're working for. In all cases, you'd like for these default values to be assigned when you create an object, rather than your having to assign them manually in the code that uses the class.

Visual Basic offers a neat way to achieve this goal. In fact, all you have to do is write some statements in the Class_Initialize event of the class module. To have the editor create a template for this event procedure, you select the Class item in the leftmost combo box in the code editor. Visual Basic automatically selects the Initialize item from the rightmost combo box control and inserts the template into the code window. Here's a Citizenship property that defaults to "American":

```
' The Private member variable
Private m_Citizenship As String
Private Sub Class_Initialize()
    m_Citizenship = "American"
End Sub
* Code for Public Property Get/Let Citizenship procedure ... (omitted)
```

If you now run the program you have built so far and trace through it, you'll see that as soon as Visual Basic creates the object (the Set command in the form module), the Class_Initialize event fires. The object is returned to the caller with all the properties correctly initialized, and you don't have to assign them in an explicit way. The Class_Initialize event has a matching Class_Terminate event, which fires when the object instance is destroyed by Visual Basic. In this procedure, you usually close your open files and databases and execute your other cleanup tasks. I will describe the Class_Terminate event at the end of this chapter.

Debugging a class module

In most respects, debugging code inside a class module isn't different from debugging code in, say, a form module. But when you have multiple objects that interact with one another, you might easily get lost in the code. Which particular instance are you looking at in a given moment? What are its current properties? Of course, you can use all the usual debugging tools—including Debug.Print statements, data tips, Instant Watch, and so on. But the one that beats them all is the Locals window, which you can see in Figure 13.2 Just keep this window open and you'll know at every moment where you are, how your code affects the object properties, and so on. All in real time.

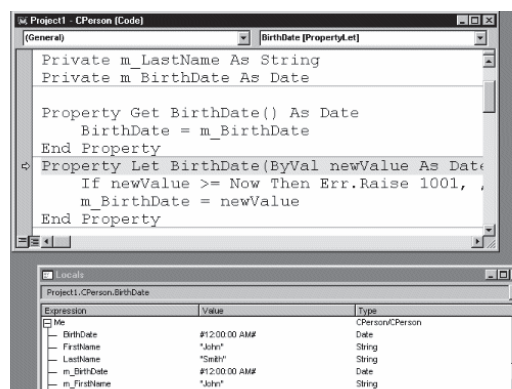


Figure 13.2 The Locals window is a great debugging tool when you're working with multiple objects.

13.6 Object instantiation

The first time Visual Basic creates an object of a given class, its run-time module performs the following sequence of operations (here in a simplified form):

Allocates a block of memory for the compiled code generated from the class module and loads the code from disk.

Allocates a (smaller) block of memory for the VTable itself, and then fills it with the entry point addresses of each public routine in the class module.

Allocates a block for the particular object instance and sets its first 32-bit value to point to the VTable. At this point, it also fires the `Class_Initialize` event procedure so that the variable area can be correctly initialized.

Stores the address of the instance data area in the target object variable. At this point, the client code can do whatever it wants with the object.

This long sequence has to be performed only the very first time your code creates an object of a given class. For all subsequent objects of the same class, steps 1 and 2 are skipped because the VTable is already in place. And when you're simply assigning object variables (that is, `Set` commands without a `New` clause), step 3 is also skipped and the whole operation becomes just an assignment of a 32-bit value.

13.7 Object termination

When no more object variables point to a given instance data block, the object is destroyed. Just before releasing the memory, the Visual Basic runtime invokes the `Class_Terminate` event procedure in the class module-if the programmer created one. This is the routine in which you place your clean-up code.

Visual Basic never goes further than that, and for example, it doesn't release the VTable either even if there isn't any other object pointing to it. This is an important detail because it ensures that the next time another object of this class is created, the overhead will be minimal. There are just a couple of other things that you should know about the termination phase:

Visual Basic relies on a safety mechanism that prevents an object from being destroyed while its procedures are being executed. Think of the following scenario: You have a global object variable that holds the only reference to an object, and within a procedure in the class module you set the global variable to `Nothing`, thus destroying the only reference that keeps the object alive. If Visual Basic were really stupid, it would immediately terminate the procedure as soon as it performed the assignment. Instead, Visual Basic patiently waits for the procedure to end and only then destroys the object and invokes its `Class_Terminate` event procedure. A class module should never reference a global variable because this would break its self-containment.

Once you're executing code in the `Class_Terminate` event procedure, Visual Basic has already started the object termination process and you can't do anything to prevent the object from being destroyed. For example, in a situation like the preceding one, you might believe that you could keep an object alive by assigning a new object reference to the global variable, hoping to reincrement the internal reference counter and prevent the object destruction. If you tried that, however, what actually would happen is that Visual Basic would first complete the destruction of the current object and then create a new instance that had nothing to do with the one you were in.

13.8 The Class Terminate Event

Visual Basic fires the `Class_Terminate` event one instant before releasing the data instance block and terminating the object's life. You usually write code for this event when you need to undo things that you did at initialization time or during the life of the instance. Typically in this event procedure, you close any open files and release Windows resources obtained through direct API calls. If you want to make the object's properties persist in a database for a future session, this is where you usually do it. All in all, however, you'll rarely write code for

this event or at least you'll need it less frequently than code for the Class_Initialize event. For example, the CPerson class module doesn't actually require code in its Class_Terminate event procedure.

On the other hand, the mere fact that you can write some executable code and be sure that it will be executed when an object is destroyed opens up a world of possibilities that couldn't be exploited using any other, non-OOP technique. To show you what I mean, I've prepared three sample classes that are almost completely based on this simple concept. It's a great occasion to show how you can streamline several common programmer tasks using the power that objects give you.

Caution Visual Basic calls the Class_Terminate event procedure only when the object is released in an orderly manner-that is, when all references pointing to it are set to Nothing or go out of scope, or when the application comes to an end. This includes the case when the application ends because of a fatal error. The only case when Visual Basic does not invoke the Class_Terminate event is when you abruptly stop a program using the End command from the Run menu or the End button on the toolbar. This immediately stops all activity in your code, which means that no Class_Terminate event will ever be invoked. If you inserted critical code in the Terminate events-for example, code that releases Windows resources allocated via APIs-you'll experience problems. Sometimes these are big problems, including system crashes. By the same token, never terminate a program using an End statement in code: This has exactly the same effect, but it's going to create problems even after you compile the application and run it outside the environment.

Example 1: managing the mouse cursor

Programmers commonly change the shape of the mouse cursor, typically to an hourglass, to inform the user that some lengthy operation is going on. Of course, you also have to restore the cursor before exiting the current procedure; otherwise, the hourglass stays visible and the user never realizes that the wait is over. As simple as this task is, I've found that a good number of commercial applications fail to restore the original shape under certain circumstances. This is a clear symptom that the procedure has exited unexpectedly and therefore missed its opportunity to restore the original shape. How can classes and objects help you avoid the same error? Just have a look at this simple CMouse class module:

The CMouse class - complete source code

```
Dim m_OldPointer As Variant
' Enforce a new mouse pointer.
Sub SetPointer(Optional NewPointer As MousePointerConstants = vbHourglass)
' Store the original pointer only once.
If IsEmpty(m_OldPointer) Then m_OldPointer = Screen.MousePointer
Screen.MousePointer = NewPointer
End Sub

' Restore the original pointer when the object goes out of scope.
Private Sub Class_Terminate()
' Only if SetPointer had been actually called
If Not IsEmpty(m_OldPointer) Then Screen.MousePointer = m_OldPointer
End Sub
```

Not bad, eh? Just eight lines of code (not counting comments) to solve a recurring bug once and for all! See how easy it is to use the class in a real program:

```
Sub ALengthyProcedure()
Dim m As New CMouse
```



```

m.SetPointer vbHourglass ' Or any other pointer shape
' ... slow code here ... (omitted)
End Sub

```

The trick works because as soon as the variable goes out of scope, the object is destroyed and Visual Basic fires its `Class_Terminate` event. The interesting point is that this sequence also occurs if the procedure is exited because of an error; even in that case, Visual Basic releases all the variables that are local to the procedure in an orderly fashion.

Example 2: opening and closing files

Another common programming task is opening a file to process it and then closing it before exiting the procedure. All the procedures that deal with files have to protect themselves against unanticipated errors because if they were exited in an abrupt way they wouldn't correctly close the file. Once again, let's see how a class can help us to deliver more robust code with less effort:

The `CFile` class--complete source code

```

Enum OpenModeConstants
    omInput
    omOutput
    omAppend
    omRandom
    omBinary
End Enum

Dim m_Filename As String, m_Handle As Integer

Sub OpenFile(Filename As String, _
    Optional mode As OpenModeConstants = omRandom)
    Dim h As Integer
    ' Get the next available file handle.
    h = FreeFile()
    ' Open the file with desired access mode.
    Select Case mode
        Case omInput: Open Filename For Input As #h
        Case omOutput: Open Filename For Output As #h
        Case omAppend: Open Filename For Append As #h
        Case omBinary: Open Filename For Binary As #h
        Case Else ' This is the default case.
            Open Filename For Random As #h
    End Select
    ' (Never reaches this point if an error has occurred.)
    m_Handle = h
    m_Filename = Filename
End Sub

' The filename (read-only property)
Property Get Filename() As String
    Filename = m_Filename

```

End Property

' The file handle (read-only property)

Property Get Handle() As Integer

Handle = m_Handle

End Property

' Close the file, if still open.

Sub CloseFile()

If m_Handle Then

Close #m_Handle

m_Handle = 0

End If

End Sub

Private Sub Class_Terminate()

' Force a CloseFile operation when the object goes out of scope.

CloseFile

End Sub

This class solves most of the problems that are usually related to file processing, including finding the next available file handle and closing the file before exiting the procedure:

' This routine assumes that the file always exists and can be opened.

' If it's not the case, it raises an error in the client code.

Sub LoadFileIntoTextBox(txt As TextBox, filename As String)

Dim f As New CFile

f.OpenFile filename, omInput

txt.Text = Input\$(LOF(f.Handle), f.Handle)

' No need to close it before exiting the procedure!

End Sub

Example 3: creating a log of your procedures

In the end simple class that you'll probably find useful when debugging dozens of nested procedures that call one another over and over. In such cases, nothing can preserve your sanity more effectively than a log of the actual sequence of calls. Unfortunately, this is easier said than done because while it is trivial to add a Debug.Print command as the first executable statement of every procedure, trapping the instant when the procedure is exited is a complex matter-especially if the procedure has multiple exit points or isn't protected by an error handler. However, this thorny problem can be solved with a class that counts exactly eight lines of executable code:

' Class CTracer - complete source code.

Private m_procname As String, m_enterTime As Single

Sub Enter(procname As String)

m_procname = procname: m_enterTime = Timer

' Print the log when the procedure is entered.

Debug.Print "Enter " & m_procname

End Sub

```
Private Sub Class_Terminate()
' Print the log when the procedure is exited.
Debug.Print "Exit " & m_procname & " - sec. " & (Timer - m_enterTime)
End Sub
```

Using the class is straightforward because you have to add only two statements on top of any procedure that you want to trace:

```
Sub AnyProcedure()
Dim t As New CTracer
t.Enter "AnyProcedure"
' ... Here is the code that does the real thing ...(omitted).
End Sub
```

The CTracer class displays the total time spent within the procedure, so it also works as a simple profiler. It was so easy to add this feature that I couldn't resist the temptation.

This chapter introduced you to object-oriented programming in Visual Basic, but there are other things to know about classes and objects, such as events, polymorphism, and inheritance. I describe all these topics in the next chapter, along with several tips for building more robust Visual Basic applications.

13.9 Creating An Object From A Class

When you want to create an object from a code component, you first add a reference to that code component using the Visual Basic Project|References menu item (note that the code component must be registered with Windows, which most applications will do automatically when installed). Next, you create that object in code, and there are three ways to create objects from code components in Visual Basic (we'll use the CreateObject technique in this chapter):

- o Declaring the variable using the New keyword (in statements like Dim, Public, or Private), which means Visual Basic automatically assigns a new object reference the first time you use the variable (for example, when you refer to one of its methods or properties). This technique only works with code components that supply type libraries (as the code components created with Visual Basic do). These type libraries specify what's in the code component.

- * Assigning a reference to a new object in a Set statement by using the New keyword or CreateObject function.

- * Assigning a reference to a new or existing object in a Set statement by using the

Get Object function.

Let's see some examples. Here, we'll assume we've already added a reference to the code component containing the classes we'll work with. If the code component you're using supplies a

type library, as the ones built with Visual basic do, you can use the New keyword when you declare the object to create it. As an example, here we're creating an object named objSorter of

the Sorter class:

```
Dim objSorter As New Sorter objSorter.Sort
```

The object is actually only created when you refer to it for the first time. Whether or not a code component supplies a type library, you can use the `CreateObject` function in a `Set` statement to create a new object and assign an object reference to an object variable. To use `CreateObject`, you pass it the name of the class you want to create an object of. For example, here's how we created a new object named `objExcel` from the Microsoft Excel code component library in the previous topic, using `CreateObject`:

```
Dim objExcel As Object  
Set objExcel = CreateObject  
("Excel.Sheet")
```

Note the way we specify the class name to create the object from-as `Excel.Sheet` (that is, as `CodeComponent.Class`). Because the code components you create with `CreateObject` don't need a type library, you must refer to the class you want to use as `CodeComponent.Class` instead of just by class, as you can with `New`.

You can also use the `GetObject` function to assign a reference to a new class, although it's usually used to assign a reference to an existing object. Here's how you use `GetObject`:

`Set objectvariable = GetObject([pathname] [, class])` Here's what the arguments of this function mean:

- * `pathname`-The path to an existing file or an empty string. This argument can be omitted.
- * `Class`-The name of the class you want to create an object of. If `pathname` is omitted, then `class` is required. Passing an empty string for the first argument makes `GetObject` work like `CreateObject`, creating a new object of the class whose name is in `class`. For example, here's how we create an object of the class `ExampleClass` in the code component `NewClass` using `GetObject` (once again, we refer to the class we're using as `CodeComponent.Class`):

```
Set objNewClass = GetObject("", "NewClass.ExampleClass")
```

13.10 Summary

VB6 is not a full OOP in the sense that it does not have inheritance capabilities although it can make use of some benefits of inheritance. However, VB2010 is a fully functional Object Oriented Programming Language, just like other OOP such as C++ and Java. A class is a portion of the program (a source code file, in Visual Basic) that defines the properties, methods, and events-in a word, behavior-of one or more objects that will be created during execution. An object is an entity created at run time, which requires memory and possibly other system resources, and is then destroyed when it's no longer needed or when the application ends. In a sense, classes are design time-only entities, while objects are run time-only entities.

13.11 Self Assessment Questions

- 1 Explain how we can initialize an object in class.
- 2 Define class initialize event.
- 3 How do you create an object from a class.
- 4 Explain class terminate event.

Unit - 14 : Crystal Reports

Structure of Unit:

- 14.0 Objectives
- 14.1 Introduction
- 14.2 Sample Database
- 14.3 Creating a New Report
 - 14.3.1 Creating a Report By Wizard
 - 14.3.2 Creating Reports By Design Layout
- 14.4 Different Sections of Reports
- 14.5 To Add Report Header
- 14.6 To add Page Header/ Field Headers
- 14.7 Inserting Groups
- 14.8 Inserting Totals and Grand Totals
- 14.9 How to Connect Crystal Report to VB Application
- 14.10 Summary
- 14.11 Self Assessment Questions

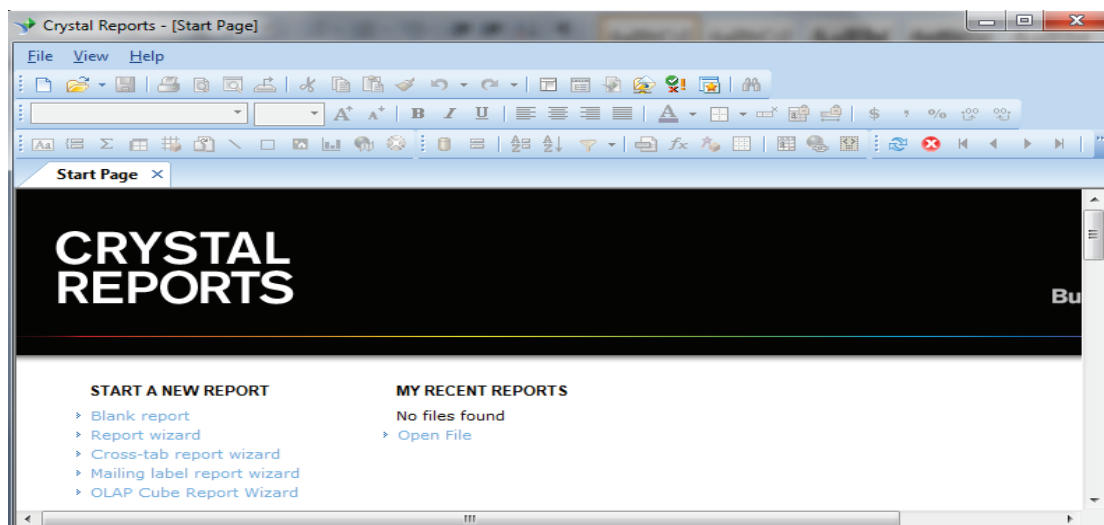
14.0 Objectives

This chapter provides a general overview of

- Creating a New Report
- Different Sections of Reports
- To Add Report Header/Page Header/ Field Headers
- Inserting Groups/Totals and Grand Totals
- How to Connect Crystal Report to VB Application

14.1 Introduction

Crystal Reports provides a relatively easy way to incorporate reporting into your application. The basis of a report is a database query. To open crystal report Click on Start 'All Programs ' Crystal Report (Whatever version you are using. This chapter is based on Crystal Report 2008). The window will be displayed like



14.2 Sample Database

The sample database used for this is an Access 2007 format database named EMPLOYEE.MDB. EMPLOYEE.MDB contains two tables: Emp, and Dept The tables are structured as follows:

Emp Table

Field Name	Data Type
EmpNo	AutoNumber
Fname	Text
Lname	Text
Deptno	Number
DOJ	Date/Time
Designation	Text

Dept Table

Field Name	Data Type
Deptno	Number
Dname	Text
Location	Text

14.3 Creating a New Report

Report can be created either by using Report wizard or manually using Design layout.

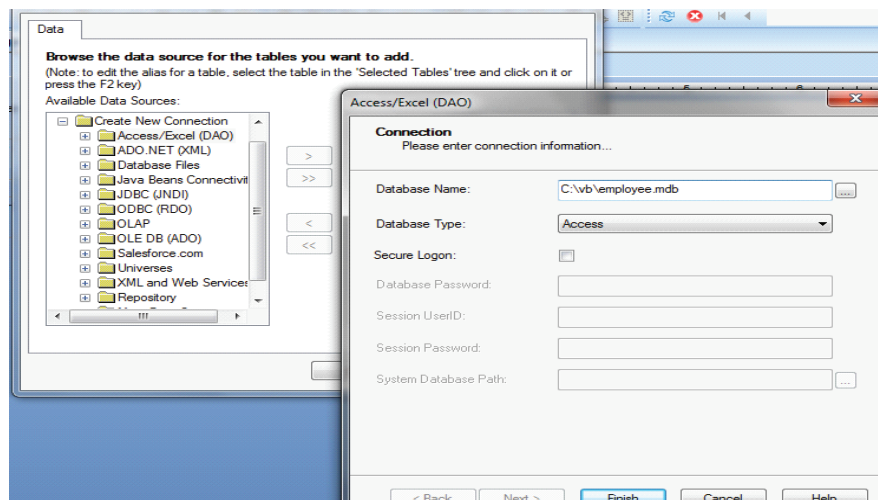
14.3.1 Creating Reports By Report Wizard


Click on File ' New 'Report Wizard

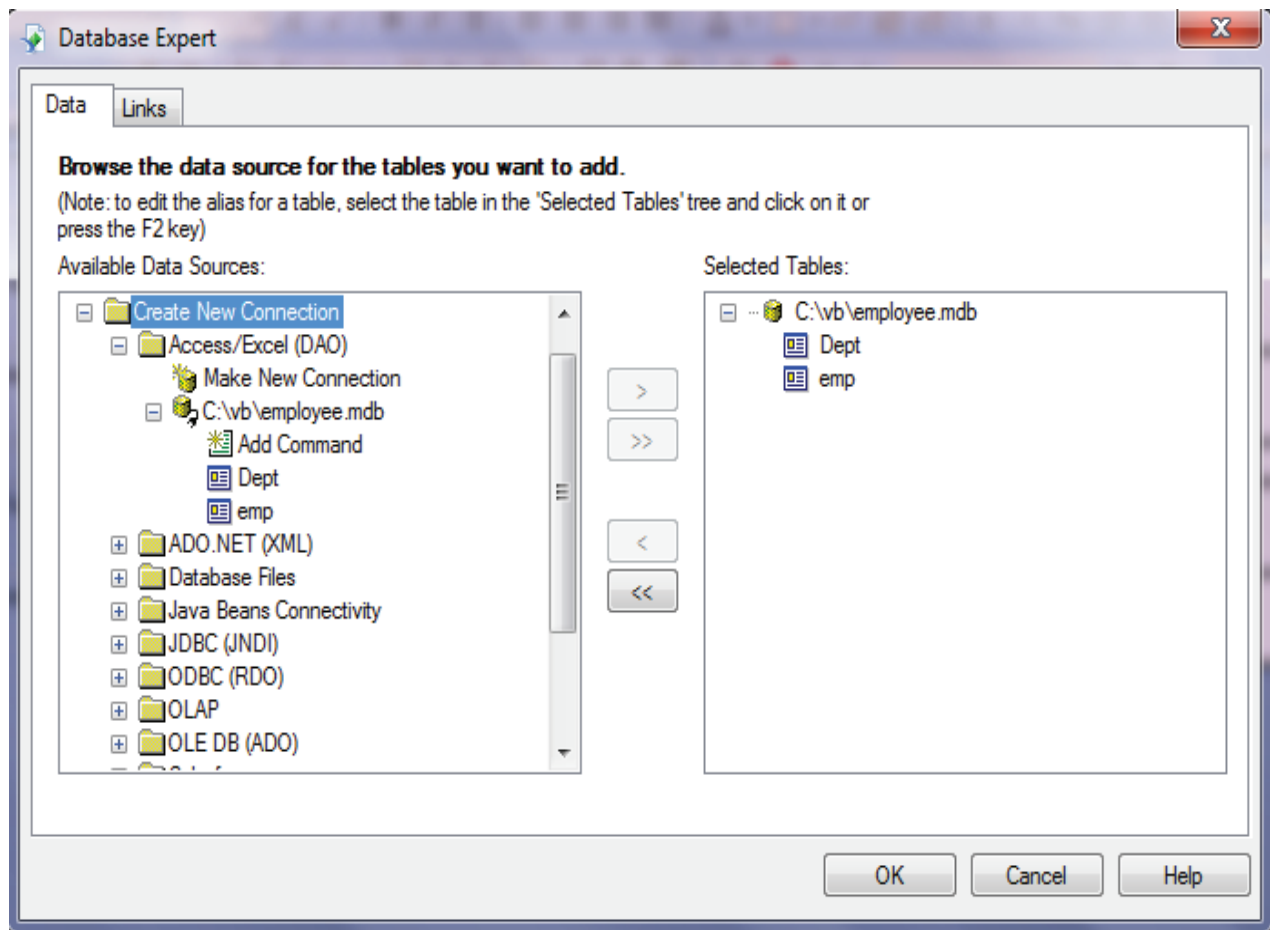
In the First window select the database on which report has to be made. To do so:

Click on Create New Connection' Select Access/Excel (DAO)' Make New Connection

A new window will be displayed(titled Access/Excel) as shown below. Select database name and click on Finish button.

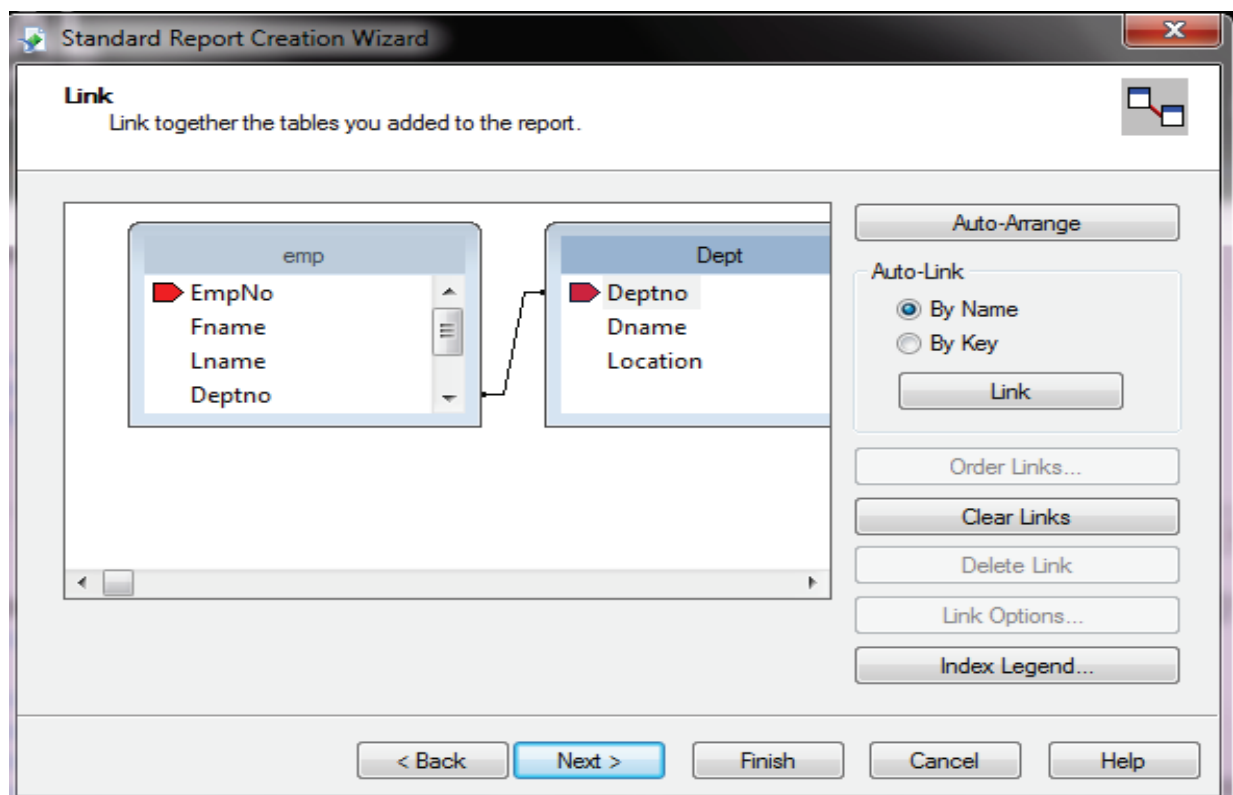


Now click on  button to add all the tables of the database or you can choose selected tables.

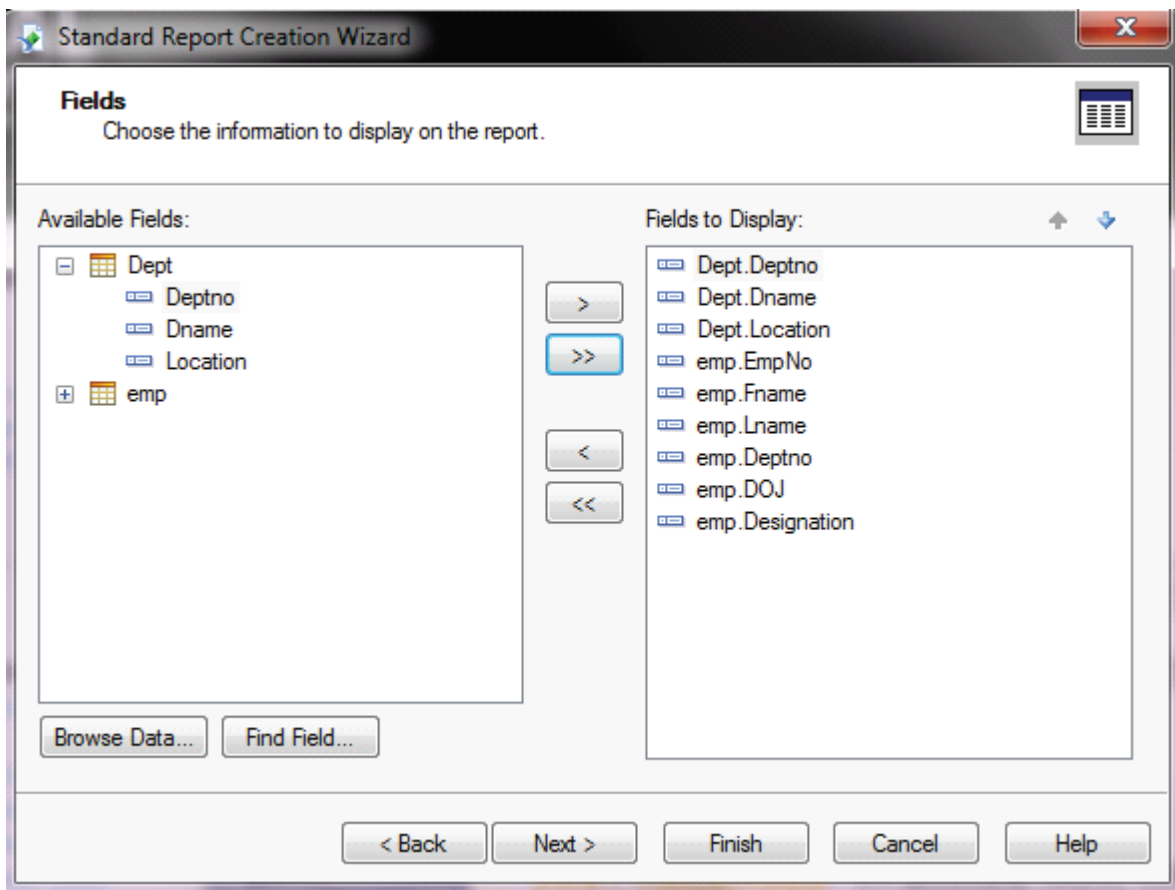


Click on OK button to complete the command.

Next window will show the relationship among different selected tables as follows:

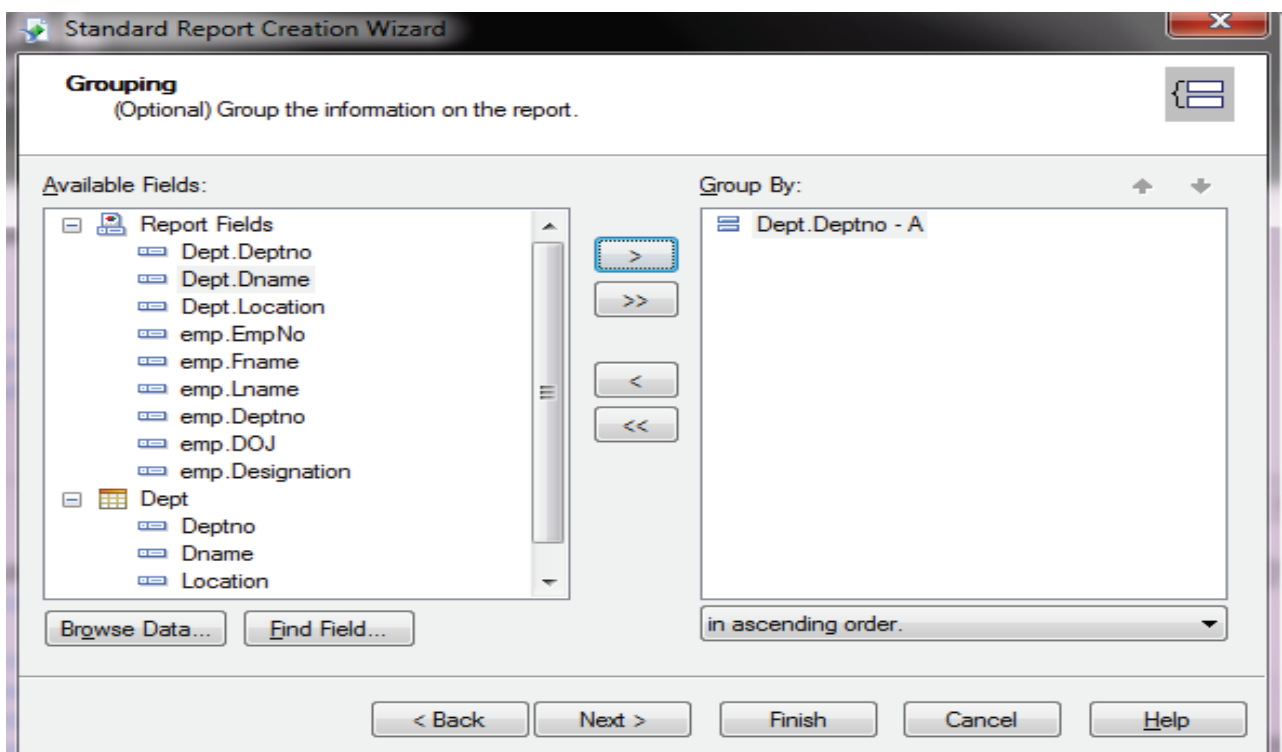


Click on Next button

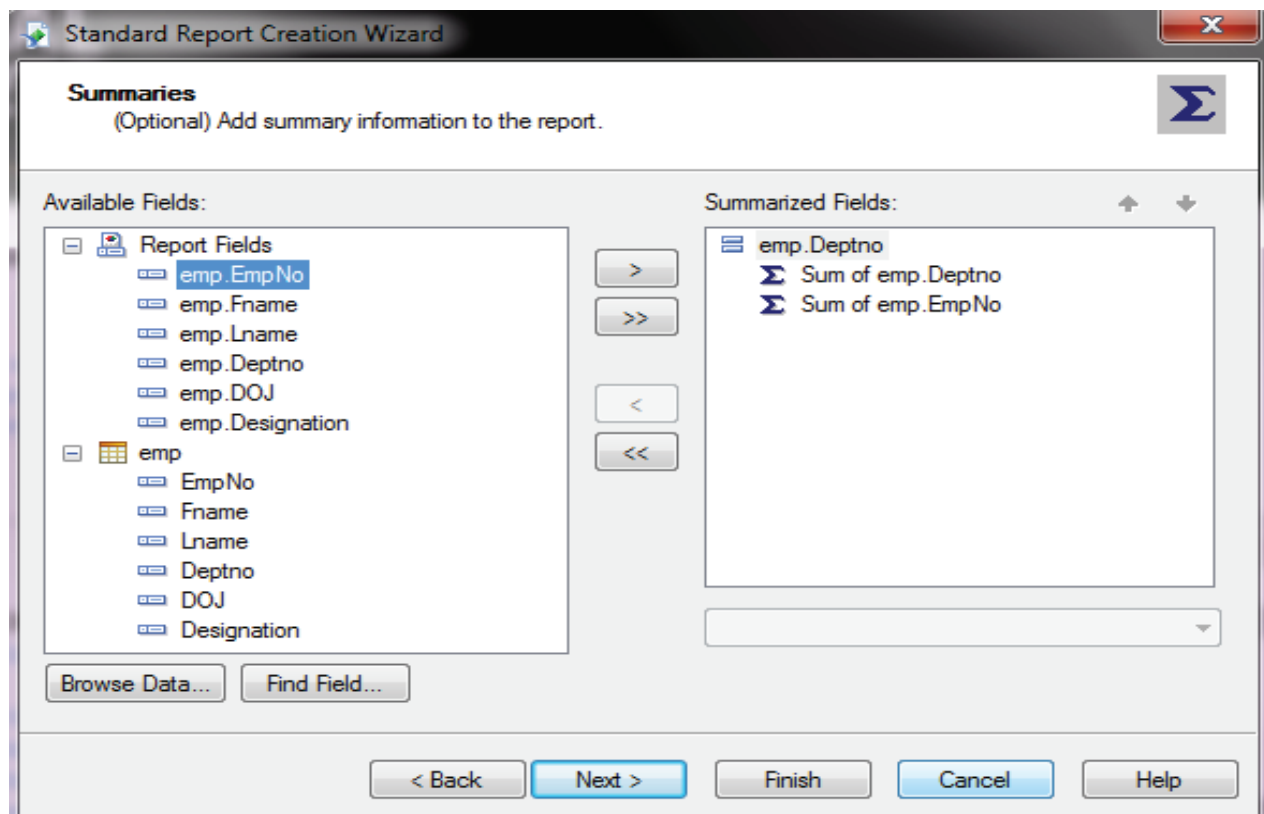


Select the fields that would be displayed the Report by selecting them in the left pane and click on  or  buttons. Click on Next button.

In next window, Select the field on which Grouping should be made. Select the field and move to the right pane by using  or  buttons

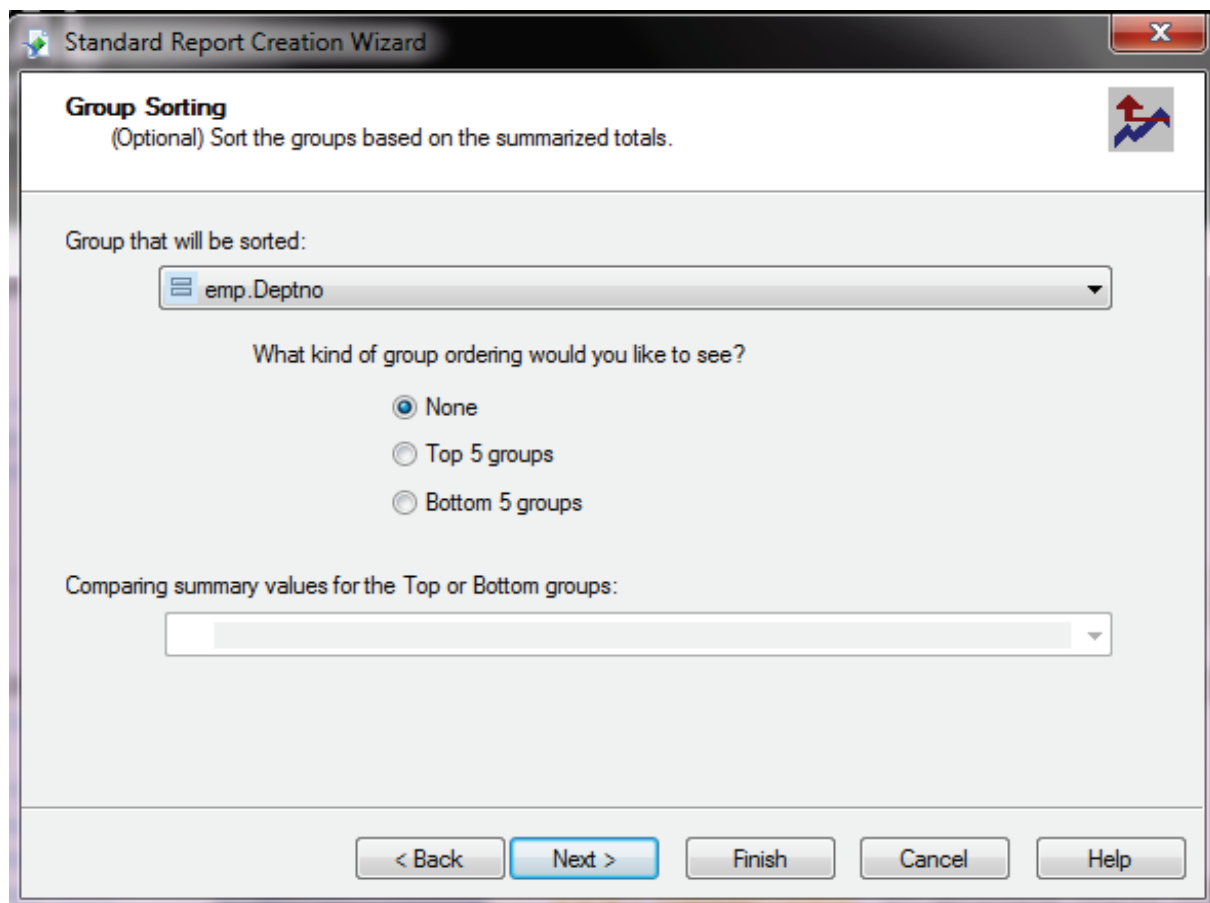


Click on Next button.



The 'Standard Report Creation Wizard' window is titled 'Summaries' with a subtitle '(Optional) Add summary information to the report.' It features a list of 'Available Fields' on the left, including 'Report Fields' (emp.EmpNo, emp.Fname, emp.Lname, emp.Deptno, emp.DOJ, emp.Designation) and 'emp' (EmpNo, Fname, Lname, Deptno, DOJ, Designation). In the center are navigation buttons: '>', '>>', '<', and '<<'. On the right, 'Summarized Fields' contains 'emp.Deptno', 'Σ Sum of emp.Deptno', and 'Σ Sum of emp.EmpNo'. At the bottom are buttons for '< Back', 'Next >', 'Finish', 'Cancel', and 'Help'.

In the next window, Select the aggregate function that will be performed. This is an Optional. Click on Next Button to view the next window.



The 'Standard Report Creation Wizard' window is titled 'Group Sorting' with a subtitle '(Optional) Sort the groups based on the summarized totals.' It shows 'Group that will be sorted:' as 'emp.Deptno'. Below this, it asks 'What kind of group ordering would you like to see?' with radio buttons for 'None' (selected), 'Top 5 groups', and 'Bottom 5 groups'. At the bottom, it says 'Comparing summary values for the Top or Bottom groups:' with a dropdown menu. Navigation buttons at the bottom include '< Back', 'Next >', 'Finish', 'Cancel', and 'Help'.

Standard Report Creation Wizard

Chart
(Optional) Include a chart on the report.

What kind of chart would you like to see?

☐ No Chart ☒ Bar Chart ☐ Line Chart ☐ Pie Chart

Chart title:
Sum of Deptno / Deptno

On change of:
emp.Deptno

Show summary:
Sum of emp.Deptno

< Back Next > Finish Cancel Help

Standard Report Creation Wizard

Template
(Optional) Select a template for the report.

Available Templates

- No Template
- Block (Blue)
- Corporate - Page Sections Only
- Corporate (Blue)**
- Corporate (Green)
- Double-Sided Page Headers/Footers
- Executive Summary or Title Page
- Form (Maroon)
- Gray Scale
- High Contrast
- Contrast Index
- Table Grid Template

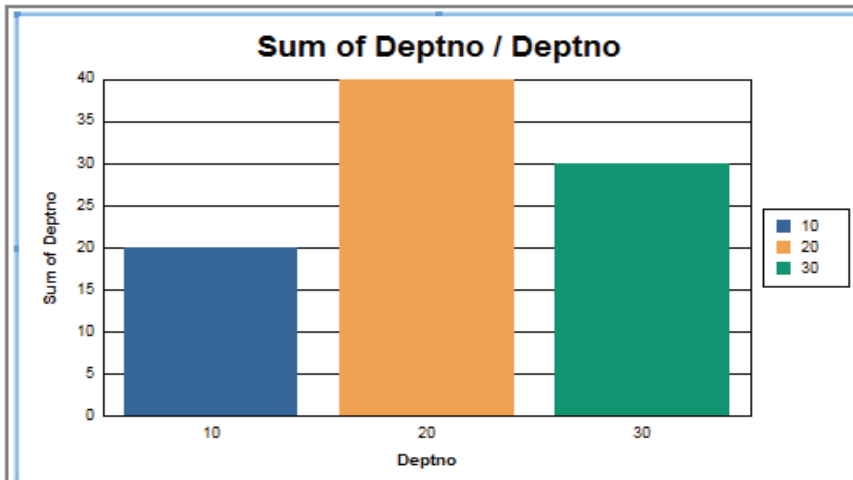
Preview

Corporate (Blue)

Yield For Available

Available	Yield For Available	Yield For Available, %
Available 1	1000000	1000000
Available 2	1000000	1000000
Available 3	1000000	1000000
Available 4	1000000	1000000
Available 5	1000000	1000000
Available 6	1000000	1000000
Available 7	1000000	1000000
Available 8	1000000	1000000
Available 9	1000000	1000000
Available 10	1000000	1000000
Available 11	1000000	1000000
Available 12	1000000	1000000
Available 13	1000000	1000000
Available 14	1000000	1000000
Available 15	1000000	1000000
Available 16	1000000	1000000
Available 17	1000000	1000000
Available 18	1000000	1000000
Available 19	1000000	1000000
Available 20	1000000	1000000
Available 21	1000000	1000000
Available 22	1000000	1000000
Available 23	1000000	1000000
Available 24	1000000	1000000
Available 25	1000000	1000000
Available 26	1000000	1000000
Available 27	1000000	1000000
Available 28	1000000	1000000
Available 29	1000000	1000000
Available 30	1000000	1000000
Available 31	1000000	1000000
Available 32	1000000	1000000
Available 33	1000000	1000000
Available 34	1000000	1000000
Available 35	1000000	1000000
Available 36	1000000	1000000
Available 37	1000000	1000000
Available 38	1000000	1000000
Available 39	1000000	1000000
Available 40	1000000	1000000
Available 41	1000000	1000000
Available 42	1000000	1000000
Available 43	1000000	1000000
Available 44	1000000	1000000
Available 45	1000000	1000000
Available 46	1000000	1000000
Available 47	1000000	1000000
Available 48	1000000	1000000
Available 49	1000000	1000000
Available 50	1000000	1000000
Available 51	1000000	1000000
Available 52	1000000	1000000
Available 53	1000000	1000000
Available 54	1000000	1000000
Available 55	1000000	1000000
Available 56	1000000	1000000
Available 57	1000000	1000000
Available 58	1000000	1000000
Available 59	1000000	1000000
Available 60	1000000	1000000
Available 61	1000000	1000000
Available 62	1000000	1000000
Available 63	1000000	1000000
Available 64	1000000	1000000
Available 65	1000000	1000000
Available 66	1000000	1000000
Available 67	1000000	1000000
Available 68	1000000	1000000
Available 69	1000000	1000000
Available 70	1000000	1000000
Available 71	1000000	1000000
Available 72	1000000	1000000
Available 73	1000000	1000000
Available 74	1000000	1000000
Available 75	1000000	1000000
Available 76	1000000	1000000
Available 77	1000000	1000000
Available 78	1000000	1000000
Available 79	1000000	1000000
Available 80	1000000	1000000
Available 81	1000000	1000000
Available 82	1000000	1000000
Available 83	1000000	1000000
Available 84	1000000	1000000

222



1/12/2012

<u>Deptno</u>	<u>EmpNo</u>	<u>Fname</u>	<u>Lname</u>	<u>DOJ</u>	<u>Designation</u>
10					
10	2	Seema	Jain	10/1/1981 12:00:00AM	LDC
10	1	Rajesh	Sharma	12/2/1980 12:00:00AM	UDC
10	20.00	3.00			
20					
20	4	Shradha	Sharma	2/2/1978 12:00:00AM	Salesman
20	3	Sangeeta	Meena	1/2/1980 12:00:00AM	Manager

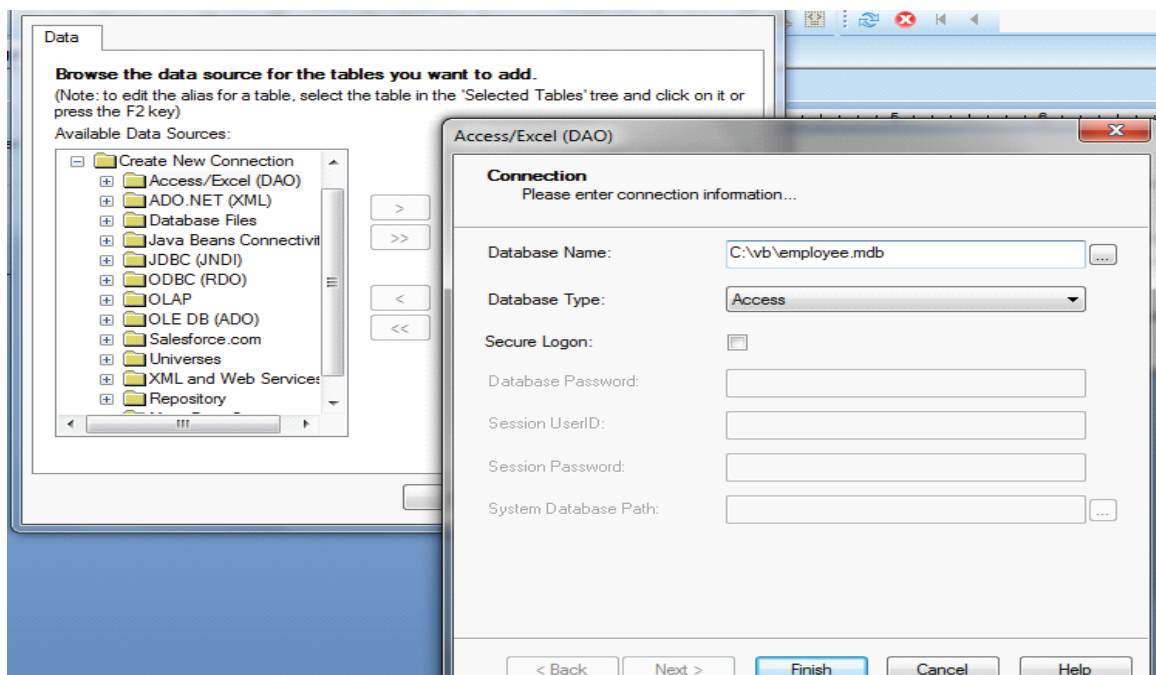
14.3.2 Creating Reports By Design Layout


Click on File → New → Blank Report

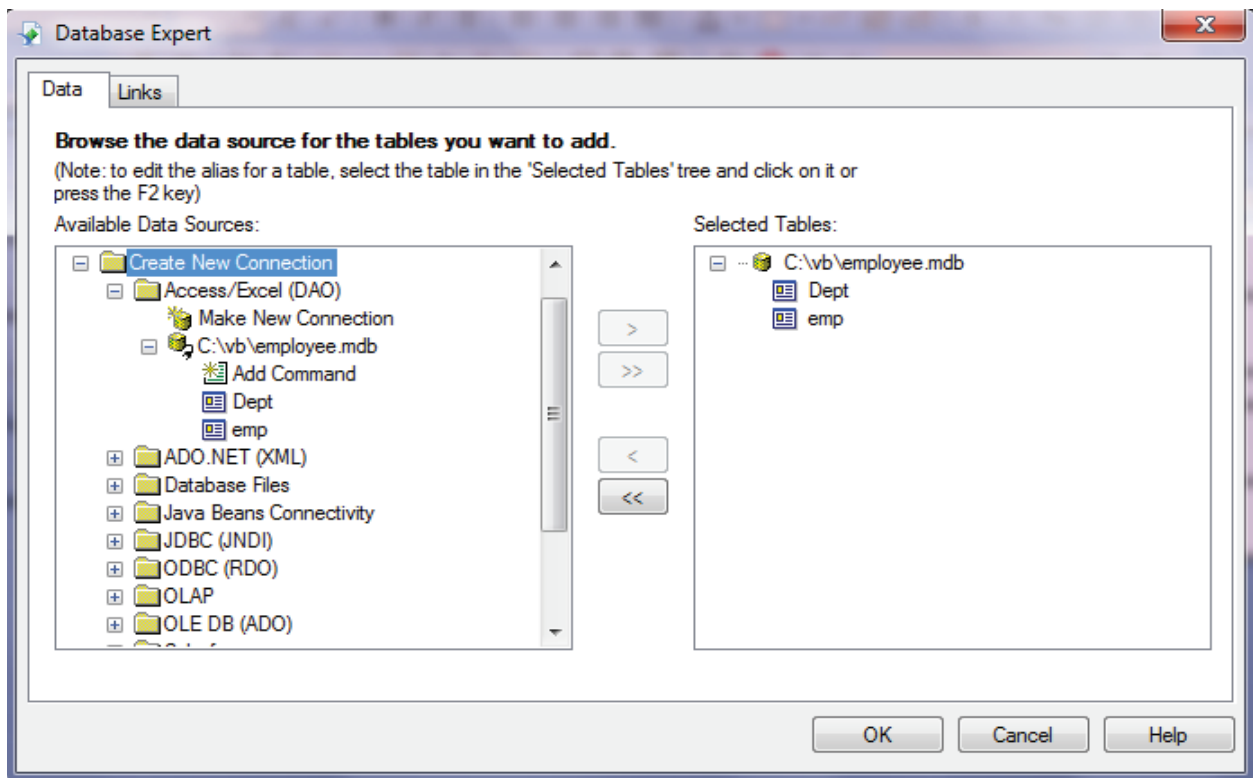
In the First window select the database on which report has to be made. To do so:

Click on Create New Connection → Select Access/Excel (DAO) → Make New Connection

A new window will be displayed (titled Access/Excel) as shown below. Select database name and click on Finish button.

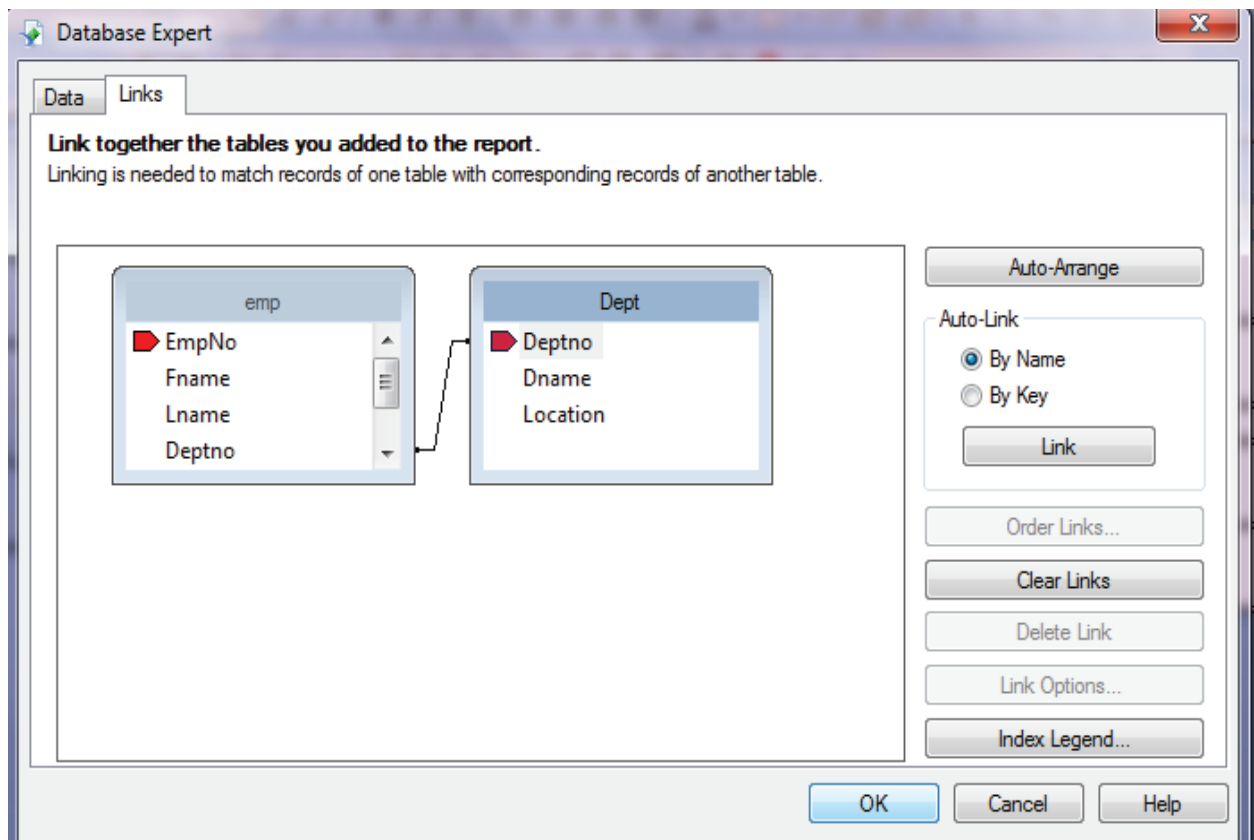


Now click on  button to add all the tables of the database or you can choose selected tables.



Click on OK button to complete the command.

Next window will show the relationship among different selected tables as follows:



To remove the current links click on Clear Links Button and to Create new links click on Link Button. Click OK to complete the operation. Next window will be the blank report layout in which you can design your report.

14.4 Different Sections of Reports

As shown below there are basically five sections of any report named:

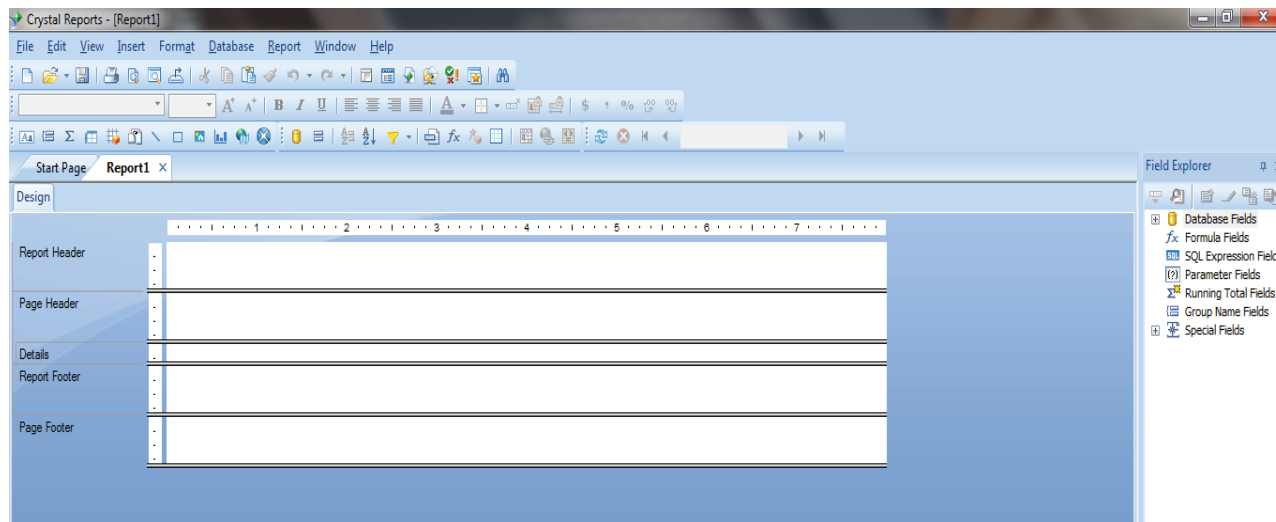
Report Header: Add text/ Data that should be displayed on Starting of Report like Report Heading, Date of Creation, and Creator's Name etc.

Page Header: The data that should be displayed on each page of report like Column Heading etc.

Details: Actual Reports' component should be added in this

Page Footer: This Section contains the text/data that should be displayed at the bottom of every report page like page number etc.

Report Footer: At the end of the complete report what should be displayed, must be added in this Section like End Of Report.

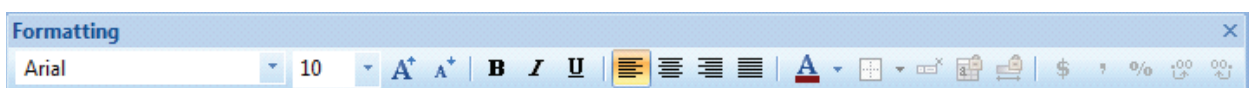


14.5 To Add Report Header



Click on First button of Insert Tools (situated at the left side of report window) and mark the area in the report header portion. Write the text that should be displayed as Report Header. To format the report header Click either on Format Menu or make use of formatting toolbar.

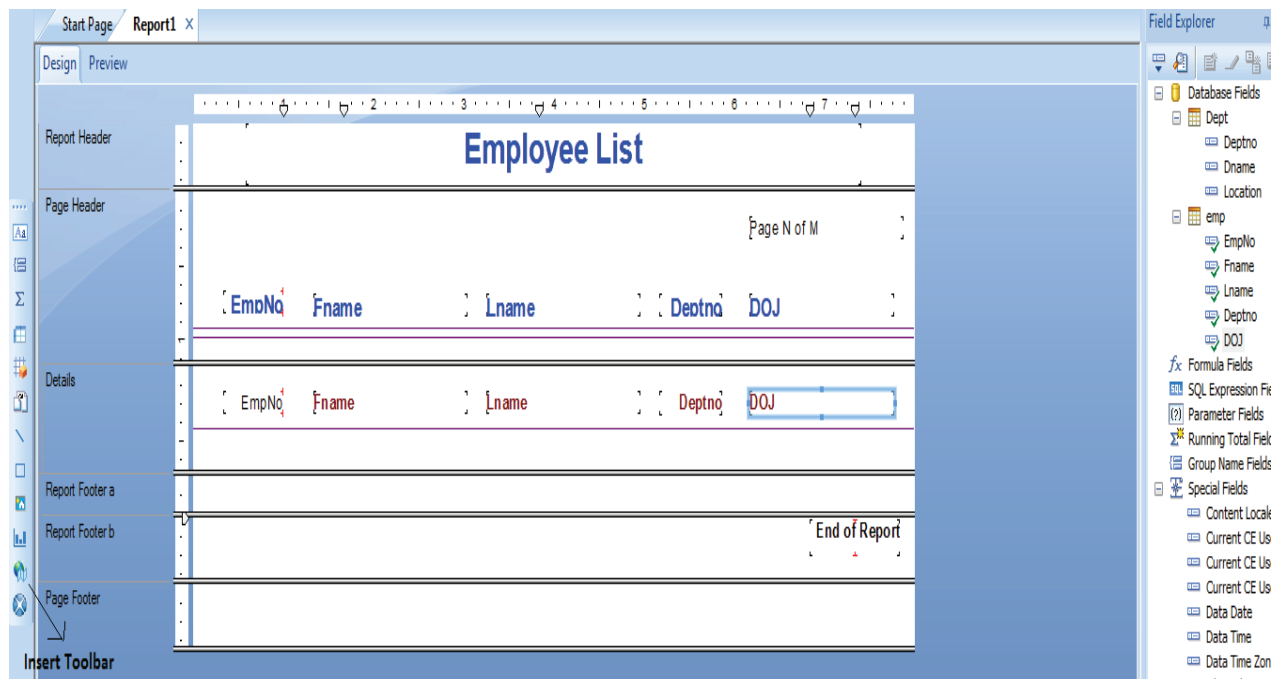
To add lines , Charts Rectangles . Use appropriate buttons from the Insert Tools. Toolbar.



To add lines , Charts Rectangles . Use appropriate buttons from the Insert Tools. Toolbar.

14.6 To add Page Header/ Field Headers

On the right side you will see field explorer window. Select Database Fields' Select Appropriate Table' Select Field(s)' Click Right Mouse Button' Select add to report' Drag and Paste in the details area of report. Field Name will be displayed in the Page header and details portion will contain actual data (as shown below).



To add page number, total number of pages, current Date etc. Click on Special Fields' Select appropriate field and drag and Paste in the appropriate report Section.

To view actual report Click on View menu' Preview.


Employee List

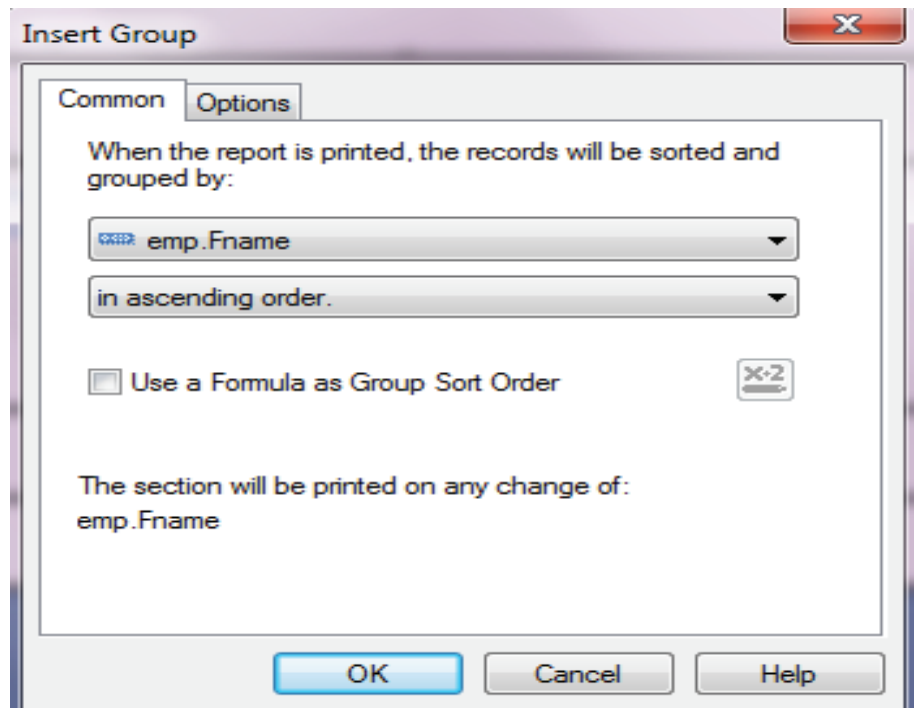
Page 1 of 1

EmpNo	Fname	Lname	Deptno	DOJ
1	Rajesh	Sharma	10	12/2/1980 12:00:00AM
2	Seema	Jain	10	10/1/1981 12:00:00AM
3	Sangeeta	Meena	20	1/2/1980 12:00:00AM

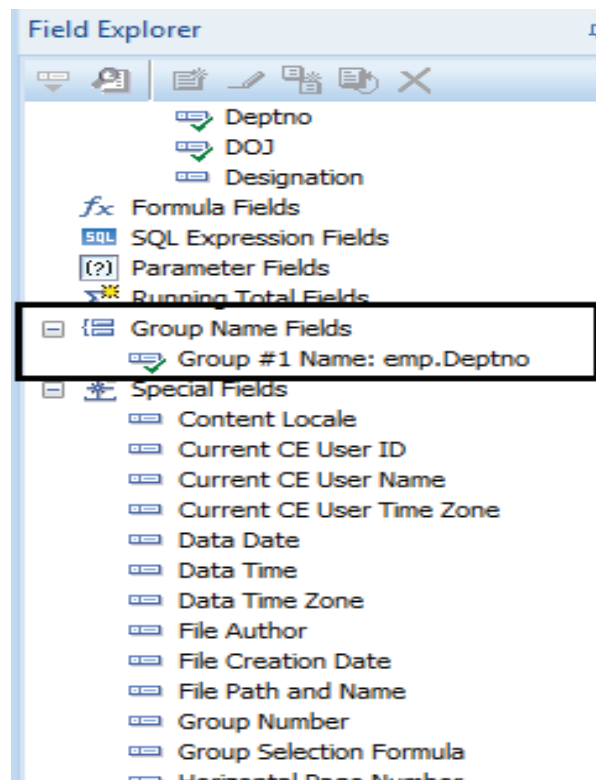
End of Report

14.7 Inserting Groups


To insert Groups Click on Insert Group Button of Insert Tools  Insert Group . the new dialog will be displayed. Select field on which grouping has to be made and also select the order (Ascending/descending).



As soon as you click on OK button, two new sections will be appeared named Group Header And Group Footer. By default group header will contain Group Name field. Or you can add by selecting group Name field of Field Explorer (As shown in the following window).



14.8 Inserting Totals and Grand Totals

To insert calculated field that can work on aggregated data like complete report data or data of one group, you can click on Insert Summary field of Insert  **Insert Summary** of Insert Tools. This the third button from the top.

Select Field to be summarize i.e., the field on which calculation has to be performed for example if you want to calculate total number of employees in the organization or department wise total number of employees select field Empno. If you want to calculate maximum salary department-wise, select sal field.

Insert Summary

Choose the field to summarize:
 emp.EmpNo

Calculate this summary:
 Sum

Summary location
 Grand Total (Report Footer)

☐ Add to all group levels Insert Group...

Options
☐ Show as a percentage of
☐ Summarize across hierarchy

OK Cancel Help

After this, select the function that has to be performed in the Calculate this summary field like sum, count, max, min etc.

Insert Summary

Choose the field to summarize:
 emp.EmpNo

Calculate this summary:
 Sum
 Average
 Sample variance
 Sample standard deviation
 Maximum
 Minimum
 Count
 Distinct count
 Correlation with
 Covariance with
 Median
 Mode
 Nth largest, N is:
 Nth smallest, N is:
 Nth most frequent, N is:
 Pth percentile, P is:
 Population variance
 Population standard deviation
 Weighted average with

OK Cancel Help

Summary Location is used to specify whether to display the summary information at the end of report or in the Group Footer Section. If you want to add the summary information at both locations, select Add all group levels.

Insert Summary

Choose the field to summarize:
 emp.EmpNo

Calculate this summary:
 Sum

Summary location
 Grand Total (Report Footer)
 Grand Total (Report Footer)
 Group #1: emp.Deptno - A

Options
☐ Show as a percentage of
☐ Summarize across hierarchy

OK Cancel Help

Click OK to complete the task. The design layout will be as follows:

Design Preview

Report Header

Employee List

Page Header

Page N of M

Insert Summary

Group Header #1:
emp.Deptno - A

Dept Number : Group #1 Name

Details

EmpNo Fname Lname DOJ

Group Footer #1:
emp.Deptno - A

Total Number of Employees: int of emp.EmpNo

Report Footer a

Report Footer b

End of Report

Page Footer

Employee List

Page 1 of 1

EmoNo	Fname	Lname	DOJ
Dept Number : 10			
2	Seema	Jain	10/1/1981 12:00:00AM
1	Rajes h	Sharma	12/2/1980 12:00:00AM
Total Number of Employees:			2
Dept Number : 20			
4	Shradha	Sharma	2/2/1978 12:00:00AM
3	Sangeeta	Meena	1/2/1980 12:00:00AM
Total Number of Employees:			2
Dept Number : 30			

14.9 How to Connect Crystal Report to VB Application

To display a report in your VB application you will need the following:

References:

Crystal Reports <Version you are using> ActiveX Designer Run Time Library

To Add these Click on

Project' →References'→Crystal Reports ActiveX Designer Run Time Library

Components:

Crystal Reports<Version you are using> Viewer Control

To Add these Click on

Project' →component' →Crystal Reports Viewer Control

Now put the Crystal reports viewer control on your form. Set Property ReportFileName=<Name of rpt File with Complete Path>

To show report

Set action property of Crystal Report control to 1

For example:

Crystalreport1.ReportFileName="Emp.rpt"

Crystalreport1.action=1

14.10 Summary

- Crystal Reports provides a relatively easy way to incorporate reporting into your application.
- Report can be created either by using Report wizard or manually using Design layout.
- There are basically five sections of any report named; report header, page header, detail, page footer and report footer.
- Report Header is used to add text/ Data that should be displayed on Starting of Report like Report Heading, Date of Creation, and Creator's Name etc.
- Page Header is used to add The data that should be displayed on each page of report like Column Heading etc.

- Actual Reports' component should be added in the details section.
- Page Footer contains the text/data that should be displayed at the bottom of every report page like page number etc.
- At the end of the complete report what should be displayed, must be added in Report footer section like End of Report.
- To display a report in your VB application you will need the following: Crystal Reports <Version you are using> ActiveX Designer Run Time Library and Crystal Reports<Version you are using> Viewer Control

14.11 Self Assessment Questions

1. Explain different Sections of Crystal Report.
2. What is the basic purpose of using Crystal Report?
3. Explain different steps to create crystal report using wizard.
4. How do you insert groups in Crystal Report? Explain
5. Explain the complete procedure of adding functions in Crystal Report.

Unit - 15 : Data Files

Structure of Unit:

- 15.0 Objectives
- 15.1 Introduction
- 15.2 Different File Mode
 - 15.2.1 Text Mode
 - 15.2.2 Binary Mode
 - 15.2.3 Random Mode
- 15.3 Sequential File Access
- 15.4 Binary File Access
- 15.5 Random File Access
- 15.6 Summary
- 15.7 SelfAssessment Questions

15.0 Objectives

This chapter provides a general overview of

- Different File Mode like Text Mode, Binary Mode, Random Mode
- Sequential File Access
- Binary File Access
- Random File Access

15.1 Introduction

There are different modes for handling files in VB6.0. Files can be stored as text files, binary files or random files. To access and manage these files VB provides different methods and statement. With the help of which you can easily perform functions like reading, writing and accessing contents of the files.

15.2 Different File Mode

In VB6.0, there are three modes in which a file can be accessed.

- Text Mode (Sequential Mode)
- Random Access Mode
- Binary Mode

15.2.1 Text Mode

In the Text Mode, data is always written and retrieved as CHARACTERS. Hence, any number written in this mode will result in the ASCII Value of the number being stored.

For Example, The Number 27 is stored as two separate characters “2” and “7”. Which means that 27 is stored as [50 55] and not as [27].

Text Files support Sequential Reading and Writing. This means that we cannot read or write from a particular point in a file. The only way of doing this is to read through all the other entries until you reach the point where you want to ‘actually’ start reading

15.2.2 Binary Mode

In the **Binary Mode**, everything is written and retrieved as a Number. Hence, The Number 27 will be stored as [27] in this mode and characters will be represented by their ASCII Value as always. Text Files support random Reading and Writing. It allows us to write and read anywhere in the file. For example we can read data directly from the 45th Byte of the file, instead of reading all the bytes one by one till we reach 45.

15.2.3 Random Mode

Just like the Binary Mode, the Random Access Mode allows us to gain instant access to any piece of information lying anywhere in the file. In this case, we must specify name and size of each piece of information.

For example, if we need to store a few names in the file RandomAccess Mode

requires us to mention the length of the ‘Names’ Field. RandomAccess Mode allows us to read or write data at a particular record position rather than a byte position like in Binary Mode.

15.3 Sequential File Access

The following sequential file-related statements and functions:

Open	Prepares a file to be processed by the VB program.
FreeFile	Supplies a file number that is not already in use
Input #	Reads fields from a comma-delimited sequential file
EOF	Tests for end-of-file
Write #	Writes fields to a sequential file in comma-delimited format
Print #	Writes a formatted line of output to a sequential file
Close #	Closes a file

The Open Statement

Open statement is used to open any file. The full syntax for the Open statement is:

Open *pathname* **For** *mode* [**Access** *access*] [*lock*] **As** [#]*filenumber* [**Len=***reclength*]

The **Open** statement syntax has these parts:

Part	Description
<i>pathname</i>	Required. String expression that specifies a file name — may include directory or folder, and drive.
<i>mode</i>	Required. Keyword specifying the file mode: Append , Binary , Input , Output , or Random . If unspecified, the file is opened for Random access.

<i>access</i>	Optional. Keyword specifying the operations permitted on the open file: Read , Write , or Read Write .
<i>lock</i>	Optional. Keyword specifying the operations restricted on the open file by other processes: Shared , Lock Read , Lock Write , and Lock Read Write .
<i>filenumber</i>	Required. A valid file number in the range 1 to 511, inclusive. Use the FreeFile function to obtain the next available file number.
<i>reclength</i>	Optional. Number less than or equal to 32,767 (bytes). For files opened for random access, this value is the record length. For sequential files, this value is the number of characters buffered.

Note: For sequential file access we can use either of three modes *Input*, *Output* or *Append*.

Examples:

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

We can specify the absolute path of the file being opened or we can use app.path to specify the path of the file existing in the same folder of current VB application (if both program and the data file reside in the same folder). App.Path specifies the path of the current VB program.

For example if we want to open a file named Employee.dat of current folder, the code would be

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

The FreeFile Function

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

```
Dim FP As Integer
```

```
FP = FreeFile
```

In the Open statement, use the integer variable rather than the hard-coded number. For example:

```
Open strFileName For Input As #FP
```

The Input # Statement

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

Input #<filenumber>, <variable list>

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

The EOF function

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

The syntax of the EOF function is

EOF(*num*)

where *num* is a number corresponding to the file number of the file from which we want to read data. *num* can either be a hard-coded number or an integer variable, depending on whether or not we used FreeFile in the Open statement.

The Close statement

When we are finished using a file in your program, we should Close that file. The Close statement tells VB that we are done using a file, and frees up the system resources needed to process that file.

Syntax of Close Statement is

Close #*num*

where *num* is a number corresponding to the file number of the file from which we want to read data. *num* can either be a hard-coded number or an integer variable, depending on whether or not we used FreeFile in the Open statement.

Example

The statement Close #1

frees the resources used by the file referenced as #1, and also terminates the association between the Windows-system file and the file number #1.

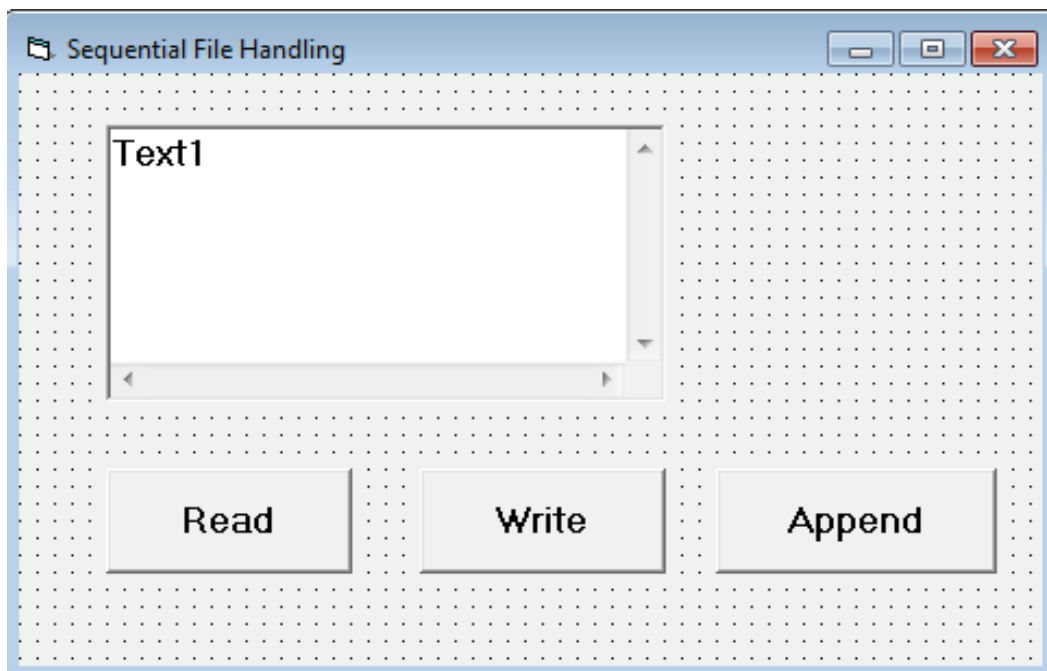
If we have more than one file open in a program, we can close multiple files with one Close statement by separating the file numbers with commas:

Close #1, #2, #68

Example

Click on Start → All Programs → Microsoft Visual Studio → Microsoft Visual Basic6.0

Design the Form as the following:

The image shows a screenshot of a Visual Basic 6.0 form titled "Sequential File Handling". The form is set against a dotted grid background. It contains a text box at the top left labeled "Text1". Below the text box, there are three buttons arranged horizontally: "Read", "Write", and "Append". Each button is a light gray rectangle with a thin border and a slight 3D effect. The window has a standard Windows 95-style title bar with minimize, maximize, and close buttons.

Write the following code

Option Explicit

Dim fp As Integer

```
Private Sub Read_Click()  
    Text1=""  
    Open "stud.txt" For Input As fp  
    Text1 = Input(LOF(fp), fp)  
    Close fp  
End Sub
```

```
Private Sub Write_Click()  
    Open "stud.txt" For Output As fp  
    Print #fp, Text1  
    Close fp  
    Text1=""  
End Sub
```

```
Private Sub Append_Click()  
    Open "stud.txt" For Append As fp  
    Print #fp, Text1  
    Close fp  
    Text1=""  
End Sub
```

```
Private Sub Form_Load()  
    fp = FreeFile  
End Sub
```

Now Click F5 key and write text in the textbox. Click on Write Button to save the textbox content to the File named 'stud.txt'.

To read the contents of the file

Click on Read Button.

To add more Contents

Write contents to the textbox

Click on Append Button

15.4 Binary File Access

The Open Statement

Syntax:

Open *filename with pathname* For Binary As #*filename*

If the file specified by *pathname* doesn't exist, it is created when a file is opened for **Binary** modes.

If the file is already opened by another process and the specified type of access is not allowed, the **Open** operation fails and an error occurs.

The **Len** clause is ignored if *mode* is **Binary**.

The Get Statement

The **Get** statement is used read data from a file opened in binary mode. The syntax, as it applies to binary files is:

Get [#]*filename*, [*byte position*], *varname*

- The *filename* is any valid filename as defined above.
- *Byte position* is the byte position within the file at which the reading begins
- *Varname* is a string variable into which the data will be read. This string variable is often referred to as a "buffer" when processing binary files. It is important to note that the length, or size, of this string variable determines how many bytes of data from the file will be read. Thus, it is necessary to set the length of the string variable prior to issuing the Get statement.

Example.

To read first 10 bytes code would be

```
Dim st as String*10
```

```
Get #FP,1, st
```

The Input Function

The *Input function* can be used as an alternative to the Get statement. The syntax is:

***varname* = Input(*number*, [#] *filename*)**

where

- *varname* is the string variable into which the file data will be stored
- *number* is the number of characters to be read
- *filename* is a valid filename identifying the file from which you want to read.

Example

```
st = Input(10, #FP)
```

```
st = Input(LOF(intMyFile), #intMyFile) ' To read whole file
```

The Put Statement

The **Put** statement is used write data to a file opened in binary mode.

The syntax is:

Put [#]*filename*, [*byte position*], *varname*

- The *filenumber* is any valid filenumber as defined above.
- *Byte position* is the byte position within the file at which the writing begins.
- *Varname* is a string variable from which the data will be written.

For example, the following statements write 10 bytes of data to file number “FP” at the 50th byte position:

```
Dim st as String*10
```

```
St=Inputbox(“Enter Text to be written”)
```

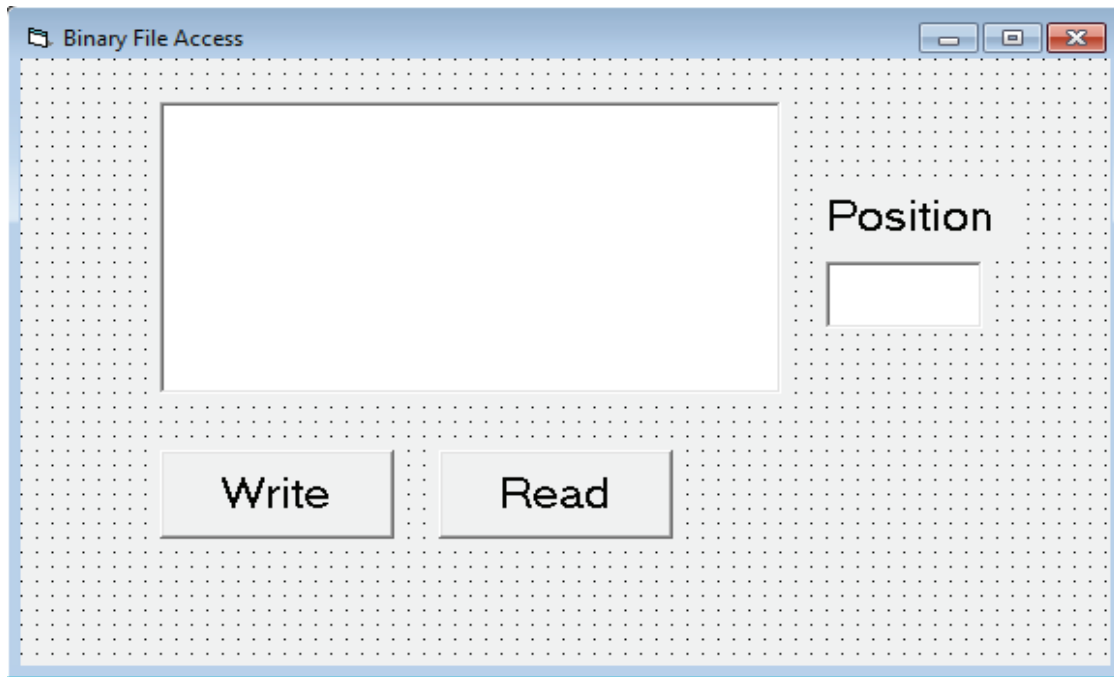
```
Put #FP,50, st
```

Example:

Design the following form in VB6.0

Text1

Text2



Write the Following code in code window

```
Option Explicit
```

```
Dim fp As Integer
```

```
Private Sub Write_Click()
```

```
Text1=""
```

```
Put #fp, Val(Text2), Text1.Text
```

```
End Sub
```

```
Private Sub Read_Click()
```

```
Dim st As String * 20
```

```
Get #fp, Val(Text2), st
```

```
Text1 = st  
End Sub
```

```
Private Sub Form_Load()  
fp = FreeFile  
Open "binary_file" For Binary As fp  
End Sub
```

Program Execution

- ✓ Now Click F5 Key to execute the file.
- ✓ To write data in the file:
 - Write data in the textbox(Text1) to be written
 - Write down byte position in the position textbox(Text2)
 - Click on Write button
- ✓ To Read data from the file:
 - Write down byte position in the position textbox(Text2)
 - from where reading should be started. Maximum 20 characters can be read as the size of string st is 20.
 - Click on Read button

15.5 Random File Access

Random files are record-based files with an internal structure that supports “direct access” by record number. This means that your program can read from or write to a specific record in a random access file, say the 35th record, without reading through the previous 34 records.

Opening a file

Syntax:

Open *filename* For Random As #*filenumber* Len = *reclength*

Note: To get the length of the record use function Len function.

The Get Statement

The **Get** statement is used read data from a file opened in random mode. The syntax, as it applies to random files is:

Get [#]*filenumber*, [*recnumber*], *varname*

- The *filenumber* is any valid filenumber.
- *Recnumber* is the record position within the file that is read.
- *Varname* is the record variable into which the data will be read. We can use Type keyword to define record structure.

Syntax:

```
Type <Type name>  
    Member declaration with size  
End Type
```

Example:

```
Type student  
    Roll As String * 5  
    Nm As String * 25  
End Type
```

To declare variable of type student use the following syntax:

```
Dim <varname> as <recordtype>
```

Example:

```
Dim e1 As student
```

The Put Statement

The **Put** statement is used write data to a file opened in random mode.

Syntax:

```
Put [#]filename, [recnumber], varname
```

- The *filename* is any valid filename.
- *Recnumber* is the record position within the file which is written.
- *Varname* is the record variable from which the data will be written. The record variable is a variable defined on the User Defined Type record structure as described above.

Example:

Right Click on Project1 (In Project Explorer Window) Add→Module

In module window

Write the following code:

```
Type student  
    Roll As String * 5  
    Nm As String * 25  
End Type
```

Kill Statement

Kill statement is used to delete any file.

Syntax:

```
Kill "<filename with path as string>"
```

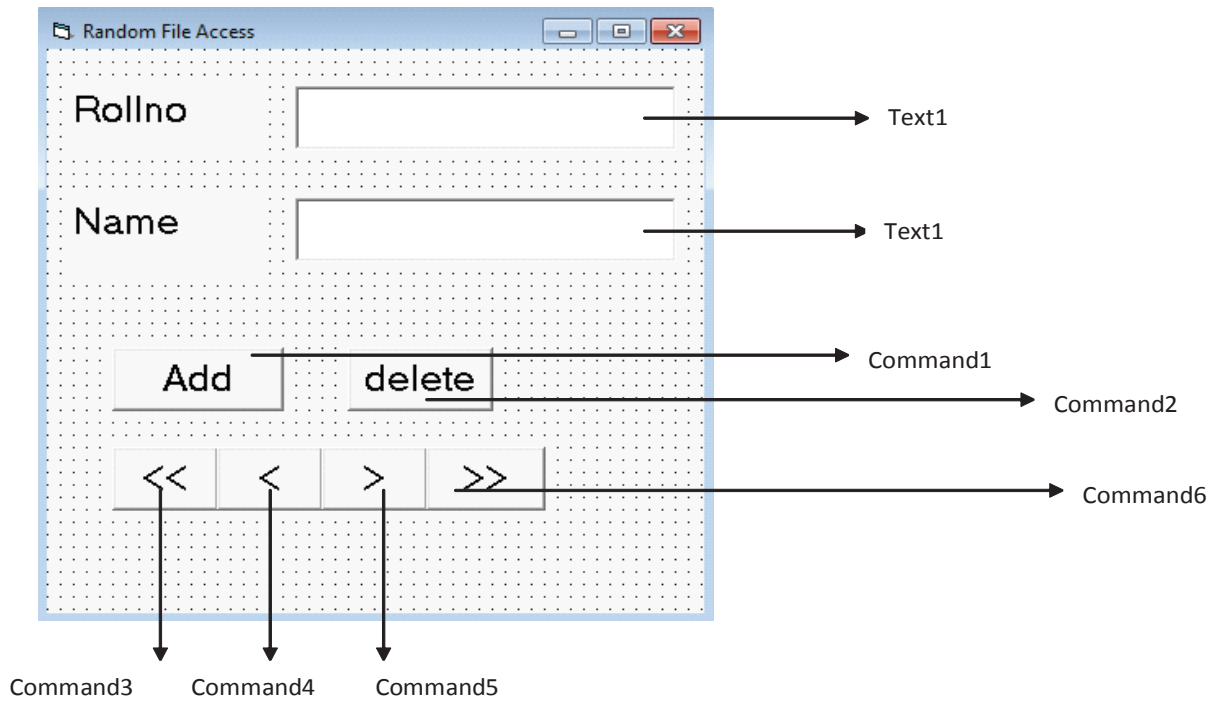
FileLen() Function

Filelen function is used to get the size of the file.

Syntax:

```
FileLen("<Filename>") and returns size of the file in integer form.
```

Design the following Form in VB6.0



Write the code in the code window

Option Explicit

Dim e1 As student

Dim rec_len As Integer

Dim fp As Integer, lst_rec, curr

Private Sub Command1_Click()

 lst_rec = lst_rec + 1

 e1.nm = Text2

 e1.roll = Text1

 Put #fp, lst_rec, e1

 curr = lst_rec

 Text1 = ""

 Text2 = ""

End Sub

Private Sub Command2_Click()

 Kill "stud.txt" ' to delete file

End Sub

Private Sub Command3_Click()

 curr = 1

 Get #fp, curr, e1

```
Text1 = Trim(e1.roll)
Text2 = Trim(e1.nm)
End Sub
```

```
Private Sub Command4_Click()
    curr = curr - 1
    If curr = 0 Then
        MsgBox "Beginning of file"
        Exit Sub
    End If
    Get #fp, curr, e1
    Text1 = Trim(e1.roll)
    Text2 = Trim(e1.nm)
End Sub
```

```
Private Sub Command5_Click()
    curr = curr + 1
    If curr > FileLen("stud.txt") Then
        MsgBox "end of file"
        Exit Sub
    End If
    Get #fp, curr, e1
    Text1 = Trim(e1.roll)
    Text2 = Trim(e1.nm)
End Sub
```

```
Private Sub Command6_Click()
    curr = lst_rec
    Get #fp, curr, e1
    Text1 = Trim(e1.roll)
    Text2 = Trim(e1.nm)
End Sub
```

```
Private Sub Form_Load()
    fp = FreeFile
    rec_len = Len(e1)
    Open "stud.txt" For Random As fp Len = rec_len
    lst_rec = FileLen("stud.txt")
    curr = lst_rec
End Sub
```

Program Execution:

- Now press F5 key to execute the program.
- Initially as there are no records in the file, so first enter rollno and name in the appropriate textbox.
- Click on Add Button.
- Again type the values in rollno and name textbox.
- Click on Add Button.
- Now you can use navigation buttons (Command3 i.e. First Record, Command4 i.e. Previous Record, Command5 i.e. Next Record and Command6 i.e., Last record) to view all records.

15.6 Summary

- In VB6.0, there are three modes in which a file can be accessed i.e., Text Mode (Sequential Mode), Random Access Mode and Binary Mode.
- In the Text Mode, data is always written and retrieved as CHARACTERS.
- In the Binary Mode, everything is written and retrieved as a Number.
- the Random Access Mode allows us to gain instant access to any piece of information lying anywhere in the file.
- Open statement is used to open any file.
- The Close statement tells VB that we are done using a file, and frees up the system resources needed to process that file.
- The Get statement is used read data from a file opened in binary mode or random mode.
- The Input function can be used as an alternative to the Get statement.
- The Put statement is used write data to a file opened in binary mode or random mode.
- Random files are record-based files with an internal structure that supports “direct access” by record number.

15.7 Self Assessment Questions

1. Differentiate among Sequential, Binary and Random File Access modes.
2. Explain Any Three functions or statements used in Sequential File handling.
3. Write a program to demonstrate Binary File handling.
4. Explain the Following:
 - a. Get Statement
 - b. Put Statement
 - c. FileLen function
 - d. Kill function

Reference Books

- Beginning Visual Basic 6 *By Peter Wright*
- *Mastering Visual Basic 6 By Evangelos Petroutsos*
- *Visual Basic 6.0 By Peter Norton*