



Bharat Ratna Puratchi Thalaivar Dr.MGR Government Arts and Science College, Palacode – 636808

B.Sc. COMPUTER SCIENCE SEMESTER

SEMESTER V

CORE VI - GUI PROGRAMMING

UNIT - I

Introduction to Visual Basic: Programming Languages, Procedural, Object Oriented and Event Driven, Writing VB Projects, VB Environment, Writing First Project-Finding and Fixing Errors - VB Help. Controls in VB - Coding for Controls.

UNIT - II

Variables, Constants and Calculations: Data Types-Variables and Constants - Val function - Arithmetic operations - Formatting data. Decisions and conditions: If Statements - Conditions - nested If Statements - Using If Statement with option buttons and checkboxes - displaying messages - Input validation - Calling event Procedures.

UNIT - III

Menus, Sub procedures and sub functions: Menus - Common Dialog Boxes - Writing General Procedures. Multiple Forms: Multiple Forms - Standard Code Modules-Variables and Constants in Multiple-Form Projects. UNIT – IV

List Boxes and Combo boxes - Do/Loop - For/Next Loop - Using MsgBox Function - Using String Function - Arrays: Control Arrays - Single Dimension Array - For Each/Next Statements - User defined data types - Multidimensional Arrays.

UNIT - V

Accessing Database Files: Visual basic and Database Files - Using Data Control - Viewing a Database File - Navigating the Database in Code - Using List Boxes and Combo boxes as Data-Bound Controls.

TEXT BOOK:

1. Julia Case Bradley and Anita C.Millspaugh, "Programming in Visual Basic 6.0", Tata McGraw-Hill Edition, 2011.
2. REFERENCE BOOKS: 1. Gary Cornell, "Visual Basic 6 from the Ground up", McGraw-Hill Education,1998 2. Mohammed Azam, "Programming with Visual Basic 6.0", 1st Edition, Vikas Publishing House Pvt. Ltd., Chennai, 2001.

DEPARTMENT OF COMPUTER SCIENCE

GUI PROGRAMMING

UNIT – I

Introduction to Visual Basic: Programming Languages, Procedural, Object Oriented and Event Driven, Writing VB Projects, VB Environment, Writing First Project-Finding and Fixing Errors - VB Help. Controls in VB - Coding for Controls.

Introduction to Visual Basic 6 & Programming Languages

The concept of computer programming

- ✚ Programming means designing a set of instructions to ask the computer to carry out certain jobs that are very much faster and more efficient than human beings can do.
- ✚ As the microchips of a CPU can only understand 0 and 1 in the binary system, the earliest programming language uses combinations of 0 and 1 code to communicate with computer, a language that is called **machine language**.
- ✚ Machine language is extremely difficult to learn. Fortunately, scientists have invented human language-like program languages or high level programming languages which are much easier to master.
- ✚ Some of the high level programming languages are Fortran, Cobol, Java, C, C++, c#, Visual Basic, Turbo Pascal, flash action script, JavaScript, HTML and more.

What is Visual Basic?

- VISUAL BASIC is a high level programming language which evolved from the earlier DOS version called **BASIC**.
- **BASIC** means **B**eginners' **A**ll-purpose **S**ymbolic **I**nstruction **C**ode.
- It is a relatively easy programming language to learn.
- The code looks a lot like English Language. Different software companies produced different versions of BASIC, such as Microsoft QBASIC, QUICKBASIC, GWBASIC, IBM BASICA and so on.
- However, people prefer to use Microsoft Visual Basic today, as it is a well-developed programming language and supporting resources are available everywhere.
- Now, there are many versions of VB exist in the market, the most popular one and still widely used by many VB programmers is none other than Visual Basic 6 .

Object Oriented Programming Languages:

- We also have VB.net, Visual Basic 2005, Visual Basic 2008 , Visual Basic 2010, Visual Basic 2012 and Visual Basic 2013 .
- VB2008, VB2010, VB2012 and VB2013 are fully object oriented programming (OOP) languages.
- VISUAL BASIC is also a VISUAL and **Event-driven Programming Language**. These are the main divergence from the old BASIC.
- In BASIC, programming is done in a text-only environment and the program is executed sequentially.
- In VB6, programming is done in a graphical environment. In the old BASIC, you have to write program code for each graphical object you wish to display it on screen, including its position and its color.
- However, In VB6, you just need to drag and drop any graphical object anywhere on the form, and you can change its properties using the properties window.
- ❖ In addition, Visual Basic 6 is **Event-driven** because we need to write code in order to perform some tasks in response to certain events.
- ❖ The events usually comprises but not limited to the user's inputs. Some of the events are load, click, double click, drag and drop, pressing the keys and more.
- ❖ We will learn more about events in later lessons. Therefore, a VB6 Program is made up of many subprograms, each has its own program code, and each can be executed independently and at the same time each can be linked together in one way or another.

The Visual Basic 6 Integrated Development Environment:

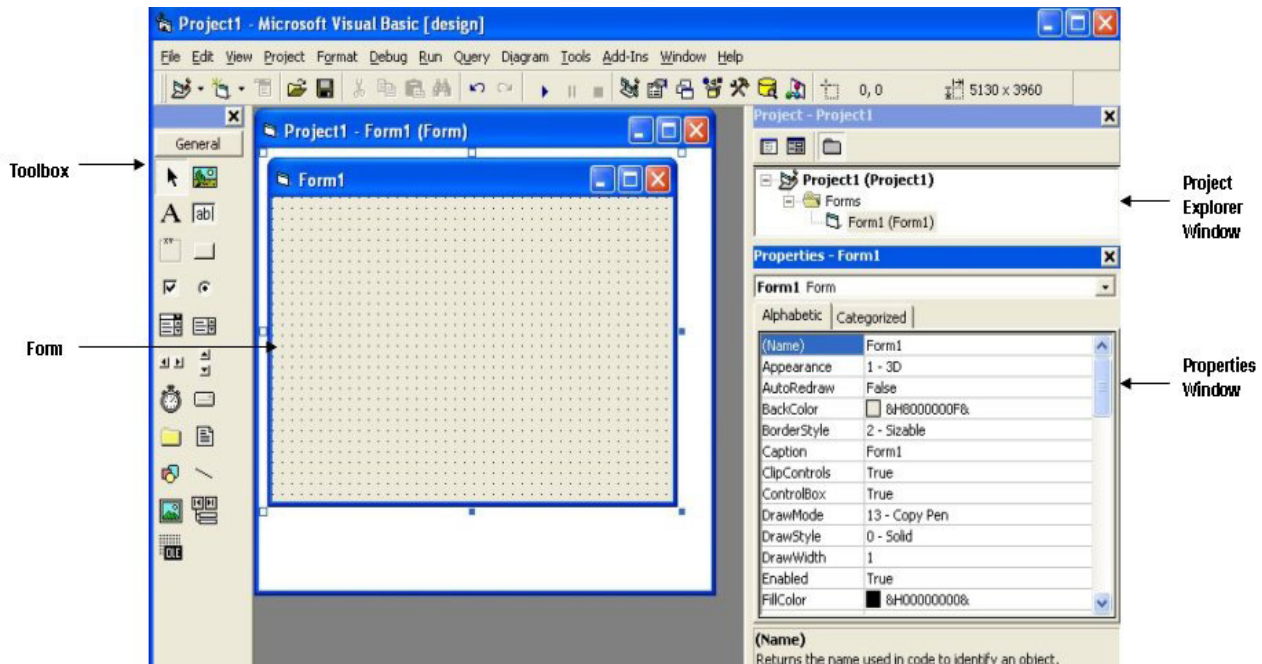
- Before you can program in VB 6, you need to install Visual Basic 6 compiler in your computer.
- You can purchase a copy of Microsoft or Microsoft Visual Basic Professional 6.0 with Plus Pack from Amazon.com, both are vb6 compilers.
- If you have already installed Microsoft Office in your PC or laptop, you can also use the built-in Visual Basic Application in Excel to start creating Visual Basic programs without having to spend extra cash to buy the VB6 compiler.
- After installing vb6 compiler, the icon with appears on your desktop or in your programs menu. Click on the icon to launch the VB6 compiler. On startup, Visual Basic 6.0

WRITING A NEW PROJECT:



- You can choose to start a new project, open an existing project or select a list of recently opened programs.
- A project is a collection of files that make up your application.
- There are various types of applications that we could create, however, we shall concentrate on creating Standard EXE programs (EXE means executable).
- Before you begin, you must think of an application that might be useful, have commercial values. educational or recreational. click on the Standard EXE icon to go into the actual Visual Basic 6 programming environment.
- When you start a new Visual Basic 6 Standard EXE project, you will be presented with the Visual Basic 6 Integrated Development Environment (IDE).
- The Visual Basic 6 Integrated Programming Environment is show in Figure 1.2. It consists of the toolbox, the form, the project explorer and the properties window.

VB6 Programming Environment



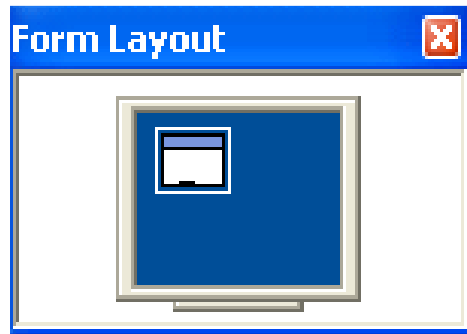
Form is the primary building block of a Visual Basic 6 application. A Visual Basic 6 application can actually comprises many forms; but we shall focus on developing an application with one form first.

We will learn how to develop applications with multiple forms later. Before you proceed to build the application, it is a good practice to save the project first. You can save the project by selecting **Save Project** from the File menu, assign a name to your project and save it in a certain folder.

Altering a form:

In many Visual Basic applications, the size and location of the form at the time you finish the design are the size and shape that the user initially sees at run time. Although how you set the form's location and size is important this doesn't mean that this **is** fixed forever. Visual Basic changes the size and location of forms while the program is running or even when it starts up.

We can resize a form is common to all Microsoft Windows applications: move the mouse to one of the hot spots of the form. In a form, the hot spots are the sides or corners of the form. The mouse pointer changes to a double-headed arrow when you're at a hot spot. At this point, we can hold the mouse button down and drag the form to change its size or shape. To change where the form will appear at run time, you need to work with the Form Layout window, which is in the lower-right corner of your screen. This window looks like a blank monitor, as shown here.



To change the position of a form at run time, follow these steps:

1. Move the cursor to the Form Layout **window**.
2. Drag the form to the position in which you want it to appear when the user starts your program.

The Control Properties

Before writing an event procedure for the control to response to an event, you have to set certain properties for the control to determine its appearance and how will it work with the event procedure. You can set the properties of the controls in the properties window or at runtime.

Figure 3.1 on the right is a typical properties window for a form. You can rename the form caption to any name that you like best. In the properties window, the item appears at the top part is the object currently selected (in Figure 3.1, the object selected is Form1). At the bottom part, the items listed in the left column represent the names of various properties associated with the selected object while the items listed in the right column represent the states of the properties. Properties can be set by highlighting the items in the right column then change them by typing or selecting the options available.

For example, in order to change the caption, just highlight Form1 under the name Caption and change it to other names. You may also try to alter the appearance of the form by setting it to 3D or flat. Other things you can do are to change its foreground and background color, change the font type and font size, enable or disable minimize and maximize buttons and etc.

You can also change the properties at runtime to give special effects such as change of color, shape, animation effect and so on. For example the following code will change the form color to red every time the form is loaded. VB uses hexadecimal system to represent the color. You can check the color codes in the properties windows which are showed up under ForeColor and BackColor .

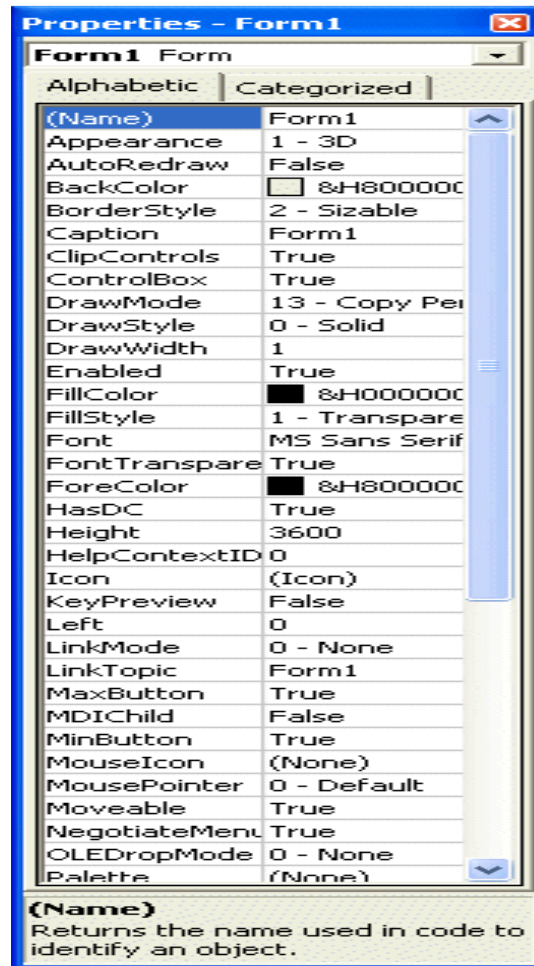


Figure 3.1

Example: Program to change background color

```
Private Sub
Form Load ()
Form1.Show
Form1.BackColor = &H000000FF&
End Sub
```

Example: Program to change shape

This example is to change the control Shape to a particular shape at runtime by writing the following code. This code will change the shape to a circle at runtime.

```
Private Sub Form Load ()
Shape1.Shape = 3
End Sub
```

Please note that knowing how and when to set the objects' properties is very important as it can help you to write a good program or you may fail to write a good program. So, I advise you to spend a lot of time playing with the objects' properties.

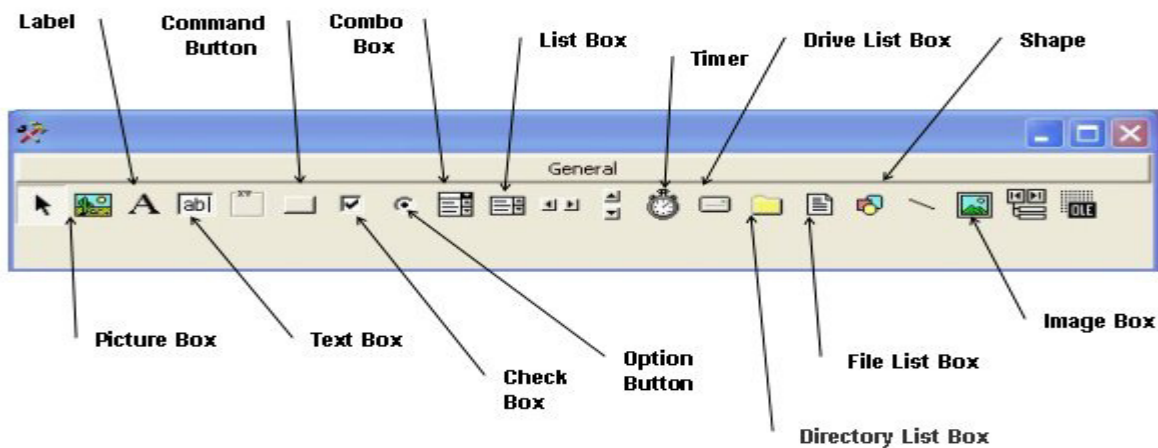
We are not going into the details on how to set the properties. However, I would like to stress a few important points about setting up the properties.

- You should set the Caption Property of a control clearly so that a user knows what to do with that command.
- Use a meaningful name for the Name Property because it is easier to write and read the event procedure and easier to debug or modify the programs later.
- One more important property is whether to make the control enabled or not.

Finally, you must also considering making the control visible or invisible at runtime, or when should it become visible or invisible.

3.2 Handling some of the common controls

Figure 3.2 below is the VB6 toolbox that shows the basic controls.



The Text Box

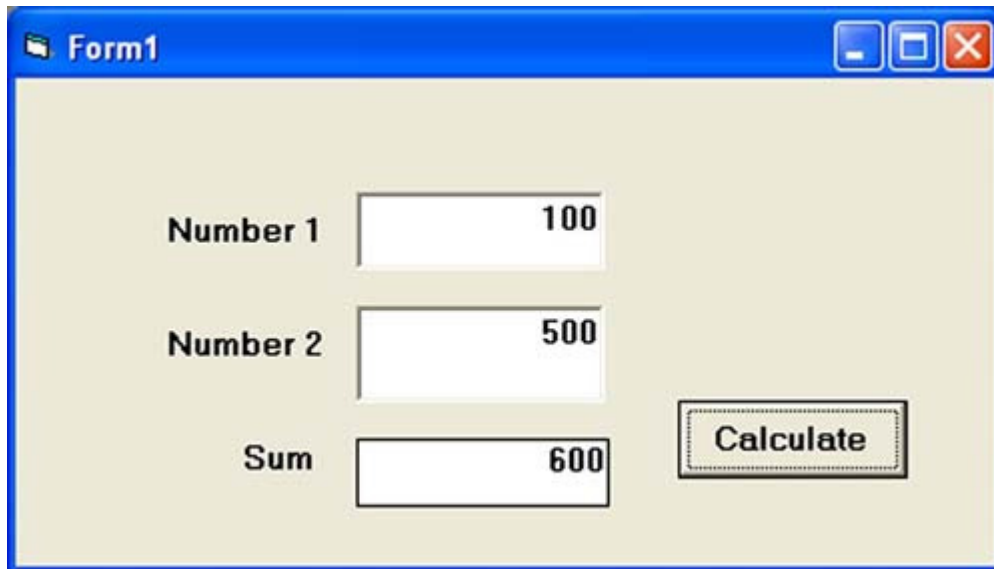
The text box is the standard control for accepting input from the user as well as to display the output. It can handle string (text) and numeric data but not images or pictures. Just like text fields in websites, powered not by Windows, but typically [linux web hosting platforms like iPage](#), these fields collect user input. String in a text box can be converted to a numeric data by using the function Val(text). The following example illustrates a simple program that processes the input from the user.

Example:

In this program, two text boxes are inserted into the form together with a few labels. The two text boxes are used to accept inputs from the user and one of the labels will be used to display the sum of two numbers that are entered into the two text boxes. Besides, a command button is also programmed to calculate the sum of the two numbers using the plus operator. The program use creates a variable sum to accept the summation of values from text box 1 and text box 2. The procedure to calculate and to display the output on the label is shown below. The output is shown in Figure 3.3

```
Private Sub Command1_Click()
'To add the values in text box 1 and text box 2
Sum = Val(Text1.Text) + Val(Text2.Text)
'To display the answer on label 1
Label1.Caption = Sum
End Sub
```


Figure 3.3

A screenshot of a Windows application window titled "Form1". The window has a blue title bar with standard minimize, maximize, and close buttons. The main area has a light beige background. It contains three text boxes and one button. The first text box is labeled "Number 1" and contains the value "100". The second text box is labeled "Number 2" and contains the value "500". The third text box is labeled "Sum" and contains the value "600". To the right of the "Sum" text box is a button with a dashed border and the text "Calculate".

The Label

The label is a very useful control for Visual Basic, as it is not only used to provide instructions and guides to the users, it can also be used to display outputs. One of its most important properties is **Caption**. Using the syntax **label.Caption**, it can display text and numeric data. You can change its caption in the properties window and also at runtime. Please refer to Example 3.1 and Figure 3.1 for the usage of label.

The Command Button

The command button is one of the most important controls as it is used to execute commands. It displays an illusion that the button is pressed when the user click on it. The most common event associated with the command button is the Click event, and the syntax for the procedure is

```
Private Sub Command1_Click ()  
Statements  
End Sub
```

The Picture Box

The Picture Box is one of the controls that is used to handle graphics. You can load a picture at design phase by clicking on the picture item in the properties window and select the picture from the selected folder. You can also load the picture at runtime using the **LoadPicture** method. For example, the statement will load the picture grape.gif into the picture box.

```
Picture1.Picture=LoadPicture ("C:\VB program\Images\grape.gif")
```

You will learn more about the picture box in future lessons. The image in the picture box is not resizable.

The Image Box

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 Load Picture 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")
```

The List Box

The function of the List Box is to present a list of items where the user can click and select the items from the list. In order to add items to the list, we can use the **AddItem method**. For example, if you wish to add a number of items to list box 1, you can key in the following statements

Example:

```
Private Sub Form_Load ( )  
List1.AddItem "Lesson1"  
List1.AddItem "Lesson2"  
List1.AddItem "Lesson3"  
List1.AddItem "Lesson4"  
End Sub
```

The items in the list box can be identified by the **ListIndex** property, the value of the ListIndex for the first item is 0, the second item has a ListIndex 1, and the third item has a ListIndex 2 and so on

The Combo Box

The function of the Combo Box is also to present a list of items where the user can click and select the items from the list. However, the user needs to click on the small arrowhead on the right of the combo box to see the items which are presented in a drop-down list. In order to add items to the list, you can also use the **AddItem method**. For example, if you wish to add a number of items to Combo box 1, you can key in the following statements

Example

```
Private Sub Form_Load ( )  
Combo1.AddItem "Item1"  
Combo1.AddItem "Item2"  
Combo1.AddItem "Item3"  
Combo1.AddItem "Item4"  
End Sub
```

The Check Box

The Check Box control lets the user selects or unselects 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...Elsif 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
```

The Option Box

The Option Box control also lets the user selects one of the choices. However, two or more Option Boxes must work together because as one of the Option Boxes is selected, the other Option Boxes will be unselected. In fact, only one Option Box can be selected at one time. When an option box is selected, its value is set to “True” and when it is unselected; its value is set to “False”. In the following example, the shape control is placed in the form together with six Option Boxes. When the user clicks on different option boxes, different shapes will appear. The values of the shape control are 0, 1, and 2,3,4,5 which will make it appear as a rectangle, a square, an oval shape, a rounded rectangle and a rounded square respectively.

Example:

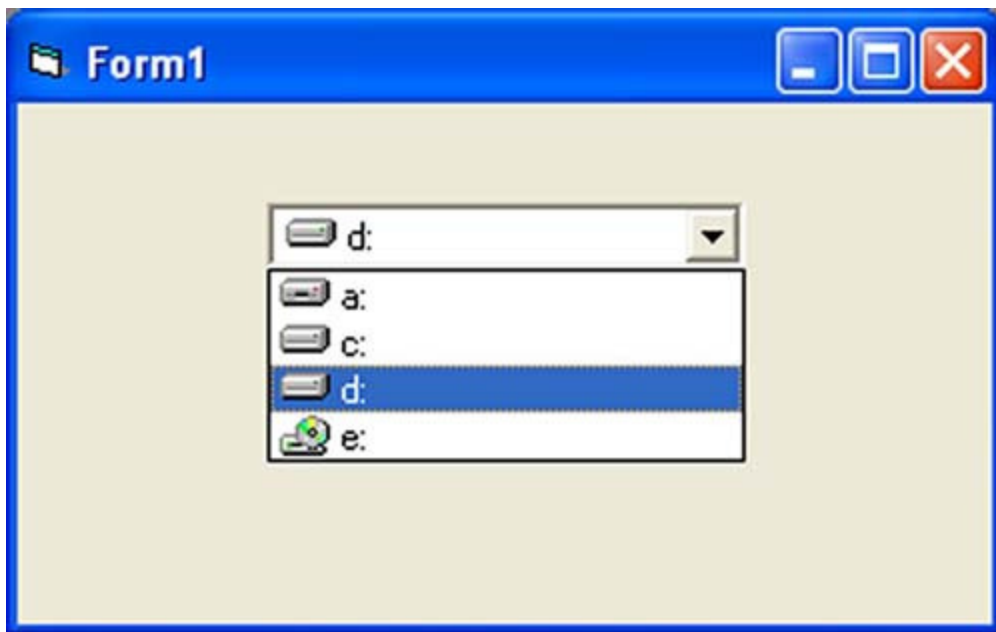
```
Private Sub Option1_Click ()  
Shape1.Shape = 0  
End Sub  
Private Sub Option2_Click()  
Shape1.Shape = 1  
End Sub  
Private Sub Option3_Click()  
Shape1.Shape = 2  
End Sub  
Private Sub Option4_Click()  
Shape1.Shape = 3  
End Sub  
Private Sub Option5_Click()  
Shape1.Shape = 4  
End Sub  
Private Sub Option6_Click()  
Shape1.Shape = 5
```

End Sub

The Drive List Box

The Drive ListBox is for displaying a list of drives available in your computer. When you place this control into the form and run the program, you will be able to select different drives from your computer as shown in Figure 3.4

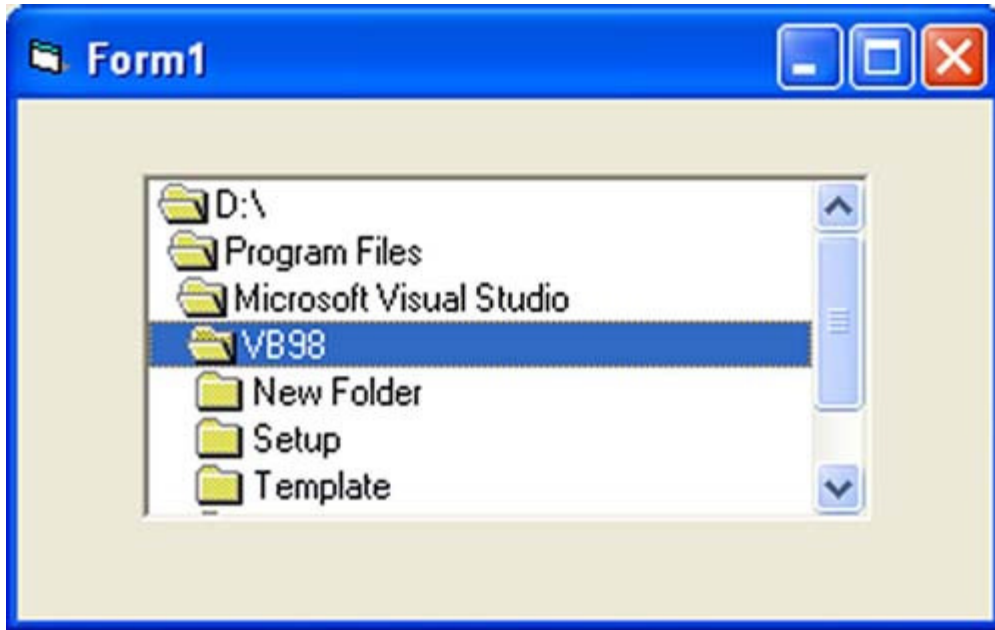
Figure 3.4 The Drive List Box



The Directory List Box

The Directory List Box is for displaying the list of directories or folders in a selected drive. When you place this control into the form and run the program, you will be able to select different directories from a selected drive in your computer as shown in Figure 3.5

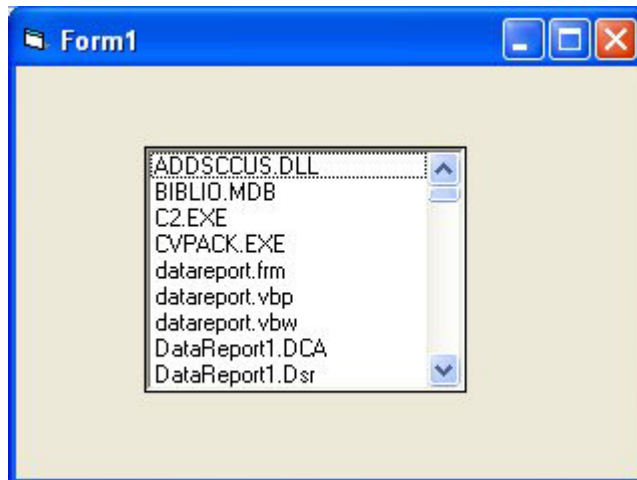
Figure 3.5 The Directory List Box



The File List Box

The File List Box is for displaying the list of files in a selected directory or folder. When you place this control into the form and run the program, you will be able to show the list of files in a selected directory as shown in Figure 3.6

Figure 3.6 The File List Box



You can coordinate the Drive List Box, the Directory List Box and the File List Box to search for the files you want. The procedure will be discussed in later lessons.

In this lesson, we shall learn some basic rules about writing the VB program code. Each control or object in VB can run many kinds of events; these events are listed in the dropdown list in the code

window that is displayed when you double-click on an object and click on the procedures' box (refer to Figure 2.3). Among the events are loading a form, clicking of a command button, pressing a key on the keyboard or dragging an object and more. For each event, you need to write an event procedure so that it can perform an action or a series of actions.

To start writing an event procedure, you need to double-click an object. For example, if you want to write an event procedure when a user clicks a command button, you double-click on the command button and an event procedure will appear as shown in Figure 2.1. It takes the following format:

```
Private Sub Command1_Click  
(Key in your program code here)>  
End Sub
```

You then need to key-in the procedure in the space between Private Sub Command1_Click..... End Sub. Sub actually stands for sub procedure that made up a part of all the procedures in a program. The program code is made up of a number of statements that set certain properties or trigger some actions. The syntax of Visual Basic's program code is almost like the normal English language though not exactly the same, so it is very easy to learn.

The syntax to set the property of an object or to pass certain value to it is:

Object.Property

Where Object and Property is separated by a period (or dot).

For example, the statement **Form1.Show** means to show the form with the name Form1, **Label1.Visible=true** means label1 is set to be visible, **Text1.text="VB"** is to assign the text VB to the text box with the name Text1, **Text2.text=100** is to pass a value of 100 to the text box with the name text2, **Timer1.Enabled=False** is to disable the timer with the name Timer1 and so on.

Example:

```
Private Sub Command1_click  
Label1.Visible=false  
Label2.Visible=True  
Text1.Text="You are correct!"  
End sub
```

Example 4.2

```
Private Sub Command1_click  
Label1.Caption=" Welcome"  
Image1.visible=true
```

```

End sub
Example 4.3
Private Sub Command1_click
Picture1.Show=true
Timer1.Enabled=True
Label1.Caption="Start Counting
End sub

```

In Example 4.1, clicking on the command button will make label1 become invisible and label2 become visible; and the text " You are correct" will appear in TextBox1. In example 4.2, clicking on the command button will make the caption label1 change to "Welcome" and Image1 will become visible. In example 4.3 , clicking on the command button will make Picture1 show up, timer starts running and the caption of label1 change to "Start Counting".This type of operation could be particularly useful in applications such as a website stat counter (most [web hosting plans](#) include some analytics or stat package).

Syntaxes that do not involve setting of properties are also English-like, some of the commands are **Print**, **If...Then....Else....End If**, **For...Next**, **Select Case.....End Select** , **End** and **Exit Sub**. For example, **Print " Visual Basic"** is to display the text Visual Basic on screen and **End** is to end the program. Other commands will be explained in details in the coming lessons.

Program code that involves calculations is fairly easy to write,just like what you do in mathematics. However, in order to write an event procedure that involves calculations, you need to know the basic arithmetic operators in VB as they are not exactly the same as the normal operators , except for + and -

For multiplication, we use *, for division we use /, for raising a number x to the power of n, we use x^n and for square root, we use **Sqr(x)**. VB offers many more advanced mathematical functions such as **Sin**,**Cos**, **Tan** and **Log**, they will be discussed in lesson 10. There are also two important functions that are related to arithmetic operations,

i.e. the functions **Val** and **Str\$** where Val is to convert text to numerical value and Str\$ is to convert numerical to a string (text). While the function Str\$ is as important as VB can display a numeric values as string implicitly, failure to use Val will results in wrong calculation. Let's examine example 4.4 and example 4.5.

Example 4.4

```

Private Sub Form_Activate()
Text3.text=text1.text+text2.text
End Sub

```

Example 4.5

```
Private Sub Form_Activate()  
    Text3.text=val(text1.text)+val(text2.text)  
End Sub
```

When you run the program in example 4.4 and enter 12 in textbox1 and 3 in textbox2 will give you a result of 123, which is wrong. It is because VB treat the numbers as string and so it just joins up the two strings. On the other hand, running example 4.5 will give you the correct result, i.e., 15.

The methods of working with the properties window:

1. Display the Properties window by pressing F4 if it isn't visible.
2. *Move* to the Properties window and select an item from the properties in the list box.
3. Enter the new setting for the property.
4. Press *ENTER* to accept the new setting.

COMMON FORM PROPERTIES:

Caption:

The Caption property sets the title of the form. The caption is also the title that Microsoft Windows uses for the application icon when the user minimizes the application.

Name :

This property is used to give the name for refer to the Form. The default value is, "Form1" and so its value starts out the same as the default value of the Caption property. This property is mostly used in larger projects.

Appearance :

Determines whether the form will have a three-dimensional look. If you leave it at the default value of 1, the form will look three-dimensional. Change it to 0, and the form will appear flat.

BorderStyle:

- Drop down the list and you can see the value and a description. We can choose among five values for this property. The default value, 2-Sizable, allows the user to size and shape the form via the hot spots located on the boundary of the form.
- Change this setting to 1-Fixed Single, and the user will no longer be able to resize the window. All the user will be able to do is minimize or maximize the window.

- If the BorderStyle value is 0-None, and the application will show no border whatsoever, and therefore no minimize, maximize, or control box buttons. Because of this, a form created without a border cannot be moved, resized, or reshaped.
- The third setting, 3-Fixed Double, is not often used for ordinary forms, but it is commonly used for dialog boxes. It gives a nonsizable border that is twice as thick as normal.
- The fourth setting, 4-Fixed ToolWindow, is not used very often. Under Windows 95/98 and NT 4, this displays the form with a Close button.
- The fifth setting, 5-Sizable ToolWindow, works much like the Fixed ToolWindow setting. This will display the form with a Close button, and the text from the title bar will appear in a reduced size. The form does not appear in the Windows 95/98 and NT 4 task bar, but the user can change the size of the form.

ControlBox :

Changes to this property go into effect only when a user runs the application. As in any Microsoft Windows application, control boxes are located in the far left corner of the title bar. Clicking the box displays a list of common window tasks, such as window minimizing, maximizing, and closing, along with keyboard equivalents when they exist.

Enabled :

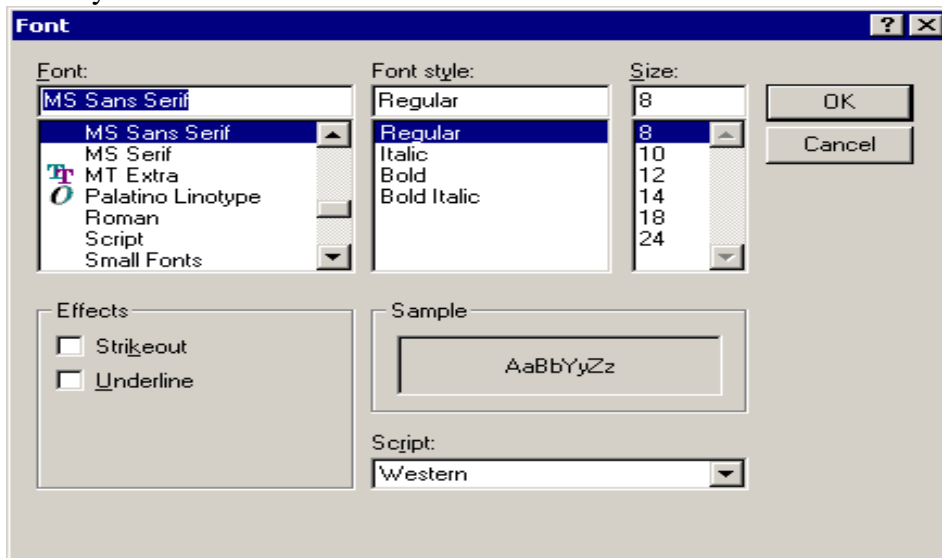
If we set Enabled to False, the form cannot respond to any events such as the user clicking on the form.

Height, Width:

Height and Width are interesting properties, and not only because they can be set two ways. They are examples of properties that default to using the Microsoft Windows twips scale to measure the sizes of the objects involved.

Font :

This is the first example you have seen of a property that is set from an ordinary Windows dialog box. If you move the focus to this item and then double-click on the three dots.



Icon :

The Icon property is one you will use frequently. This property determines the icon your application will display when it is minimized on the toolbar or turned into a stand-alone application on the Windows desktop.

Left, Top :

These properties determine the distance between the left or top of the form and the screen. Set the value of the Top property to 0, and the form you're designing is flush with the top. Set the value of the Left property to 0, and it will be flush with the left side of the screen.

MousePointer, MouseIcon :

MousePointer is a useful property that lets you set the shape of the mouse pointer. The default value is 0, but as the pull-down list indicates, there are 17 other values. A setting of 4-Icon, for example, turns the mouse pointer into a rather pretty square within a square. The settings you will use most often are 0 (the default arrow shape) and a value of 11. Using a value of 11-Hourglass changes the mouse pointer to the usual hourglass, and as in other Microsoft Windows applications, this setting is useful for indicating to the user has to wait until the computer finishes what it is doing.

We can change it at design time by following these steps:

1. Set the MousePointer property to 99.
2. **Pick** the icon you want to be the custom mouse icon as the value of the MouseIcon property. Note that you cannot set the MouseIcon property until you set the MousePointer property to be 99.

StartupPosition :

This neat property gives another way to decide on the initial position of your form at run time. It is generally more precise than using the Form Layout window. We have four choices; the most common (besides the default value) is a setting of 2, which lets you center the form on the screen.

Visible :

This is another property that is dangerous to change by mistake. Set **the** value of this property to False, and the form will no longer be visible. We usually will want **to** make a form invisible only when you are designing an application with multiple forms. Then we can hide one or more of the forms by using the Visible property.

Window/State :

This property determines how the form will look at run time. There are three possible settings. A setting of 1 reduces the form to an icon, and a setting **of** 2 maximizes the form. A setting of 0 is the normal default setting. This property is **most** often changed in code.

SCALE PROPERTIES :

This properties are used to position the objects or text in form accurately. Visual Basic provides five properties that affect the scale used in a form.

ScaleMode ScaleMode:

It allows you to change the units used in the form's internal coordinate system. There are seven other possibilities. You can create your own units (the value of this setting is 0), keep the default twips (this value is 1), or use one of the six remaining choices. An interesting setting—especially for graphics—is 3. This uses one pixel (a picture element—the smallest unit of resolution on your monitor) as the scale. And of course, if you are more comfortable with them, you can choose inches (5), millimeters (6), or centimeters (7).

ScaleHeight, ScaleWidth:

Use the ScaleHeight and ScaleWidth properties when you set up your own scale for the height and width of the form. Resetting these properties has the side effect of setting the value of the ScaleMode property back to 0.

Scaleleft, ScaleTop :

These properties describe what value Visual Basic uses for the left or top corner of the form. The original value for each of these properties is 0. Like ScaleHeight and ScaleWidth, these properties are most useful when you are working with graphics. For example, if you are writing a program .that works with a graph, you rarely want the top left corner to be at point 0,0.

COLOR PROPERTIES :

We can specify the background color (BackColor) and the foreground color (ForeColor) for text and graphics in the form.

The BackColor and ForeColor Properties via the Color Palette :

Suppose we try to set the BackColor property. If we open the Properties window and select BackColor, then the default setting is,

&H8000000F&

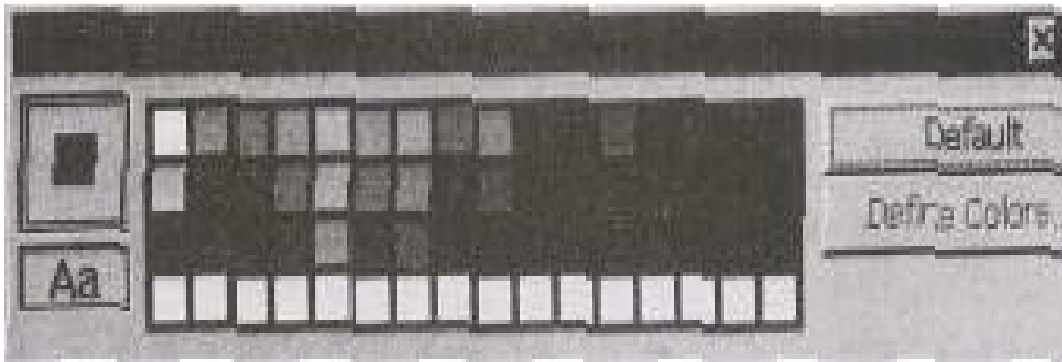
which is rather cryptic, to say the least. In fact, Visual Basic describes color codes by using a hexadecimal code (base 16), which is described in the section "Bits, Bytes, and Hexadecimal (Base 16) Numbers in Visual Basic".

The most common way for you to set colors is **to choose** one of the color properties and click the down arrow in the Settings box. This opens up a tabbed dialog box with two tabs. The System tab on this dialog box gives you a list **of the** colors currently used by Windows for its various elements. If you click on the Palette tab, the color grid shown here pops up.

Click one color, and the color code for that color is placed in the Settings box. The background color of the form will automatically show your **changes** to the Background property. You won't see the effect of changing the ForeColor property until you do something like displaying text on the form.

Working with the Color Palette :

We can also create your own colors by working with the color palette directly. **Open** the color palette by going to the View menu and choosing the color palette. The left of the palette, a dark box enclosed in a lighter box. The inner box displays the current foreground color, and the outer box displays the current background color.

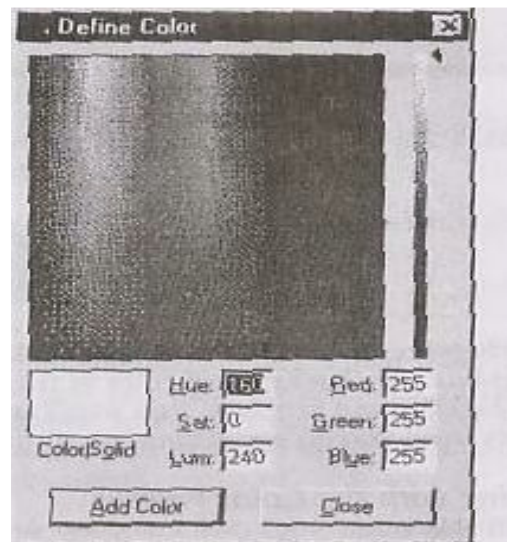


We can change the foreground color by clicking the inner box and then clicking any **of** the colored boxes displayed. To change the background color, click the outer **box** and then click any of **the** colored boxes displayed.

The text box in the lower-left corner of the color palette displays the foreground and background colors for any text in the form. To go back to the default colors specified in the windows control panel, click the default command button at the right.

We can create own colors for the color palette. Each of the blank boxes on the bottom of the color palette represents a possible custom color. This contains following steps.

- Click one of these blank boxes, and then click the Define Colors command button. This opens the Define Color dialog box.
- Change the amount of red, green, or blue color, hue, saturation, or luminosity of the color to suit your needs by adjusting the controls in the dialog box.
- Press the Add Color button to create the custom color or the close button to cancel.

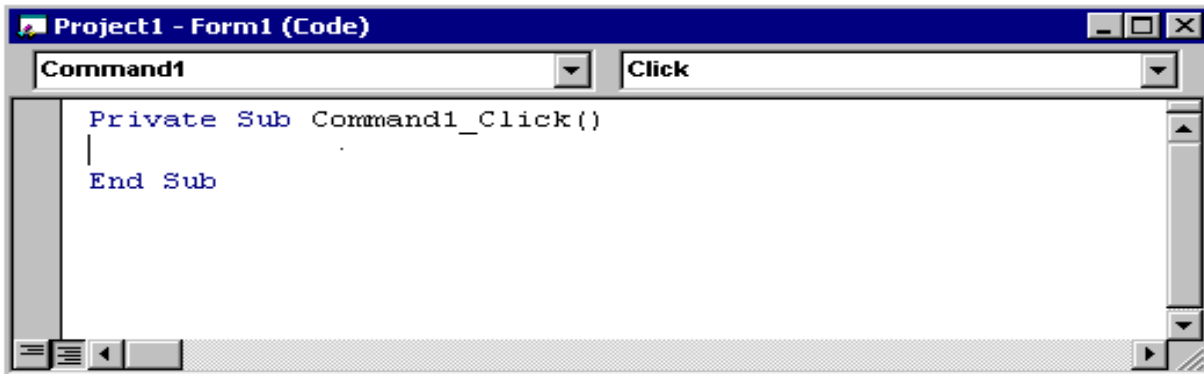


MAKING A FORM RESPONSIVE :

We can make a form in the right size, shape, and having the right background color is hardly what windows programming is all about. The essence of a Microsoft windows program is to make your forms respond to user actions.

The code window and writing a simple event procedure :

We can make form respond to a mouse click. Double click in any blank part of the Form1. The following screen will appear.



The two drop-down list boxes in the top part of the screen. If you click the arrow in the right hand box, a list of all the events a form can recognize, that is Load, LostFocus, MouseMove, MouseDown, MouseUp, paint, etc.

If pull down the left-hand box in the Code window, we can see a list of the controls on your form. Since you have yet to put any text boxes, command buttons or other controls on this form, no objects except the form itself are listed in this box.

```
Private Sub Form_Load ()  
  
End Sub
```

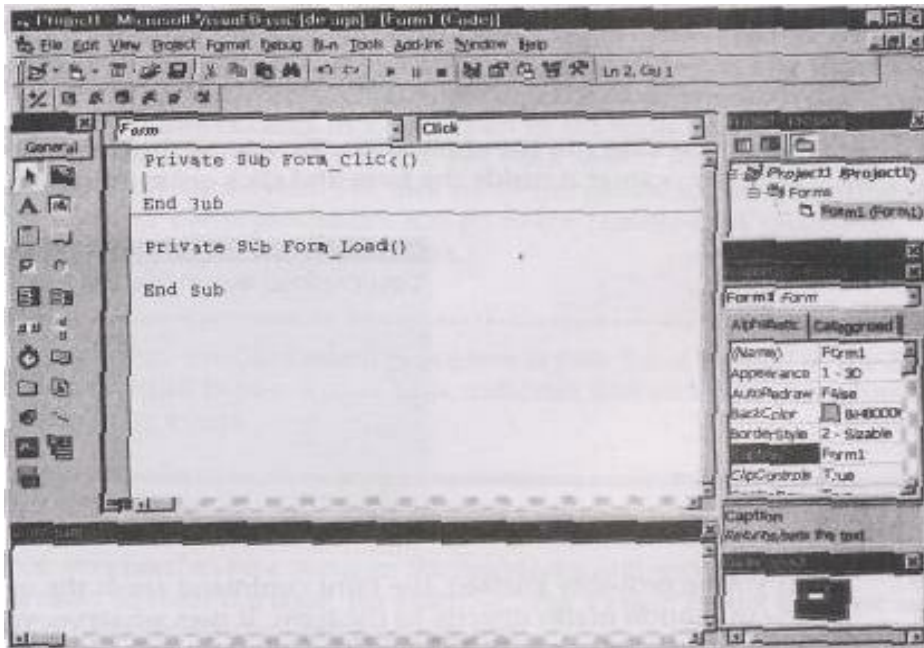
This code is an example of an *event procedure template*. Like any template, it gives framework in which you'll work. The Form-Load event is triggered when Visual Basic loads a form in memory. Also we can write code in this event in order to set the initial properties of forms via code. For example, if you make the Form_Load event procedure read:

```
Private Sub Form_Load ()  
  
    BackColor = vbRed  
  
End Sub
```

and run the program you'll see that the background color of the form indeed changes to red.

Working with the Form-Click Event :

First start up a new Project (Standard EXE type) to show off this event **procedure**. What we want to do is trigger some code being processed in response to the user clicking on the form. To do this, you have to bring up the template for the Form_Click event **procedure**. For this, make sure you are in the code window and then:



1. Move to the event **drop-down list box on the right** and click the down arrow.

2. **Move** through the box until you get to the Click item. The Items in the Event list box are listed in "shorthand form"—that's why we see Click rather than Form_Click. You have to look at the left drop-down list box to see what object are working with in order **to** conclude that this is really **the** Fonn_Click event.

3. **Click** on it.

Then Visual Basic does the following:

- Gives a new event procedure template for the Form_Click event procedure
- Adds a dotted line between the Form_Load event and the Click event
- Moves the cursor to the blank line before the End Sub line in the Click event procedure template .

Also we can write the code necessary for Visual Basic to respond to a mouse click with a message. If the cursor is not at the blank line before the End Sub in the Form_CLICK template, move it there by scrolling through the Code window and clicking on the blank line. Press the TAB key once or the SPACEBAR a few times and type **Print "Welcome to BCA"** Coding should be,

```

Private Sub Form_Click ( )
    Print "Welcome to BCA "
End sub

```

Now press F5 to run the application. As soon as the form pops up, move the mouse until the pointer is inside the form and click once. We get the message “Welcome to III – CS” as output.

More General Event Procedures :

In general, no matter what event you want the form to respond to, the code for an event procedure for a form in Visual Basic begins with something that looks like this:

```
Private Sub Form_NameOfTheEvent ( )
```

Event Procedure	Tells the Form
Private Sub Form_Click()	→ to respond to a click
Private Sub Form_DblClick()	→ to respond to a double-click
Private Sub Form_Resize	→ to respond when the user resizes the form.

The following table gives you some of the most common examples of event procedures that are user driven and when you will want to use them.

PRINTING A VISUAL REPRESENTATION OF A FORM:

Visual Basic relies on the underlying Windows program to handle its printing needs. For this reason, we should make sure we have configured Windows with the name of our printer. Most of the time we won't need to get involved with the Windows Print Manager; Visual Basic takes care of the interface pretty well. It uses whatever printer information is contained in the Microsoft Windows environment control panel.

However, getting an image of the form, including whatever is currently displayed on the form, to the printer requires only a single command-. PrintForm. Notice that since this also affects what the form does, as opposed to what it is, it is another example of a Visual Basic method. The PrintForm method tries to send to your printer a dot-for-dot image of the entire form. As an example, add the line PrintForm to the Double-click procedure before the End Sub line.

TYPOS :

Visual Basic can point out many typing errors when we enter a line of code, and it will even correct some (such as leaving off a closing quote). But someone enter code without typos. So to see what might happen, let's suppose we made a typo when we were writing the Click event procedure presented earlier, and you misspelled the command word Print by typing Printf instead.

Now the word Printf should be in a different color (probably red or black instead of blue). If try to run the program and click in the form, Visual Basic will immediately respond with an error message box, and your screen will be,

he offending word is highlighted, and the message box tells you, "Sub or Function not defined"—what you entered isn't recognizable to Visual Basic.

If you press ENTER or click the OK command button, the offending word remains highlighted, and you can either type the correct replacement or move the mouse pointer to the "f" and press DEL. After you make the correction, the program will **run** as before.

Another common typo with the Print method is to forget it completely, we just type the text and run the program. To see what happens when you do this, delete the keyword Print and run the program again. This time Visual Basic responds with a box saying it cannot the line of code. Visual Basic can even find some syntax errors after we finish typing a line.

CREATING STAND-ALONE WINDOWS PROGRAMS :

One of the most exciting features of Visual Basic is the ability to change the projects into stand-alone Microsoft Windows programs. These will be files that users can simply double-click on in Explorer in order to run them.

To make a stand-alone VB application, simply go to the File menu and choose the Make Project EXE File option. This opens a dialog box that is,

The default name for the .exe version of your file is the project name. For the stand-alone program, the Windows desktop uses the same icon that Visual Basic uses for the executable version of the project.

When you distribute a Visual Basic program, it is also necessary to supply one or more dynamic link libraries (*DLL*) to the user. Dynamic link libraries are the cornerstone of Windows programming; among other features, they allow many programs to use the same code simultaneously. The VB DLL file contains various support routines that a Visual Basic program needs to handle the screens, numbers, and other parts of the application.

THE TOOLBOX:



The tool box contains a set of tools for working with a form are found on the menus. This is usually located on the far left of the VB screen, but it need not visible at all times.

The pointer :

This is not a control but is used to manipulate controls after create them. Click the pointer when we want to select, resize, or move an existing control. It is automatically activated after place a control on a form.

Command Buttons :

When click a command button, it gives the illusion of being pressed. This optical illusion comes from the shading used by VB for command buttons, and it is inherited from the windows operating system.

Image Controls :

The image control is used to display pictures. Since image controls also recognize the Click event. We can use them as graphical replacements for command buttons.

Text Boxes :

This is used to display text or to accept user input. Most of the code you write for text boxes is to process the information users enter into them. It automatically display multiple lines of text.

Labels :

Use labels for information that users shouldn't be able to change. It identifies the objects and use them to display output.

CONTROLS IN VB :

We can get a control on a form by double clicking on its icon in the toolbox. This gives the control in its default size and shape in the middle of the form. We can use a combination of pointing, clicking, and dragging to manipulate the toolbox.

Working with a control on a form :

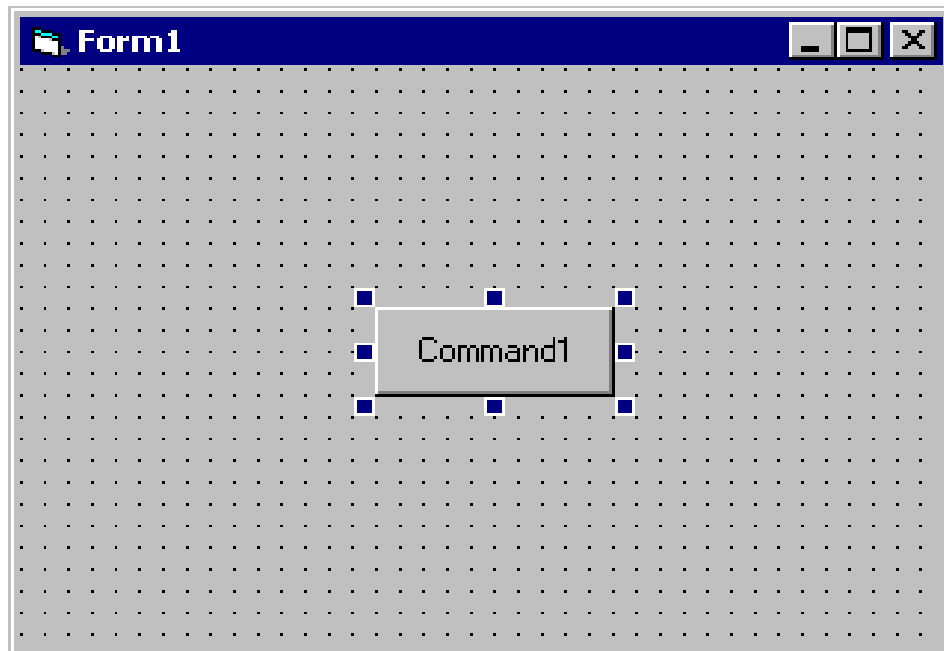
When we create a command button, it appears with a centered caption : Command1, Command2, and so on. We can cut and paste controls by cutting out of the control and using the cop item on the edit menu. When we paste a control that cut out of a form, it always appears in the top left-hand corner of the form.

Resizing an Existing Control :

To change the size of an existing command button at design time.

- Use the properties window to adjust the width and height properties (or)
- Work with the sizing handles

The following figure shows a command button with its eight sizing handles on an otherwise blank form. Simply click inside the control by moving the mouse pointer to the control and clicking once.



Try resizing a control with the sizing handles by following these steps:

- Move the mouse pointer to a sizing handle and click and hold down the left mouse button.
- Drag the mouse until the control is the size you want.

Moving an Existing Control :

To move an existing control with the mouse, the focus must be at the control. When we move the mouse so that the mouse pointer is inside a form or a control, its shape returns to the form of an arrow.

- Move the pointer anywhere inside the control, click the left mouse button, and hold it down.
- Drag the mouse until the control is at the location we want it to be, and then release the left button.

Using the Double-Click shortcut for creating Controls :

We can use the double-click method to quickly create them. If we double click on the any of the toolbox icons, the matching control appears in the center of the screen.

Suppose you want to create an application with five command buttons. The easiest way to do this to double click on the command button icon five times. This five command buttons in the center of the form. Then we can easily place the button.

Working with Multiple controls :

We can work with multiple controls as a single unit. The following steps for dragging method to select multiple controls.

- Imagine a rectangle that surrounds only those controls we want to select. Move to corner of this imagined rectangle and click the left mouse button.
- Hold the left mouse button down and drag the dotted rectangle until it covers all the controls you want to select. Then release the mouse button.

Once you have selected a group of controls, when you move any control in the group, VB moves the other controls in a similar way.

Locking controls :

We can lock the form by choosing FormatLock Controls or the Lock control tool from the toolbar, we can prevent from inadvertently moving a properly positioned control.

Deleting Controls:

We can delete a control from the form by,

- Move the mouse pointer until it is inside the control, and click the left the left mouse button to select it.
- Press DEL, or open the Edit menu and choose the Delete option by pressing ALT+E, D.

THE NAME (CONTROL NAME) PROPERTY :

This property determines the name VB uses for the event procedures we write to make the control respond to the user. The name property is for forms, the Name property for a control is even more important. While we can avoid using the name of a form.

If we want to change the height of a command button whose Name property is,

MyCommandButton.Height = 500

Picking meaningful names for controls goes a long way toward making the inevitable debugging of your application easier. For example, when we have five command buttons in an application, writing code that looks like this,

```
Private Sub Command4_Click ()  
  
End Sub
```

[
To make a form move left is a lot more confusing than writing it like this,

```
Private Sub LeftButton_Click ()  
  
End Sub
```

The limits on a control name are,

- The name must begin with a letter
- After that, we can use any combination of letters, digits, and underscores

- The name cannot be longer than 40 characters.

PROPERTIES OF COMMAND BUTTONS :

The properties window to customize the size and the shape of blank forms by customize controls.

The Caption property :

The caption property of a form determines the name that shows in the title bar. Similarly, the Caption property on a command button determines what the user sees on the face of the button. If we want to change the Caption property of the command button, then double click on the Command button icon to create the button in the center of the screen.

- Move to the properties window
- Go to the Caption property by using the mouse or the UP ARROW and DOWN ARROW keys.

Visible :

This property determines whether the Command button is visible or not. If this has value is true then the button is visible, otherwise invisible.

Enabled :

This is used to determine whether the button can respond to any event. If this has the value is false, then VB will not respond to any event. Changing this property also changes the appearance of an item.

Font :

This is used to change the font style on the command button. We can use one font at a time.

Height, Width :

This is used to define the height and width of the command button.

Left, Top :

This is used to determine the distance between the command button and the left edge and top of the container, respectively.

MousePointer :

Setting the MousePointer to something different than the usual arrow is good way to give a user feedback that they moved the focus to the Command button.

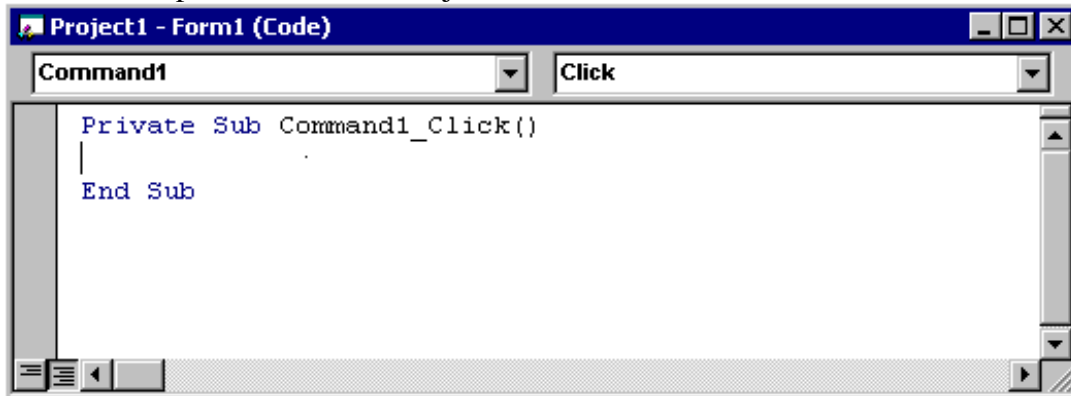
Shortcuts for setting properties :

We can set the Caption property for all the command buttons on your form. First set it for one command button and immediately select another command, the Caption property for the new controls is highlighted in the Properties window.

Also we can change the properties of group of controls. First select a group of controls, then Properties window will show only the common properties that the controls in the group share. Change one of them and all of them change.

SIMPLE EVENT PROCEDURES FOR COMMAND BUTTONS :

Writing an event procedure for a command button is similar to writing one for a form. Whenever we double click a control, VB opens the code window. It presents with a template for the most common event procedure for that object.



The general form of the event procedure template for controls is,

```
Private Sub ControlName_EventName ( )  
  
End Sub
```

Click the down arrow to the right of the command1 in the Properties window. It contains the list of objects in the Object list box. We can write event procedures for any of them by opening the Code window, moving through the Object list box, and selecting the object.

Command buttons can respond to 12 events, but clicking is by far the most common. Two others we may find useful are GotFocus and LostFocus. We can move the focus by his event procedure. An event procedure that look like,

```
Private Sub cmdButton_LostFocus  
  
End Sub
```

Also we can move the focus away from that button by using GotFocus. That is,

```
Private Sub cmdButton_GotFocus  
  
End Sub
```

For activating a command button is common to all windows applications. Move the focus by pressing TAB, and then press the SPACEBAR when the focus is where we want it to be.

ACCESS KEYS :

Visual Basic allows the access keys for quickly activate a control or a menu item. It easy to set up an access key for any object that has a Caption property. When set the Caption, to place an ampersand (&) in front of the letter we want to be the access key.

IMAGE CONTROLS :

The image control is used to display an image with a program such as Microsoft Paintbrush. Since image controls respond to the Click event, you can use these images to substitute for the command buttons. We can load a picture into an image control by resetting the value of the picture property. If you choose the picture property, this opens up a dialog box to load the image file.

TEXT BOXES :

The text boxes are the primary method for accepting input and displaying output in Visual Basic. It is also called edit field or edit control. There are 50 properties for text boxes. 39 properties are used at design time via the Properties Window. Some properties are,

Text :

This is used to controls the text the user sees. When we create a text box, the default value for this property is set to Text1, Text2, and so on.

Alignment :

This is used to control how the text is displayed. The default value is 0, which leaves the text left-aligned. Use a value of 1 and text is right aligned. Use a value of 2 and text is centered.

Multiline :

This property determines whether a text box can accept more than one line of text when the user runs the application, and it is usually combined with resetting the value of the ScrollBars property.

ScrollBars ;

This determines whether a text box has horizontal or vertical scroll bars. It has four possible settings. They are,

- 0 → this is the default value, the text box lack the both scroll bars.
- 1 → it has horizontal scroll bars only
- 2 → the text box has vertical scroll bars only
- 3 → it has both horizontal and vertical scroll bars.

BorderStyle :

This is used to determine whether the text box has borderstyle. The default value is 1, which gives a single-width border, and the value 0 gives the border disappears.

MaxLength :

This determines the maximum number of characters the text box will accept. The default value is 0, this means there is no maximum other than 32,000 characters.

PasswordChar :

If we want to get limited details from the name, then set password to that text box. Once set this property, all the user sees is a row of asterisks.

Locked :

This is used to prevent the users from changing the contents of the text box.

LABELS :

This is used to identify a text box or other control by describing its contents. Also used to display help information. The labels have 34 properties, and 30 properties are displayed in properties window. The most useful properties are,

Alignment :

This property has 3 possible settings. The default value is 0, which means the text in the label is left-justified. If the value is 1, then the text are right-justified. If the value is 2 then the text is centered.

BorderStyle, BackStyle :

The Borderstyle has tow values. Default value is 0, which means the labels do not start out with a border. Set the value 1, and the labels resembles a text box. The BackStyle peoperty determines whether the label is transparent.

Autosize, WordWrap :

Labels can be made automatically in a horizontal direction to encompass the text we place in them. This is the function of the AutoSize property. If the WordWrap property to true, the label will grow in the vertical direction to encompass its contents, but the horizontal size will stay the same.

NAVIGATING BETWEEN CONTROLS :

Using the mouse is the most common way to move from control to control in a Windows application. *Tab order* is used in a windows application for the sequence of controls that pressing TAB moves through.

In VB, we can create the controls is the order used for the tab order. The first control we create at design time is the one that receives the focus when the application starts. If you press TAB once when the application is running, we move to the second control we created at design time, and so on.

Assigning Access Keys for Text Boxes :

By using access key we can move to the focus to the text boxes. The labels have the captions, so we can set an access key for them by using ampersand (&) in front of the letter. If the user pressed the access key for a control, such as label, that does not respond to focus events, the focus moves to the next control that will accept it in tab order.

This makes it easy to give an access key for a text box.

- Create a label for the text box.
- Set up the access key for the label.

- Then create the text box.

MESSAGE BOXES :

Message boxes display information in a dialog box on the form. Users cannot switch to another form in your application as long as VB is displaying a message box. It is used for short messages. The simplest form of the message box command looks like this,

MsgBox “ Welcome to III – CS “

Message boxes can hold a maximum of 1024 characters, and VB automatically breaks the lines at the right side of the dialog box.

We can add our own, more informative, title to a message box. For this we have to use the full form of the message box statement by adding two options to it.

Syntax of the MsgBox command is,

MsgBox MessageInbox, TypeOfBox, TitleOfBox

we can combine three different groups of built-in integer constants to specify the kind of message box.

Symbolic Constant	Value	Meaning
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
vbCritical	16	display Critical Message icon
vbQuestion	32	display Warning Query icon
vbExclamation	48	display Warning Message icon
vbInformation	64	display Information Message icon.

for example,

MsgBox (“will have yes and No Buttons “, vbYesNo)

The next group of numbers controls which button is the default button for the box. They are,

Symbolic Constant	Value	Meaning
vbDefaultButton1	0	First button is default
vbDefaultbutton2	256	Second button is default
vbDefaultButton3	512	Third button is default

for example,

```
MsgBox " Examples of buttons " , vbOKCancel + vbExclamation +  
vbDefaultButton2, "Yest Message box"
```

THE GRID :

The Grid control is used to accurately positioning the controls for applications. The properties are,

Show Grid :

We can turn the grid on or off by changing the Show Grid setting. The default setting is on. There is usually little reason to turn the grid off.

Grid Width, Grid height Boxes :

The width and Height text boxes let we set the distance between grid marks. The default is 120 twips. Change these both to 60, and the grid becomes twice as fine.

Align Controls to Grid :

The align controls to Grid check box determines whether controls automatically move to the next grid mark or whether they can be placed between grid marks.

THE ASCII REPRESENTATION OF FORMS:

The text representation of a program makes it easy to check that the properties of the various controls and forms. The ASCII representation of a form is,

1. Start a new Standard EXE project.
2. Set the caption of the form to "Form description as saved example"
3. Add a command button in the default size, in the default location, and using the default name of command1 by double-clicking on the command button tool.
4. Add a click procedure to the command button with the single line of code.

```
Private Sub Command1_Click  
    Print "You clicked me"  
End Sub
```

5. Save the form with the name ASCII.frm.

..... **I – UNIT COMPLETED**

UNIT – II

Variables, Constants and Calculations: Data Types–Variables and Constants – Val function - Arithmetic operations – Formatting data. Decisions and conditions: If Statements – Conditions – nested If Statements– Using If Statement with option buttons and checkboxes – displaying messages – Input validation – Calling event Procedures.

STATEMENTS IN VISUAL BASIC:

Comment Statement:

Remark Statements are also called as *Comment Statements* and these statements are useful in explaining the code to people. These statements are not processed by Visual Basic and so they won't make the compiled version of the code bigger.

In Visual Basic, there are two ways to indicate a comment. The most common method is to use a single quotation mark (').

```
Private Sub Command1_Click()  
    'A Comment describing the procedure could go here  
End Sub
```

Instead of single quote, The older Rem Keyword for a comment can also be used.

```
Private Sub Command1_Click()  
    Rem Comments describing the procedure could go here  
End Sub
```

Everything on a line following a comment symbol or the Rem keyword is ignored, regardless of whether it is an executable Visual Basic statement or not.

```
PrintForm 'Dump the Current window  
PrintForm : rem a bit more
```

Rem requires a colon (:) before it.

End Statement:

When Visual Basic processes an **End** statement, the program stops. In a stand-alone program, after the End statement, all windows opened by the program are closed and the program is cleared from memory.

We can have as many as End statements within a Visual Basic program, but professional programmers prefer to use only one. They place this End statement in the **Query UnLoad Event** for main form. End statements can be replaced by **UnloadMe** statement.

VARIABLES:

Variables in Visual Basic hold information (value). Whenever we use a variable, Visual Basic set up an area in the computer's memory to store the information.

Rules for Naming Variables:

- Variable names in Visual Basic can be up to 255 characters long.
- First character should be a letter.
- Can have any combination of letters, numerals and underscores.
- Variables are not Case Sensitive.
- Usage of Special symbol is restricted.
- Mixed case variable names can be used. Eg. InterestRate
- It always changes the names of the variables to reflect the capitalization pattern.
- Reserved words cannot be used as variable names.

Variable Assignment Statements:

Assignment statements are used to give a Visual Basic variable a (new) value. Assignment statements are means of copying information from a source to a destination. Visual Basic uses an equal sign for these operations.

General Form:

Var_name = Value

For example,

Interest Rate = .05

```
NewRate = .05 + .1
```

Assigning Values to Variables

After declaring various variables using the Dim statements, we can assign values to those variables. The general format of an assignment is

```
Variable=Expression
```

The variable can be a declared variable or a control property value. The expression could be a mathematical expression, a number, a string, a Boolean value (true or false) and more.

The following are some examples:

```
firstNumber=100  
secondNumber=firstNumber-99  
userName="John Lyan"  
userpass.Text = password  
Label1.Visible = True  
Command1.Visible = false  
Label4.Caption = textbox1.Text  
ThirdNumber = Val(usernum1.Text)  
total = firstNumber + secondNumber+ThirdNumber
```

SETTING PROPERTIES WITH CODE:

The variable assignments use the equal sign for resetting the properties. If we want to change a property setting with code, place the object's name followed by a period and then name of the property on the left side of the equal sign, and put the new value on the right hand side;

Object.property = value

For example

```
Text1.Text = " "
```

Since there is nothing between the quotation marks, the next assigned to this property is blank. Similarly, a line like,

```
Text1.Text = "Welcome to BCA"
```

In an event procedure changes the setting for the text property to the text in the quotation marks. If we want to change the caption of the command button called command1 to OK, then do like this.

```
Command1.Caption = " OK"
```

Default properties :

Every VB object has a default property that is text boxes have a text property. When referring to the default property, we don't need to use the property name. for example,

```
Text1 = " This is new text "
```

Boolean properties :

Properties that take only the value True or False are called Boolean properties for example Visible property of a control is Boolean property.

```
Command1.Visible = False
```

In an event procedure hides the command button by resetting the Visible property to be false. The control stays hidden until VB processes the statement

Command1.Visible = True

The Not Operator and Boolean properties :

The usual way to toggle Boolean properties is with the Not operator, that is

Command1.Visible = Not(Command1.Visible)

In an event procedure. This statement finds the current value of Command1.Visible, and then the Not operator reverses this value, that is if the value was true, it changes to false, and vice versa.

DATA TYPES:

Visual Basic handles 14 standard types of data. There are various kinds of Numeric data types, the String data types and even Boolean Data type.

Numeric Data Types

Numeric data types are types of data that consist of numbers, which can be computed mathematically with various standard operators such as add, minus, multiply, divide and more. Examples of numeric data types are examination marks, height, weight, the number of students in a class, share values, price of goods, monthly bills, fees and others.

In Visual Basic, numeric data are divided into 7 types, depending on the range of values they can store. Calculations that only involve round figures or data that does not need precision can use Integer or Long integer in the computation. Programs that require high precision calculation need to use Single and Double decision data types, they are also called floating point numbers. For currency calculation , you can use the currency data types. Lastly, if even more precision is required to perform calculations that involve a many decimal points, we can use the decimal data types.

Numeric Data Types

Table 5.1: Numeric Data Types

Type	Storage	Range of Values
Byte	1 byte	0 to 255
Integer	2 bytes	-32,768 to 32,767
Long	4 bytes	-2,147,483,648 to 2,147,483,648
Single	4 bytes	-3.402823E+38 to -1.401298E-45 for negative values 1.401298E-45 to 3.402823E+38 for positive values.
Double	8 bytes	-1.79769313486232e+308 to -4.94065645841247E-324 for negative values 4.94065645841247E-324 to 1.79769313486232e+308 for positive values.
Currency	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	12 bytes	+/- 79,228,162,514,264,337,593,543,950,335 if no decimal is use +/- 7.9228162514264337593543950335 (28 decimal places).

Non-numeric Data Types

Nonnumeric data types are data that cannot be manipulated mathematically. Non-numeric data comprises string data types, date data types, boolean data types that store only two values (true or false), object data type and Variant data type .They are summarized in Table 5.2

Nonnumeric Data Types

Data Type	Storage	Range
String(fixed length)	Length of string	1 to 65,400 characters
String(variable length)	Length + 10 bytes	0 to 2 billion characters
Date	8 bytes	January 1, 100 to December 31, 9999
Boolean	2 bytes	True or False

Object	4 bytes	Any embedded object
Variant(numeric)	16 bytes	Any value as large as Double
Variant(text)	Length+22 bytes	Same as variable-length string

Suffixes for Literals

Literals are values that you assign to data. In some cases, we need to add a suffix behind a literal so that VB can handle the calculation more accurately. For example, we can use num=1.3089# for a Double type data. Some of the suffixes are displayed in Table 5.3.

Suffix	Data Type
&	Long
!	Single
#	Double
@	Currency

In addition, we need to enclose string literals within two quotations and date and time literals within two # sign. Strings can contain any characters, including numbers.

The following are few examples:

```
memberName="Turban, John."
TelNumber="1800-900-888-777"
LastDay=#31-Dec-00#
ExpTime=#12:00 am#
```

Managing Variables

Variables are like mail boxes in the post office. The contents of the variables changes every now and then, just like the mail boxes. In term of VB, variables are areas allocated by the computer memory to

hold data. Like the mail boxes, each variable must be given a name. To name a variable in Visual Basic, you have to follow a set of rules. All modern programming languages such as PHP (PHP runs on [hosts like iPage - see hosting review](#)) allow us developers to use variables to store and retrieve data. Each language has its own special syntax to learn.

Variable Names

The following are the rules when naming the variables in Visual Basic

- It must be less than 255 characters
- No spacing is allowed
- It must not begin with a number
- Period is not permitted

Examples of valid and invalid variable names are displayed in Table

Valid Name	Invalid Name
My_Car	My.Car
this year	1NewBoy
Long_Name_Can_beUSE	He&HisFather acceptable * & is not

Declaring Variables Explicitly

In Visual Basic, it is a good practice to declare the variables before using them by assigning names and data types. They are normally declared in the general section of the codes' windows using the **Dim** statement. You can use any variable to hold any data , but different types of variables are designed to work efficiently with different data types . The syntax is as follows:

Dim VariableName As DataType

If you want to declare more variables, you can declare them in separate lines or you may also combine more in one line , separating each variable with a comma, as follows:

Dim VariableName1 As DataType1, VariableName2 As DataType2, VariableName3 As DataType3

Example

Dim password As String

Dim yourName As String

Dim firstnum As Integer

Dim secondnum As Integer

Dim total As Integer

Dim doDate As Date

Dim password As String, yourName As String, firstnum As Integer

Unlike other programming languages, Visual Basic actually doesn't require you to specifically declare a variable before it's used. If a variable isn't declared, VB will automatically declare the variable as a Variant. A variant is data type that can hold any type of data.

Example

*Dim yourName as String * 10*

yourName can hold no more than 10 Characters.

String:

The *string* data type holds characters. One method of identifying variables of this type is to place a dollar sign (\$) at the end of the variable name.

AStringVariable\$

Once the dollar sign is added, that variable can only hold strings. String variables can theoretically hold about 2 billion characters. One of the common usages of string type is picking up characters from a text box.

Integer:

Integer variables hold relatively small integer values (between - 32768 and + 32767). Integer arithmetic is very fast. One way to make sure that a variable will only be capable of holding integers is to use the percent sign (%) at the end of the variable name.

AnIntegerVariable% = 3

Long Integer:

These variables hold integers between -2, 147,483, 684 and +2, 147, 483, 647. The identifier used for these variables is the ampersand (&). Long integer arithmetic is also fast, and there is very little performance penalty on modern machines. They do take up twice as much memory per variable as the smaller integers.

```
A LongIntegerVariable& = 1234567890
```

Single Precision:

It is used to represent numbers with decimal points. The least accurate is called single precision. They have a decimal point, but an accuracy is only of seven digits. The size (range) of these numbers is up to 38 digits. Arithmetic with these numbers is slower than with integer or long integer variables. For single precision numbers, the identifier used to hold a single precision number is an exclamation point (!).

Double precision:

This data type is used when we need numbers with up to 16 places of accuracy; they will also allow us more than 300 digits. Calculations are also approximate for these variables and relatively slow. Double precision variables are used in scientific calculations in Visual Basic. The identifier used for these variables is a pound sign (#). The # sign should also be used at the end of the actual number. For example, if we write

```
A DoublePrecisionVariable# = 12.34#
```

Currency:

This type can have 4 digits to the right of the decimal place and up to 15 to the left of the decimal point. Arithmetic will be exact within this range. The identifier used for currency variable is an “at” sign @. While calculations other than addition and subtraction are about as slow as for double precision.

Acurrency Variable@ = 12.345@.

Date:

The date data type is used to store both data and time information between midnight on January 1, 100 to midnight on December 31, 9999. The date variables are surrounded with two #'s.

For example,

Millenium = #January 1, 2005#

Millenium = #Jan 1, 2005#

Millenium = #1/1/2005#

If time is not included in a date, Visual Basic assumes it is midnight. We can use ordinary AM/PM for time or a 24 – hour clock as in the following examples:

PreMillenium = #December 31, 1999 11: 59 PM#

PreMillenium = #December 31, 1999 23:59#

Byte:

This byte type was added to Visual Basic 5; it can hold integers between 0 and 255. This is a great convenience when we need to save space, and it makes certain arrays much smaller then they

would have been in earlier versions of Visual Basic. It is also needed for handling binary files in versions of Visual Basic after version 5.

Boolean:

Use the Boolean type when we need data to be either True or False. It is considered good programming practice to use this data type rather than integers for True / False Values.

Variant:

The variant data type was added to Visual Basic way back in version 2. The variant data type is designed to store all the different possible Visual Basic data received in one place.

If we don't tell Visual Basic the type of information a variable holds, it will use this data type. Visual Basic automatically performs conversions of data stored in variants to data of another type when required.

Declaring the type of Variables:

The type of a variable is specified, using a new keyword, Dim. The technical term for these kinds of statements is declarations but many people simply call them 'Dim statements',

For Example,

```
Dim Years As Integer
```

```
Dim Rate As Currency
```

```
Dim Text Box As String
```

We can combine declarations on a single line, for example:

```
Dim years As Integer, Rate As Currency, Name As String
```

We can also write as below if we prefer using a type identifier.

```
Dim years%, Rate@, Names$
```

To give a variable the variant data type, just use the dim statement without any As clause or identifier:

```
Dim x
```

We can also use

```
Dim Foo As Variant
```

Requiring Declaration of Variables:

By default, Visual Basic allows users to create a variable without declaring it. By using the keyword **Option Explicit**, Visual Basic restricts users to declare all the variables before using them. The **Option Explicit** Statement should be placed in the (General) section of your code window.

After Visual Basic processes an **Option Explicit** command, it will no longer allow we to use a variable unless we declare it first. If we try to use a variable without declaring it, Visual Basic will pop up an error message.

To insert **Option Explicit** command automatically into the (General) section of all the forms and other code modules, choose **Tools / Options** and then go to the **Editor** page in order to require variable declaration.

Changing the Default for the Type of a Variable:

In many situations, it is sometimes convenient to change the defaults built into Visual Basic so that variables declared without a type specification identifier are no longer variants. We can change the default variable types with what is called a Def Type statement.

For example:

DefInt s-z

DefType Statement	What it does
DefInt A - Z	Changes the default-all variables to integer
DefInt I - J	All variables beginning with I & J default to being integer variables
DefInt S - Z	All variables beginning with the letter S through Z default to string

The general forms of the various DEfType Statements are:

DEfLng letter range (for long integers)
DefSng letter range (for single precision)
DefDb1 letter range (for double precision)
DefCur letter range (for currency)
DefStr letter range (for Strings)
DefVar letter range (for variants)
DefBool letter range (for Booleans)
DefByte letter range (for bytes)

DefDate letter range (for dates)

DefType statements should be placed in the (General) section of the code.

WORKING WITH VARIABLES:

When we assign one variable to another, we lose what was there before. Swapping is done directly. Suppose we have two variables, x and y, then

```
x = y
```

```
y = x
```

This doesn't work. The solution is to use a temporary variable as shown below:

```
temp = x
```

```
x = y
```

```
y = temp
```

Scope of Variables:

One of the most important aspects of Visual Basic programming is *Variable Scope*. There are three levels of variable scope in Visual Basic, as follows:

- Variable declared in procedures are private to the procedure.
- Variables declared at the form or module level in the form or module's (General) section using `Dim`, `ReDim`, `Private`, `Static` or `Type`, are form or module level variables. These variables are available throughout the module.
- Variables declared at the module level in the module's (General) section using `Public` are global and are available throughout the project, in all forms and modules

Scope of Declaration

Other than using the `Dim` keyword to declare the data, you can also use other keywords to declare the data. Three other keywords are `private`, `static` and `public`. The forms are as shown below:

Private *VariableName* as Datatype

Static *VariableName* as Datatype

Public *VariableName* as Datatype

The above keywords indicate the scope of declaration. Private declares a local variable, or a variable that is local to a procedure or module. However, Private is rarely used, we normally use Dim to declare a local variable. The Static keyword declares a variable that is being used multiple times, even after a procedure has been terminated. Most variables created inside a procedure are discarded by Visual Basic when the procedure is finished, static keyword preserve the value of a variable even after the procedure is terminated. Public is the keyword that declares a global variable, which means it can be used by all the procedures and modules of the whole program.

MORE ON STRINGS:

A string is a collection of characters surrounded by double quotes. When we enter information into a text box, VB stores that information as a string. The operations of string is concatenate two strings. To do this operation, use ampersand (&).

For example,

```
Title$ = "AVS"
```

```
Name$ = "COLLEGE"
```

```
' gives AVS COLLEGE
```

```
Collegename = Title$ & Name$
```

The & joins the strings in the order in which they present.

Evil Type Conversions I : beware of the + sign for Strings :

VB also use a + sign to join together strings. Although the + sign works in VB to join strings together, don't use it, it can lead incredibly hard to find bugs.

ASCII / ANSI Codes:

Usually, the code for translating text to numbers is called the ASCII code. It associates with each number from 0 through 255 control characters., although Windows cannot display all 255 ASCII characters and uses a more limited set of characters called the ANSI character set.

The value of the function Chr(n) is the string consisting of the character of ASCII value n. the statement is,

```
Print Chr(n)
```

Either displays the character numbered n in the ASCII sequence for the font currently in use or produces the specified effect that the control code will have on screen.

Fixed – Length Strings :

A fixed- length string is a special type of string. These variables are created with a Dim statement.

```
Dim ShortString As String * 10
```

```
Dim strShort As String * 10
```

Both of these statements set up string variables. This variables will always hold strings of length 10. If we assign a longer string to ShortString, then

```
ShortString = "AVScollegeofarts&science"
```

We get the same thing as

```
ShortString = "AVScollege"
```

MORE ON NUMBERS :

Numeric variables can be declared as

Dim I as integer

I = 1,234

* Usage of the decimal point is accepted

* If the value is assigned to long variable it will be rounded.

The Numeric Operators :

To compute inputs from users and to generate results, we need to use various mathematical operators. In Visual Basic, except for + and -, the symbols for the operators are different from normal mathematical operators, as shown in Table 6.1.

Arithmetic Operators

Operator	Mathematical function	Example
^	Exponential	$2^4=16$
*	Multiplication	$4*3=12,$
/	Division	$12/4=3$
Mod	Modulus (returns the remainder from an integer division)	$15 \text{ Mod } 4=3$
\	Integer Division(discards the decimal places)	$19\backslash 4=4$
+ or &	String concatenation	"Visual"&"Basic"="Visual Basic"

Conditional Operators

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

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

Logical Operators

In addition to conditional operators, there are a few logical operators that offer added power to the VB programs. They are shown in Table 7.2.

Operator	Meaning
And	Both sides must be true
Or	One side or other must be true

Xor	One side or other must be true but not both
Not	Negates truth

* You can also compare strings with the operators. However, there are certain rules to follow where upper case letters are less than lowercase letters, and number are less than letters.

Parenthesis & Precedence:

The following table gives the order of operations:

- Exponential (^)
- Negation (Making a number negative)
- Multiplication & division
- Integer division
- The remainder (Mod) function
- Addition & subtraction

Conversion function:

Conversion function	Explanation
CInt	Converts to an integer
CLng	Converts to long integer
Csng	Converts to single precision
CDbl	Converts to double precision
CCur	Converts to the currency data

The Rnd function:

Rnd – this function will generate a random number between the values 0&1 it can be 0, but not 1

Print rnd

it means it could generate values like

.75432

.2439 etc

the next number can be obtained by linear congruent method

$\text{nextno} = (A * \text{previous no} + B) \bmod m$

where a,b,m are values given from users

The val function:

Val: This is to obtain value in integers

Val("sss") = 30

Val (7.5 % = 7.5

Converting numeric to string :

Str- is the reverse function of val

Str(30) = "30"

If you use numbers in your program and do not assign them to variable of variant type VB assumes the following

1. If a number has no decimal point and is in range -32768 to 32767 it is integer
2. If a number has no decimal point and is in range for long int (-2,147,483,648 to 2,147,483,647) its long integer
3. If a number has a decimal point and is in range for single precision number it is assumed to be single precision
4. If a number has decimal point and is outside range for single precision number it is assumed to be double precision

CONSTANTS :

Constants are fixed values that do not change during the execution of the program. These are declared like variables.

Rules :

- 255 characters
- First character a letter
- Then any combination of letters, underscores and numerals.

We can set up a constant by using the keyword Const followed by the name of the constant, an equal sign, and then the value.

Const Pie = 3.14159

We can also set up a string constants,

```
Const Username = "jaya"  
Const Language = "VB 6.0"
```

We can use numeric expressions for constants,

```
ConstPieover = Pie / 2
```

INPUT BOXES :

The input boxes are used to get information from the user. But the InputBox function displays a modal dialog box on the screen, like a message box

Input boxes have a title bar. There are two command buttons labeled OK and Cancel. Finally, there is a text area at the bottom.

The syntax is,

```
StringVariable = InputBox(PromptString)
```

This form uses the name of the project in the title bar of the input box. The full syntax for InputBox function is,

```
VariableName = Inputbox(prompt[, title][,default][, xpos][, ypos][, helpfile, context])
```

Where,

- The prompt parameter is a string or string variable whose value VB displays in the dialog box.
- The title parameter is optional and gives the caption used in the title bar. There is no default value.
- The default, xpos and ypos parameters are also optional.
- The two parameters helpfile and context are used together when we have a help message attached to the box.

DISPLAYING INFORMATION ON A FORM:

VB displays text on a form using the print method. The syntax for the print method to a form is,

FormName.Print expression

Where the expression is any VB expression that can convert to a string. VB uses whatever settings are current for the Font object for the current form in order to determine the font information to use when it prints.

There are even more problems we need to be aware of, these problems will potentially occur when the form is either,

- Minimized and then restored
- Covered by another window
- Enlarged or shrunk

Unless the AutoRedraw property of the form is set to true, the text will disappear.

- Start up a new project
- Change the font property so the font size is, say, 18 points
- Put the following code in the form_load:

```
Private Sub Form_Load ( )  
    Show  
    Print " welcome "  
End Sub
```

Now run the program. There is no problem. Next, minimize the form and restore it, the text has disappeared.

Current X and Current Y :

In many fonts, letters like “m” take up more space than letters “i”. fonts in which all characters are the same width are called non-proportionally spaced fonts. Fonts where characters may be of different widths are called proportionally spaced fonts.

Courier new is the most common non-proportionally spaced font and Arial is a common proportionally spaced font. For example,

```
Welcome to AVS
```

```
Welcome to CS
```

proportionally spaced font gives a more polished look. Most fonts in a Windows are proportional, this makes it more difficult to position text accurately compared to older text-based systems. We cannot simply move to the left and have that be the same location for all fonts.

VB always report the current position as the values of two properties of the form or picture box. That is CurrentX and CurrentY.

- CurrentX refers to the horizontal position where VB will display the information.
- CurrentY refers to the vertical position where it will display the information.

For example, If the scale mode is pixels, and before each print statement we set the CurrentX position to be 100, then all text will start 100 pixels over from the left side of the form.

```
FormName.CurrentX = value
```

```
FormNmae.CurrentY = value
```

The value may be any numeric expression from which VB can extract a single-precision value.

THE FORMAT FUNCTION:

If we run the mortgage program, we may end up with 16 decimal digits when you really want the answer to look like 1.01. We can overcome this problem by replacing the Str function with a new function called Format function. This function works with a number and a template. The syntax is

```
Format(NumericExpression, FormatString$)
```

And this gives us a copy of the original expression in the form of a string that has the correct format. For example,

```
Me.print Format (123.456789, "###.##")
```

Yields a string "123.456" that will be printed on the form. When you use a format string like this, VB rounds the number off so there are only two digits after the decimal point. The Format function does not leave room for an implied + sign in front of the number. That is,

```
Me.Print " The interest rate is "&Format(payment, "####.##")
```

The extra space after the word "is" is essential.

Predefined Format Strings :

VB makes it even easier to deal with the most common formatting situations by adding what are called named formats to the Format function. For example,

```
Me.Print Format (amount, "Currency")
```

Instead of

```
Me.Print Format (Amount, "###, ###.##")
```

And we will get the same results in the united states. Because the Currency named format is defined to be the same as the ###,###.## format.

PICTURE BOXES :

We can use picture boxes in many different contexts, not just as passive containers for graphics or icons. The icon in the tool box for picture boxes, like image controls, can display icons, gifs, jpegs, bitmaps, and Windows metafiles.

- Bitmaps are graphical images of the screen. Each dot corresponds to one bit for black and white displays and many bits for color or gray scale displays. Image controls are used for bitmaps, when these are stores in a file, the convention is to use a .bmp extension.
- Gifs and jpegs are both formats widely used on the internet. Both are compressed formats and jpegs can be quite small.

On the other hand, for the purposes of displaying information and containing controls, we might want to think of picture boxes as being “forms within forms”. For example,

- Picture boxes have CurrentX and CurrentY properties
- We can mix fonts and font sizes when we print to a picture box.
- We can add controls to a picture box by working with the toolbox in the same way that we would add controls to a form.

The main difference between picture boxes and forms is that we use the Height and Width properties of the picture box rather than the ScaleHeight and Scale Width properties of the form.

Disadvantages :

- Information displayed in a picture box will not be obscured by any controls on the form.
- Picture boxes are less memory hungry
- We can have more than one picture box on a single form.

RICH TEXTBOXES :

Rich TextBox control is a custom control. We need to add it to the toolbox if it is not already exists there. To do this;

1. Choose project|Components to open the components dialog box
2. Choose MS Rich TextBox control 6.0.

This control displays text with multiple fonts and sizes without having to go out and buy a third-party custom control. And it is not limited to 32 K characters like an ordinary text box.

Properties of RichTextBox control ;

- SelLength === > returns or sets the number of characters selected
- SelStart === > returns or sets the starting point of the selected text.
- Seltext === > return or sets a string equal to the currently selected text.
- SelColor === > sets the color of the currently selected text and of all text added after the current insertion point.
- SelfontName === > used to change the font.
- SelfontSize === > used to change the size of the currently selected text.

THE PRINTER OBJECT :

VB uses the printer that is currently set up as the default printer in the control panel. It makes easy to use whatever resolution, font properties, and so on that the printer driver in windows can coax from the printer.

The PrintForm command sends a screen dump of a form to the printer. If our application has more than one form, then we have to use the form name in this command.

```
FormName.PrintForm
```

We can send information to a printer is the Print method applied to the printer object. For example, because th print method is page oriented, we can set the CurrentX and CurrentY properties

to precisely position text or even dots on a page. The syntax used to send text to the printer is similar to that used for forms or picture boxes.

```
Printer.PrintTextToPrint
```

Properties and methods for the printer :

ColorMDe :

This is used to determine whether a color printer prints in color or monochrome.

vbPRCMMonochrome1		prints output in monochrome
vbPRCMColor	2	prints output in color.

Copies :

Used to set the number of copies to be printed.

Height, Width :

These properties give the height and width of the paper in the printer as reported by windows.

EndDoc

This method tells Windows that a document is finished. The syntax is

```
Printer.EndDoc
```

NewPage :

This method ends the current page and tells the printer to move to the next page. The syntax is,

```
Printer.NewPage
```

PrintQuality :

This is used to set the quality of the printed output- if the printer driver supports it. The syntax is,

```
Printer.PrintQuality = value
```

We can use four built-in- constants, they are,

- vbPRPQDraft -1 Draft resolution
- vbPRPQLow -2 Low resolution
- vbPRPQMedium -3 Medium resolution
- vbPRPQHigh -4 High resolution

DETERMINATION LOOPS :

Repeating an operation a fixed number of times is called a determinate loop. The simplest of the determinate loop is the **For-Next** loop. Its syntax is

```
For Counter Variable = Starting Value to Ending value Step value
```

```
    Body of Loop
```

```
Next Counter variable
```

When Visual Basic encounters a For-Next loop, the following happens:

- ↪ Visual Basic first sets the Counter Variable to the starting value.
- ↪ Then it checks whether the value for the counter is less than or equal to the ending value, if the step value is not specified, Visual Basic assigns a value of 1 to the step value.
- ↪ If the above condition is satisfied, Visual Basic processes all the subsequent statements until it comes to the keyword Next.
- ↪ At this point, the counter variable is incremented or decremented depending on the step value, and the process is started again with the new counter variable.
- ↪ If the condition specified in 2 fails, the loop is finished and Visual Basic moves past it to the next statement after the keyword Next.

The following code prints the numbers 1 to 10 on the current form inside an event procedure.

```
Dim I As Integer
For I = 1 to 10
Print I
Next I
```

Here, Visual Basic assumes a value of 1 to the step variable. The code given below prints number from 10 to 1

```
Dim I As Integer
For I = 10 To 1 Step - 1
Print I
Next I
```

Nested For-Next Loops:

Placing one For – Next loop inside another is called nested For – Next loops. The following example prints the entire multiplication table from 2 to 12.

```
For J% = 2 To 12
```



```
For I% = 2 To 12  
    Print I% * J%  
Next I%  
Next J%.
```

Here the value of J% starts out at 2, and then Visual Basic enters the inner loops. The value of x% starts out at as well. Now visual Basic makes 11 passes through the loop before it finishes. Then it process the extra print statement before it processes the Next J\$ statement. At this point, Visual Basic changes the value of J% to 3 and starts the processes again.

INDETERMINATE LOOPS :

These loops must either keep on repeating an operation or not, depending on the results obtained within the loop. The following pattern shows how to write this type of loop in Visual Basic:

```
Do  
    Visual Basic Statements  
Until Condition is met
```

The following are the relational operators used in addition to the equal (=) operator.

Do ... Loop Until:

This looping construct has the following syntax

```
Do  
    Visual Basic Statements  
Loop Until <Condition>
```

If the condition specified is met, Visual Basic executes the statement next to Loop Until <Condition> Otherwise, the statements within the Do ... Loop Until <Condition> is executed repeatedly. The Do ... Loop Until statement is executed at least once.

The following example asks input from the user repeatedly until the input is "AVS".

```
Private Sub Form_Load()  
    Dim X$  
    Do  
        X$ = InputBox ("Password Please")  
    Loop Until X$ = "AVS"  
End Sub
```

Here, if X\$ = "AVS", Visual Basic comes out of the Do... Until loop. Otherwise, it keeps on asking the password from the user repeatedly.

Do Until Loop:

The body of this looping construct is executed only when the condition specified is false. Its syntax is

```
Do Until <Condition>  
    Visual Basic Statements  
Loop
```

As an example, the following code uses a variable NameCount to count the number of Names entered.

```

NameCount = 0

Entry$ = InputBox ("Name – ZZZto end")

Do Until Entry$ = "ZZZ"

NameCount = NameCount + 1

Entry$ = InputBox (("Name – ZZZto end"))

Loop

```

Here the user types the first name before the loop starts. Now the program does an initial test. The loop is entered, and 1 starts being added to the counter only if this test fails.

The Do – While Loop:

These “loops consist of replacing the keyword Until with the keyword While. A Do-Until loop can be changed into a Do-While loop by reversing the relational operator. For example

```

Do

Loop Until Number > 5

```

Is the same as

```

Do

While Number <= 5

```

Do Loops With And, Or, Not: These logical operators are used to combine the conditions specified in the Do Loops. And, Or logical operators reside between two conditions whereas Not logical operator comes and a text box is empty, then

```

Do While Number > 0 And Text 1.Text = ""

```

Is much easier to specify than

```
Do Until Number <=0 Or Text1.Text <> ""
```

Although they both mean the same thing.

While – Wend Loop:

There is open other loop possible in Visual Basic. To preserve compatibility, Visual Basic allows a variant on the Do-While loop. For example,

```
Do While x = 0
```

Is the same as

```
Loop
```

```
While X = 0
```

```
Wend
```

Example :

```
Dim sum, n As Integer
Private Sub Form_Activate()
List1.AddItem "n" & vbTab & "sum"
While n <> 100
n = n + 1
Sum = Sum + n
List1.AddItem n & vbTab & Sum
Wend
End Sub
```

MAKING DECISIONS :

Conditional statement :

The If – Then statement is used for making decisions in Visual Basic. When visual Basic encounters an If- Then statement, it checks whether the first clause is True. If that clause is True, the computer does whatever follows. If the test fails, processing steps to the next statement. For example, a statement such as

If A <= B Then Print A “ is no more than” & B

Tests for numerical order if A and B are numeric variables.

The Else:

When Visual Basic processes an If – Then – Else, if the test succeeds, Visual Basic processes the statement that follows the keyword Then. If the test fails, Visual Basic process the statement that follows the keyword Else.

Combining Conditions in an If – Then:

The keywords And, Or and Not can also be used in an If – Then to check more than one condition.

The Block If – Then:

In many situations, there is a need to process multiple statements if a condition is True or False. This Block If – then lets we process as many statements as we like in response to a True condition. These multiple statements are enclosed within if and EndIf.

Exiting a Loop Prematurely:

There are various commands in Visual Basic to come out of a looping construct. Whenever Visual Basic processes the Exit Do statement, it pops we out of the loop, directly to the statement

following the keyword Loop. Similarly, the command Exit For can be used to exit from the For – Next loop.

Visual Basic places no restriction on the number of Exit statements we place inside a loop. Most programmers use the Exit Do or Exit For only for abnormal exits from loops, such as when a program is about to divide by zero.

SELECT CASE :

The Select Case command makes it clear that a program has reached a point with many branches. For example, we were designing a program to compute grades based on the average of your example, If the average was 90 or higher, the person should get an a, 80 to 89, a B, and so on, then,

```
Select Case Grade  
Case Is > 90  
Your Grade = "A"  
Case Is > 80  
YourGrade = "B"  
Case Is > 70  
Your Grade = "C"  
Case Else  
Print "Please retake the final"  
End Select
```

Here, Case Else is the default case. The Case Else should always be the last case in a Select Case. The Select Case control structure allows we to combine many tests for equality on one line. For example,

```
Case "A", "E", "I", "O", "U",  
Print "Letter is a Vowel"
```

NESTED IF-THEN'S :

One If statement may occur in another If statement is called Nested If-then statement. The syntax is,

```
If ..... Then  
.....  
.....  
Elseif ..... Then  
.....  
.....  
Else  
.....  
.....  
End If
```

THE GOTO STATEMENT:

Like most programming language, Visual Basic retains the unconditional jump or Go To. To use a Go To in Visual Basic, we must label a line. Labels must begin with a letter and end with a colon. They must also start in the first column.

For example, suppose we are using a nested For loop to input data and want to leave the loop if the user enters ZZZ, then

```
For I = 1 To 10  
For J = 1 To 100  
GetData = InputBox ("Enter Data - ZZZ to end")
```

```

If GetData = "ZZZ" Then
Go To BadInput
Else
  `Process Data
EndIf
Next J
Next I
Exit Sub
BadInput:
  Msg Box "Data entry ended at User request"

```

The Exit Sub keywords are used to prevent from falling into the labeled code.

STRING FUNCTIONS :

All kinds of string manipulations will require mastering Visual Basic's powerful string-handling functions.

Chr () :

The value of the function Chr (n) is the string consisting of the character of ASCII value n. For Example, the statement,

```
Print Chr (97)
```

prints the letter "a".

Asc () :

This function takes a string expression and returns the ASCII / ANSI value of the first character. If the string is empty (the null string), using this function generates a run-time error. For example,

```
Print Asc ("a")
```

prints the ASCII / ANSI value 97 for the character "a".

Space () :

This is used to build up a string of spaces. For example,

```
Space (NumberOfSpaces)
```

gives us a string consisting of only spaces. With the number of spaces we can determine the value inside the parentheses.

String () :

This function is used to build up a string of repeated characters. For instance,

```
String (Number, StringExpressin$)
```

gives us a string (in the form of a variant) of repeated character. For example,

```
X$ = String (10, "z")
```

Trim () :

The Trim function (Trim\$) removes spaces from both the left and right ends of a string. For example,

```

A$ = "          This has far too many spaces.          "
B$ = Trim$(A$)
          B$ = "This has far too many spaces".

```

Similarly, LTrim (LTrim\$) removes spaces from the left end, and RTrim (RTrim\$) removes spaces from the right.

LCase() , UCase() :

These functions are used to change the case of the letters. The LCase (LCase\$) function forces all the characters in a string to be lower case, Similarly, UCase (UCase\$) switches all the characters in a string to upper case.

```

[
S1$ = LCase$ ("HOW ARE YOU")           `gives how are you

```

Len() :

In Visual Basic, the function that tells you the length of a string is Len (), where the parentheses following the function hold the string expression. This function counts all spaces and non-printing characters that appear in the string.

Mid() , Left() and Right() :

The Mid function returns a substring stored in a variant, whereas the Mid\$ function, returns an actual substring. The syntax for these functions is:

```

Variable = Mid (String, Start[, Length])
Variable = Left (String, NumberOfCharacters)
Variable = Right (String, NumberOfCharacters)

```

The functions Left (Left\$) and Right (Right\$), make copies of characters from the beginning of a string or the end. The Mid function makes changes within a string but never changes the length of the original string.

InStr () :

The InStr function tells you whether a string is part of another string, and, if it is, InStr tells you the position at which the substring starts. The usual form of the InStr function looks like

InStr ([where to start,] string to search, string to find [, compare])

InStrRev () :

This function starts searching from the end of the string. The InStrRev function looks like

InStrRev (original string, substring[, compare])

Replace () :

This function is used to find a substring in a given string and replaces the substring with another substring, if the substring occurred in the original string. The syntax for the Replace function is:

Replace (Expression, find, Replace With[, Start[, Count[, Compare]])

StrComp () :

This function can be used instead of the relational operators to compare strings. By adding a third argument to StrComp, one can control the case sensitivity of the comparison.

NUMERIC FUNCTIONS:

Parenthesis & Precedence:

The order of operations are:

- ↩ Parenthesis
- ↩ Exponential (^)
- ↩ Negation (Making a number negative)
- ↩ Multiplication & division
- ↩ Integer Division
- ↩ The remainder (Mod) function
- ↩ Addition & Subtraction

Conversion Functions:

Conversion function	Explanation
CInt	Converts to an Integer
CLng	Converts to long integer
CSng	Converts to single precision
Cdbl	Converts to double precision
CCur	Converts to the currency data

Rnd () :

Rnd – this function will generate a random number between the values 0 & 1 it can be 0, but not 1.

Val () :

Val() is to obtain value in integers.

Str () :

Str() is the reverse function of val, it obtain the values as strings.

Int () :

[

Int gives the floor of a number – first integer that is smaller than or equal to the number is usually called the greatest integer fraction. There is another function called Fix. Both Fix and Industry word the same way for positive numbers but are different for negative ones.

Round () :

The syntax for Round function is

Round (expressions[, Number Of Decimal Places])

where the optional second parameter allows you to round past the decimal point. If you leave it off, you get a integer.

Sgn () :

The Sgn () function gives a + 1 if what is inside the parentheses is positive, - 1 if negative, and a 0 if it's zero.

Abs () :

The Abs function gives the absolute value of whatever is inside the parentheses. All this function does is remove minus signs.

$$\mathbf{Abs (-1) = 1 = Abs (1)}$$

Sqr () :

The Sqr function returns the square root of the numeric expression inside the parentheses, which must be non-negative or a run-time error follows.

Exp () :

The Exp function gives e ($e \approx 2.71820$) to the power x, where e is the base for natural logarithms and x is the value in the parentheses. The answer is single precision if x is an integer or is itself a single precision number; otherwise, the answer is a double-precision number.

Log () :

The Log() gives the natural logarithm of a number. To find the common log (log to base 10) use

$$\mathbf{Log_{1.0}(x) = Log (x) / Log (10)}$$

which gives the common logarithm of the value (which must be positive) inside the parentheses.

Trigonometric Functions:

Visual Basic has the built-in trigonometric functions Sin (Sine), Cos (Cosine) and Tan (Tangent). The angle inside the parentheses should be in radians. To convert from degrees to radians, the value of π is needed. The formula is

$$\text{Radians} = \text{degrees} * \pi / 180$$

The following table summarizes the inverse trigonometric functions, as well as some other useful functions.

Function	Result
Pi = 4 * Atn (1#)	Value of π in double precision
E = Exp (1#)	Value of e in double precision
Degrees to radians	radians = degrees * π / 180
Radians to degrees	degrees = radians * 180 / π
Sec (x)	1 / Cos (x)
Cosec (x)	1 / Sin (x)
Cot (x)	1 / Tan (x)
ArcCos (x)	Atn (x / Sqr (-x* x + 1)) + π / 2
ArcSin (x)	Atn (x / Sqr (-x * X + 1))
ArcCot (x)	Atn (x) + π / 2
Cosh (x)	(Exp (x) + Exp (-x))/2
Sinh (x)	(Exp (x) – Exp (-x)) /2
Log ₁₀ (x)	Log (x) / Log (10)
Log _a (x)	Log (x) / Log (a)

DATE AND TIME FUNCTIONS :

Date () :

The Date function returns a date of the form mm-dd-yyyy for the current date. The month and day always use two digits; the year uses four digits. This function can also be used as a statement to reset the current date in the system.

The time in the system clock can also be read or temporarily reset it with the time function. The Time function returns an eight-character date of the form hh:mm:ss.

Example of Time Command	Effect
Time = "hh"	Sets the hour; minutes and seconds are set to 0.
Time = "hh:mm"	Sets the hour and minutes; seconds are set to 0.
Time = "hh:mm:ss"	Sets the hour, minutes and seconds

Date Value (String) :

It yields an expression of Date type representing the date defined by the string expression inside the parentheses. This function can also accept string of different date formats. Some more functions are:

Now	Returns the date and time as stored in the system clock.
Date	Returns the current date.
Time	Returns the current time.

The Date Serial Function returns a number that can be used for date calculations. The syntax is

Date Serial (Year, Month, Day)

FINANCIAL FUNCTIONS:

Pmt () :

This function returns the payment for an annuity based on periodic, constant payment and a constant interest rate. An annuity is a series of payments made over time. The syntax for the Pmt function is:

Pmt(*RatePerPeriod, NumPeriods, What YouStart With, WhatYouEndUpWith, When Do You Pay*)

FV () :

This function gives the future value of an annuity based on periodic payments (or withdrawals) and a constant interest rate. The syntax is

FV (*InterestRatePerPeriod, NumPeriods, PaymentPerPeriod [,StartAmount[, WhenDue]]*)

IPmt () :

This give the interest paid over a given period for an annuity based on periodic, equal payments and a constant interest rate. The syntax is

IPmt (*rate, per, nper, fv, due*)

rate is the interest rate per period, and per is the period in the range 1 through the number of periods (nper).

NPer () :

This function tells how long it will take (the number of periodic deposits / withdrawals) to accumulate (disburse) an annuity. The syntax is

NPer (rate, pmt, pv, fv, due)

PV () :

This function tells how much periodic payments made over the future are worth now. The syntax is

PV (rate, nper, pmt, fv, due)

The rate is the interest rate per period, nper is the total number of payments made, pmt is the number of payments made each period. The fv is the future value or cash balance you want after you have received (made) the final payment.

NPV () :

The net present value function is used, for example, if you start out by paying money as start up costs but then get money in succeeding years. The syntax is

NPV (RatePerPeriod, ArrayOf ())

Rate () :

This function gives the interest rate per period for an annuity. The syntax is

Rate (nper, pmt, pv, fv, due, guess)

The Rate function uses an iterative procedure to arrive at the true interest rate. The guess parameter is usually given a value .01.

SLN() and DDB () :

These Functions return the straight line and double declining balance depreciation of an asset over a given period. The syntaxes are

SLN (Cost, SalvageValue, LifeExpectancy) &

DDB (Cost, SalvageValue, LifeExpectancy, PeriodOfCalculation)

IRR() and MIRR() :

These functions give versions of the internal rate of return for a series of payments and receipts. IRR gives the ordinary internal rate of return, and MIRR gives the modified rate in which you allow payments and receipts to have different interest rates. The syntax is

IRR (ValueArray(), Guess)

MIRR (ValueArray(), FinanceRate, ReInvestRate)

----- II – UNIT COMPLETED-----

UNIT – III

Menus, Sub procedures and sub functions: Menus – Common Dialog Boxes – Writing General Procedures. Multiple Forms: Multiple Forms – Standard Code Modules- Variables and Constants in Multiple-Form Projects.

PROCEDURES AND FUNCTIONS:

There are two kinds of general procedures in Visual Basic:

- ↳ Function Procedures
- ↳ Sub Procedures

Function Procedures or **User-Defined Functions** have self-contained piece of code designed to massage data and return a value.

Sub Procedures are smaller “helper programs” that are used as needed, and do not return any value.

The Subprograms and function procedures will do the following:

- Help we break down large tasks into smaller ones.
- Automate repeated operations.
- Make it clearer what it is we are trying to accomplish by ‘naming’ a piece of code.

List of Objects List of Objects

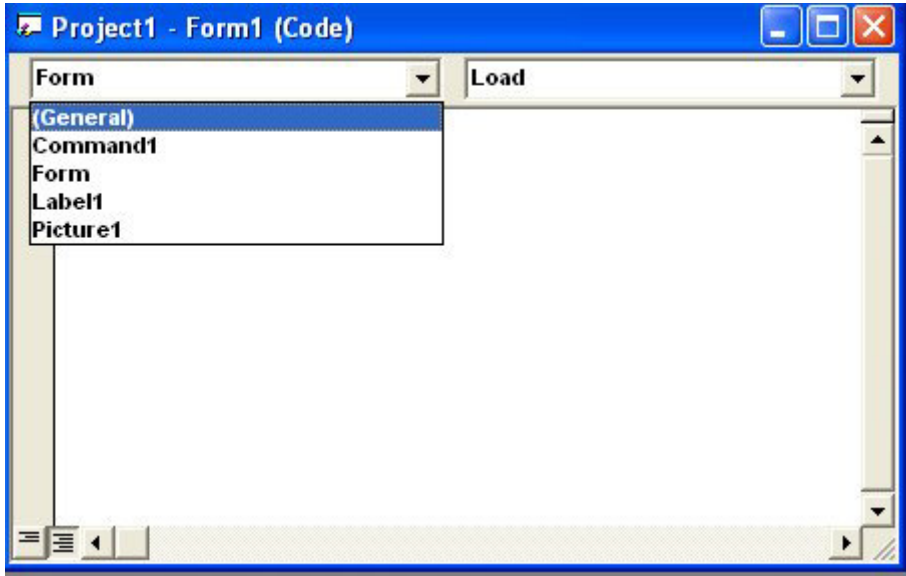
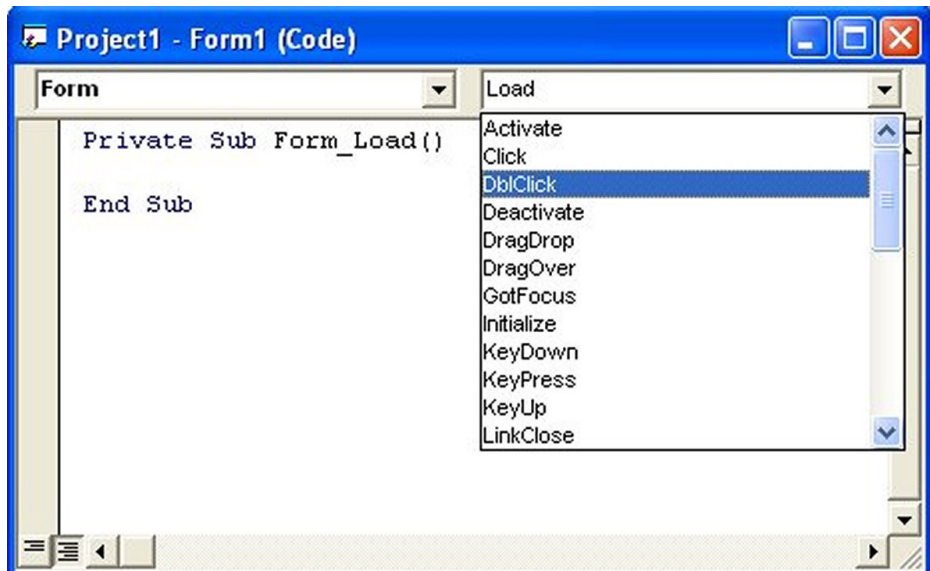


Figure 2.3: List of Procedures



FUNCTION PROCEDURES:

To insert a function in the code window, open the code window by double clicking anywhere in the form or by pressing F7. Choose Tolls | Add procedure from the Tools menu. Then click the function radio button and type a name for your function. Click on OK, and a function template for the form pops up in the code window. The following lines of code are supposed to take a string and do the following.

1. Remove all spaces at the beginning and end of the string.
2. Reduce multiple spaces inside the string to single spaces.

```
Public Function super Trim (S As String) As String  
  
Dim T As String, DS As String  
  
DS = Chr (32) &Chr (32)  
  
T = Trim (S)  
  
Do Until InStr (T, DS) = 0  
  
T = Replace (T, DS, Chr (32))  
  
Loop  
  
Super Trim = T  
  
End Function
```

The first line of the function is called the header of the function. The keyword public is called an access specified. The variable S is called the formal parameter. The data type followed by the As keyword tells about the type of the value returned by the function.

A function can be called anywhere inside the module of a form. A function procedure in Visual Basic can have any number of formal parameters, but it returns almost one value of any data type.

SUB PROCEDURES:

These procedures consist of a block of code that does something much as an Event Procedure. The general format of a sub procedure looks like

```
Sub Procedure Name (parameter1, Parameter2, ..)
Statement(s)
End Sub
```

When Visual Basic executes statements of the form

```
Procedure Name parameter1, parameter2, ...
Or the equivalent
Call procedure Name (parameter1, Parameter2, ...)
```

then the values of the parameters are passed to the corresponding parameters in the procedure, and the statements inside the sub procedure are executed. When the End Sub statement is reached, execution continues with the line that followed the call to the sub procedure.

ADVANCED USES OF PROCEDURES AND FUNCTIONS :

It is based on the fact that function or procedures you define can also use named arguments, if you choose to use this feature, the names you choose for the parameters of your function and procedures become vital. For ex: the header for our original charcount% function was simply

```
Function charcount% (x$, y$)
```

Passing Parameters:

Passing by Reference, passing by value:

When we call a function, or procedure, there are actually two ways to pass in a variable as an argument. These are called:

- Passing by reference
- Passing by value

When we pass an argument variable by reference, any changes to the corresponding parameter inside the procedure will change the value of the original argument when the procedure finishes.

When we pass an argument by value, then the original variable retains its original value after the procedure terminates regardless of what was done to the corresponding parameter inside the procedure.

Optional Arguments:

Visual Basic permits us to have optional arguments in the functions and procedures. These arguments can be of any type, but they must be the last arguments in a function or procedure.

For example,

```
Sub PAddress (Name As String, Optional Addr As String)
```

Is a sub procedure with an optional argument Addr.

One of the easiest ways to navigate through the code in we project is with the object Browser. The steps used are:

1. Bring up the Object Browser by pressing F2
2. Choose your project by name form the libraries | Projects list box.
3. When we click on the name of a form, the Members (second) column of the Object Browser will show the procedures attached to that form (including the Event Procedures).

Sub Procedures, can work from the Code Window or the Object Browser itself. It can work from both options in two different ways.

USING THE OBJECT BROWSER TO NAVIGATE AMONG YOUR SUB PROGRAMS :

One of the easiest ways to navigate through the code in your project is with the Object browser. Simply follow these steps.

1. Bring up the object Browser(f2)
2. Choose your project by name from the libraries/project list box.
3. Up to now all our project has a single have a single form and the code is attached to that form.

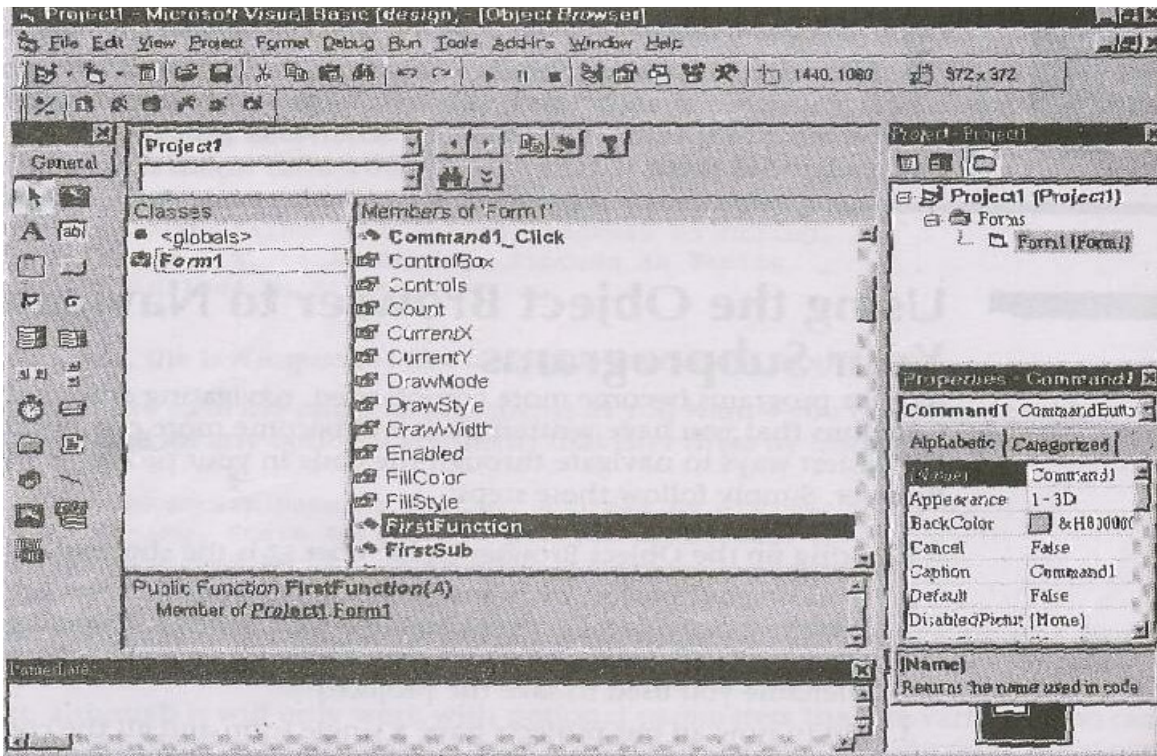
In this case, the name of the will be the same as the value of the forms name property and when you click on it, the Members(second) column of the object Browser will show the procedures attached to that form. Double click on the name of the subprogram whose code you want to work with.

Double-click on the name of the subprogram whose code you want **to** work with. It

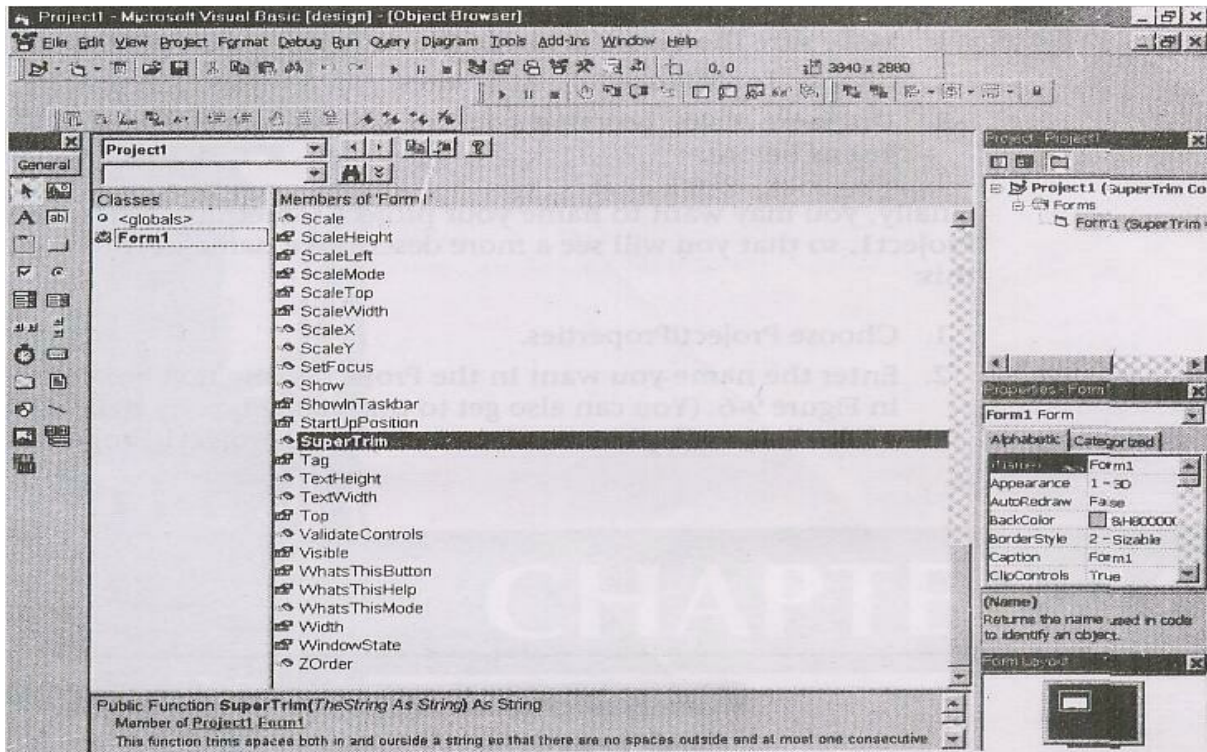
shows the Object Browser for a sample project that has three functions and three Sub procedures with the obvious names. The project also has a command button with a Click procedure on it. In that these items are in bold. This is how the Object Browser shows you that code exists for that item,

Next, remember that the easiest way to hide the Object **Browser** is **to** right-click Inside of it and then choose Hide from its context menu.

It you want to take full advantage of the Object Browser as a way to document your type. You should supply a description of your procedures that can show up in the Object Browser—just as the builders of Visual Basic did for the built-in procedures and constants. Your descriptions will also show up in the bottom pane of the object browser as for the SuperTrim test project,



Writing your own functions & procedures.



To **add these** descriptions, you can work from the Code Window or the **Object** Browser itself. To work from the Object Browser, you again use its context menu. This time, follow these steps:

1. Highlight the function or procedure you want to document.
2. Right click and choose Properties from the Context menu that pops up.
3. In the Procedure Attributes dialog box shown here, enter the description you want in the Description text **box**.

ARRAYS

Array is a collection of similar data items in same data type. **To** distinguish among the items store in an array, it **use** special **kind** of parameter called as index.

Dimension of an Array

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 with more complex problems. Let me illustrate the the arrays with tables.

One dimensional Array

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

Two Dimensional Array

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

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

Declaring Arrays

We can 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 a one dimensional array is as follow:

Dim arrayName(subs) as dataType

where subs indicates the last subscript in the array.

Example

Dim CusName(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 CusName(1) to CusName(10). Otherwise, there will be 11 elements in the array starting from CusName(0) through to CusName(10)

CusName(1)	CusName(2)	CusName(3)	CusName(4)	CusName(5)
CusName(6)	CusName(7)	CusName(8)	CusName(9)	CusName(10)

Example

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)

The general format to declare a two dimensional array is as follow:

```
Dim ArrayName(Sub1,Sub2) as dataType
```

Example

Dim StudentName(10,10) will declare a 10x10 table make up of 100 students' Names, starting with StudentName(1,1) and end with StudentName(10,10).

Sample Programs

(i) Dim studentName(10) As String

```
Dim num As Integer
```

```
Private Sub addName()
```

```
For num = 1 To 10
```

```
studentName(num) = InputBox("Enter the student name", "Enter Name", "", 1500, 4500)
```

```
If studentName(num) <> "" Then
```

```
Form1.Print studentName(num)
```

```
Else
```

```
End
```

```
End If
```

```
Next
```

```
End Sub
```

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

```

ii) Dim studentName(10) As String
Dim num As Integer

Private Sub addName( )
For num = 1 To 10
studentName(num) = InputBox("Enter the student name")
List1.AddItem studentName(num)
Next
End Sub
Private Sub Start_Click()
addName
End Sub

```

The 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 he click on the start button

LISTS : ONE DIMENSIONAL ARRAYS:

Suppose you need data for a 12-month period. You could try to write something like this:

```

For I = 1 to 12

MonthI = InputBoxt(Data for MonthI)

Next I

```

If this kind of loop were possible, then you would be all set. The information the user entered would easily be available in the various MonthI variables.

In Visual Basic, MonthI is a perfectly good variable name—**for** a single variable. Visual Basic cannot separate the I **from** the Month in **order to** make **the** 12 variables **that** you want **in** this example code.

The idea of a one-dimensional array (list) is to give you a systematic way to name groups of related variables that are part of a list. To Visual Basic, a one-dimensional Array is Just a collection of variables, each one of which is identified by two things:

- **The name of the array**

- The position of the **item on the list**.

FIXED VERSUS DYNAMIC ARRAYS

Fixed arrays are memory allocation stays the same for the whole time the program runs.

Dynamic arrays we can change size on the fly while the program is running.

To create a local array, use Private statement in a procedure

```
Dim x (10) as Integer
```

- **To create a module-level array use Private statement in declaration section of module.**
Private x (10) as Integer
- To create public array, use Public statement in the declaration section of a form
Public x (10) as Integer
- In fixed-sized arrays upper bound of the array should be given compulsorily.
- Upper bound is the upper limit for the size of the array.
- Lower bound of the array is given explicitly using the "**To**" keyword.

Example:

```
Dim x (1 To 10) as Integer
```

- The Ubound function returns the upper bound of an array.
- The Lbound function returns the lower bound of an array.

Example

```
Dim a (30)
```

```
x=Lbound (a)
```

Debug. Print x 'this will display 0 in debug window

How do change memory space for a dynamic array:

The ReDim statement use to the, change the memory space for inside a function or procedure.

Example:

```
Private name() As String Private Sub  
NameofprocedureQ  
  
    Dim Number As Integer  
  
    Number = CInt(Input Box("How many items?"))  
  
    ReDim name(Number) As String  
  
End Sub
```

PRESERVING INFORMATION IN A DYNAMIC ARRAY:

- When ReDim statement is used the previous array and its contents are destroyed.VB resets the values to empty value (for variant array), to Zero (for numeric arrays), to Zero-length string (for string arrays), or to nothing (for arrays of objects).
- To expand the size of array without destroying the data's that are entered already
- "Preserve" keyword is used.
- **The** statement is

```
ReDim Preservesaleval (12, 9)
```

- In case of single dimensional dynamic array we can enlarge the size of array by one clement without losing the values of existing elements using Ubound function.

```
ReDim Preserve x (Ubound (x) +1)
```

LOCAL ARRAYS:

To setup up a local dynamic array, use the appropriate ReDim statement in the function or procedure without first declaring the array in the Declarations section of the form.

As with a local fixed array, no other function or procedure can see the information stored in a local array. In particular, VB allocates space for that array *only* while the function or

procedure is active and reclaims the space as soon as the function or procedure ends. Here's an example using a procedure.

Example:-

```
Private SubProcedureNameO

    Dim DynamicLocalArrayf) as String, Temp As Integer

    Temp = CInt(InputBox ("How many items?"))

    ReDimDynamicLocalArray(Tempi    As
String )

    End Sub
```

Static Arrays:

As with ordinary static variables, it is occasionally useful to have an array whose contents remain the same no matter how often the function or procedure is used. By analogy with static variables, this kind of array is usually called a *static* array. To set up a static array inside a function or procedure, you use the Static keyword:

Example:-

```
Private FunctionFunctionNameO As String

    Static Errand(13) As String

    End Sub
```

One-Dimensional Arrays with Index Ranges:

Visual Basic has a command that eliminates the **zeroth entry in all** one-dimensional arrays dimensioned in a module or form. It is the Option Base 1 statement. This statement is used in the Declarations section of a form **or** module, **and it** affects **all** one-dimensional arrays in the module. All **new one-dimensional arrays** dimensioned **in** that form or module will **begin with** item 1.

```
Dim Sales$, I As Integers

    Static SalesInYear(I 7) As Single

    For I =0 TO 17
```

```
Sales$=Input:Box( "Enter the sales in year" + Str$(1980 +1))
```

```
SalesInyear(I) = Cur(Sales$)
```

```
Next I
```

However, this program requires 18 additions (one **for** each pass through the loop) and is more complicated to code. This situation, where you want to go **from the** beginning of a range of indexes to the end of the range is so common **that** Visual **Basic** enhanced the original **BASIC** language by allowing **subscript ranges**,

Instead of writing

```
DimSalesInYear(17)
```

you can now write

```
DimSalesInYear(1980 To 1997)
```

Ex:- _ . ;

```
Static salesinyear( 1980 to 1997) As single
```

```
Dim I As integer,sales$
```

```
For I =1980 to 1997
```

```
Sales$-InputBoxC'enter the Sales in year"+Str$(I))SalesInYear(i) = VCCur (sales$)
```

```
Next I
```

Assigning Arrays

One of the more useful features added to **VB6** is **the** ability to make **array** assignments. Suppose you **have a dynamic** array called MyErrands and another array called Your Errands that hold the **same type of information**. (**For example**, both are string **arrays**.) **Then the line**

```
MyErrands = YourErrands
```

will automatically change the size of the dynamic array named MyErrands to be the size of the array YourErrands and copy all the information from YourErrands into the corresponding slots in the MyErrands array.

For example, in this code

```
ReDim M(1 To 10) As String , :  
Dim foo(3 To 37) As String  
Print "Lower bound of M is "&LBound(M) ,  
Print "Upper bound of M is " &UBound(M)  
Print "Assigning the foo array to the M array" ,  
M == foo  
Print "Lower bound of M is now " &LBound(M) .  
Print "Upper bound of M is now "&UBound(M) .
```

Output:

```
Lower bound of M is 1  
Upper bound of M is 10  
Assigning the foo array .to the M array  
Lower bound of M is now 3  
Upper bound of M is now 37
```

The Array Function

The array function are used to store an array in a variant. If you store an array **in a** variant, use the ordinary index to get at it. It is a little less than elegant **to** store arrays **in** variants, but this technique does get around the limitation that only allows dynamic arrays on the left side of an assignment statement. For example, this technique gives **you a quick** way to swap the contents **of two** non dynamic arrays, as shown **here**:

Dim I As Long

```
Dim A(1 To 20000) As Long
Dim B(1 To 20000) As Long
For I = 1 To 20000
```

```
    A(I)=I
```

```
    B(I)=2*I
```

```
Next I
```

Dim Array1 As Variant, Array2 As Variant, Temp As Variant

Array1 = A()

Array2 = B()

Temp = Array1

Array1 = Array2

Array2 = Temp

Print "The third entry in Array1 is now" &Array1(3)

Output:

The third entry in Array1 is now 6

ARRAYS WITH MORE THAN ONE DIMENSION

Arrays that have more than one dimension are called "multidimensional arrays".

```
Static MultTable(1 To 12, 1 To 12) As Integer
```

```
Dim I As Integer, J As Integer
```

```
For I = 1 To 12
```

```
    For J = 1 To 12
```

```
        MultTable(I,J)=I*J
```

NextJ

NextI

To compute the number of items in a multidimensional array, multiply the number of entries. The dimension statement for the two-dimensional array that \ used here sets aside 144 elements.

The convention is to refer to the first entry as giving the number of rows **and the** second as giving the number of columns.

Visual Basic allows you up to 60 dimensions with the Dim statement and 8 with the ReDim statement. A statement like

```
Dim LargeArray%(2,2,2,2,2,2,2,2)
```

would set aside either $2^8 = 256$ or $3^8 = 6,561$ entries (depending on whether an Option Base 1 statement has been processed). But you almost never see more than four dimensions in a program, and even a three-dimensional array is uncommon. We use index ranges in multidimensional arrays as well. For example, the following

would give you a sales table for the months in the years 1990 to 1997:

```
Dim SalesTab(1 To 12, 1990 To 1997)
```

Finally, note that you can use **ReDim** for multidimensional arrays in exactly **the** same way as **with one-dimensional** arrays.

For example:

```
ReDim LargeArray%(2,2,2,2,2,2,2,2)
```

Since you can have index ranges in multidimensional arrays, you obviously need versions of LBound and UBound. The commands

```
LBound(NameOfArray, I)
```

```
UBound(NameOfArray, I)
```

Give the lower and upper bounds for the 1'th dimension of the array. (For a one-dimensional array the I is optional, as you have already seen.) Therefore,

```
Dim Test%(1 To 5,6 To 10,7  
To 12) Print LBound(Test%, 2)
```

gives a 6 and

```
Print UBound(Test%, 3)
```

gives a 12.

Using Lists and Arrays with Functions and Procedures

Visual Basic has an extraordinary facility for using lists and arrays in procedures and Functions, Unlike many languages, it's easy to send any size list or array to a procedure. Arrays are *always* passed by reference. This means any changes you make to the array or **to** the entries in the array will persist after VB leaves **the function** or procedure.

To send an array parameter to a procedure or function, just use the name of the array followed by opening and closing parentheses in the list of parameters.

For example,

The Array Function

The array function are used **to** store **an array** in a variant. **If you** store an array **in** a variant, use the ordinary index to get at it. It is a little less than elegant to store arrays in variants, but this technique does get around the limitation that only allows dynamic arrays **on** the left side of an assignment statement. **For** example, this technique gives you a **quick way to swap the contents of two non dynamic arrays**, as shown here:

```
Dim I As Long
```

```
Dim A(1 To 20000) As Long Him B.(1  
To 20000) As Long For I = 1 To  
20000
```

```
    A(I)-I
```

```
    B(I)=2*I Next I
```

```
Dim Array1 As Variant, Array2 As Variant, Temp As Variant
Array1 = AO Array2 = BQ Temp = Array1
Array1 == Array2 Array2 = Temp
Print "The third entry in Array1 is now" & Array1(3)
```

Output:

The third entry in Array1 is now 6

ARRAYS WITH MORE THAN ONE DIMENSION

Arrays that have more than one dimension are called "multidimensional arrays".

```
Static MultTable(1 To 12, 1 To 12) As Integer
Dim I As
```

```
Integer, J As Integer
```

```
For I=1 To 12 For J=1 To 12
```

```
    MultTable(I,J)=I*J
```

```
Next J . Next I
```

To compute the number of items in a multidimensional array, multiply the number of entries. The dimension statement for the two-dimensional array that I used here sets aside 144 elements.

The convention is to refer to the first entry as giving the number of rows and **the second** as giving **the number** of columns.

Visual Basic allows you up to 60 dimensions with the Dim statement and 8 with the ReDim statement. A statement like

```
Dim LargeArray%(2,2,2,2,2,2,2,2)
```

would set aside either $2^5 = 256$ or $3^8 = 6,561$ entries (depending on whether an Option Base 1 statement has been processed). But you almost never see more than four dimensions in a program, and even a three-dimensional array is uncommon. We use index ranges in multidimensional arrays as well. For example, the following

would give you a sales table for the months in the years 1990 to 1997:

```
Dim SalesTable(1 To 12, 1990 To 1997)
```

Finally, note that you can use ReDim for multidimensional arrays in exactly the same way as with one-dimensional arrays. For example:

ReDimLargeArray%(2,2,2,2,2,2,2) Since you can have index ranges in multidimensional arrays, you obviously need versions of LBound and UBound.

The commands

LKound(NameOfArray, J)

UKound(NameOfArray, I)

give the lower and upper bounds for the I'th dimension of the array. (For a one-dimensional array the I is optional, as you have already seen.) Therefore,

```
Dim Test%(1 To 5,6 To 10,7 To 12) Print
```

LBound(Test%,2) gives a 6 and

```
PrintUBound(Test%,3)
```

gives a **12**.

Using Lists and Arrays with Functions and Procedures

Visual **Basic** has an extraordinary facility for using lists and arrays in procedures and functions. Unlike many languages, it's easy to send any size list or array to a procedure. Arrays are *always* passed by reference. This means any changes you make **to** (he array or to the entries **in** the array will persist after **VB** leaves the function **or** procedure.

To send an array parameter to a procedure or function, just use the name **of the** array followed by opening and closing parentheses in the list of parameters.

For example,

Assume that List# is a one-dimensional array of double-precision variables, i Array\$ is a two-dimensional string array, and BigArray% is a three-dimensional array of integers. Then,

```
Private Sub Example(List# (), Array$(), BigArray%(), X%)
```

would allow this Example procedure **to** use (and change) a list of **double-precision** variables, an array of strings, a three-dimensional array **of** integers, and a final integer variable. Note that,

just as with variables, list and array parameters are placeholders; **they** have no independent existence. To call the procedure, you might have a fragment like this:

```
Dim popChange#(50), CityState$(3.10). TotalPop%/o(2,2,2)
```

Now,

```
Example PopChange#(), CityState$,TotalPop%(), XI# .
```

would call the Example procedure by sending **it** the current location (passed by **reference**) of the three arrays and the integer variable. And just as before, since **the** compiler **known** where the variable ,list. or array is located .It can change the contents steps for **write the** function procedure,

- Function FmdMaximum(ListQ)

Start at the top of the list

If an entry is bigger than the current max "swap it"

- Until you finish the **list**
- **Set the** value of the function to the final "Max"

Ex:

```
Function FindMax(A() As Single)As Single
```

```
Dim Start As Integer, Finish As Integer, I As Integer
```

```
Dim Max As Single
```

```
Start=LBOUND(A)
```

```
Finish-UBOUND(A)
```

```
Max=A(Start)
```

```
For I=Start To Finish
```

```
IfA(I)> Max Then Max=A(I)
```

```
Next I
```

```
FindMax=Max
```

```
End Function,
```

The New Array-Based String

Handling Functions

The first of the new array-based String functions added is the Join function. This takes an array of strings and makes a new string out of all entries. It used to join together the individual strings and it even works faster.

Example:

```
Dim MyNieces(1 to 3)
MyNieces(1)="Shara"
MyNieces(2) = "Rebecca"
MyNieces(3)="Alysa"
MsgBox Join(MyNieces) & "are the joy of my life."
```

The Split Function

Split function is used to split the text in more than one part.

Example:

```
"SachinRomesh Tendulkar"
```

The split function can take this string and return an array of three strings:

```
First array entry = "Sachin"
```

```
Second array entry = "Romesh"
```

```
Third array entry = "Tendulkar"
```

Example program:

```

Private Sub Form_Load

    Dim Teststr As String

    Dim A() As String

    Dim I As Integer

    Teststr = "SachinRamesh Tendulkar"

    A = Split(Teststr)

    For I = LBound(A) to UBound(A)

        MsgBox A(I)

    Next End

End Sub

```

The filter Function:

The filter function takes an array of strings and returns a new array by including only those entries that satisfy the filter.

```
FilteredNewsGroups = Filter (NamesOfNewsGroups."alt.", False)
```

Now the FilteredNewsGroups array contains exactly what **you want**.

The full syntax for filter function:

```
Filter (inputstrings, value[, include[, compare]])
```

Records (User defined Types)

Record is a collection of mixed variable. It usually mixes different kinds of numbers and strings.

Declaration;

```
Type SamInfo
```

Name as String

Salary as **Long**

Empno as Integer

End Type

This defines the type, Now to make a single variable of "type" SamInfo,

Private Youname As SamInfo

Now you use a dot to isolate the parts of ihis record:

YoLirname.Name = "Raja"

Yourname.Satary- 10000

Youname. Empno == 101

We can need more then one record created as,

Dim CompanyRecord (1 to 75) as SamInfo.

..... **III – UNIT COMPLETED**

UNIT – IV

List Boxes and Combo boxes - Do/Loop - For/Next Loop - Using MsgBox Function - Using String Function - Arrays: Control Arrays - Single Dimension Array - For Each/Next Statements - User defined data types - Multidimensional Arrays.

ARRAYS

Array is a collection of similar data items in same data type. **To** distinguish among the items store in an array, it **use** special **kind** of parameter called as index.

Dimension of an Array

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 with more complex problems. Let me illustrate the the arrays with tables.

One dimensional Array

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

Two Dimensional Array

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

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

Declaring Arrays

We can 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 a one dimensional array is as follow:

Dim arrayName(subs) as dataType

where subs indicates the last subscript in the array.

Example

Dim CusName(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 CusName(1) to CusName(10). Otherwise, there will be 11 elements in the array starting from CusName(0) through to CusName(10)

CusName(1)	CusName(2)	CusName(3)	CusName(4)	CusName(5)
CusName(6)	CusName(7)	CusName(8)	CusName(9)	CusName(10)

Example

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)

The general format to declare a two dimensional array is as follow:

```
Dim ArrayName(Sub1,Sub2) as dataType
```

Example

Dim StudentName(10,10) will declare a 10x10 table make up of 100 students' Names, starting with StudentName(1,1) and end with StudentName(10,10).

Sample Programs

(i) Dim studentName(10) As String

```
Dim num As Integer
```

```
Private Sub addName()
```

```
For num = 1 To 10
```

```
studentName(num) = InputBox("Enter the student name", "Enter Name", "", 1500, 4500)
```

```
If studentName(num) <> "" Then
```

```
Form1.Print studentName(num)
```

```
Else
```

```
End
```

```
End If
```

```
Next
```

```
End Sub
```

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

ii) Dim studentName(10) As String

```
Dim num As Integer
```

```
Private Sub addName( )
```

```
For num = 1 To 10
```



```

studentName(num) = InputBox("Enter the student name")
List1.AddItem studentName(num)
Next
End Sub
Private Sub Start_Click()
addName
End Sub

```

The 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 he click on the start button

LISTS : ONE DIMENSIONAL ARRAYS:

Suppose you need data for a 12-month period. You could try to write something like this:

```

For I = 1 to 12
MonthI = InputBox("Data for MonthI")
Next I

```

If this kind of loop were possible, then you would be all set. The information the user entered would easily be available in the various MonthI variables.

In Visual Basic, MonthI is a perfectly good variable name—**for** a single variable. Visual Basic cannot separate the I **from** the Month in **order to** make **the** 12 variables **that** you want **in** this example code.

The idea of a one-dimensional array (list) is to give you a systematic way to name groups of related variables that are part of a list. To Visual Basic, a one-dimensional Array is Just a collection of variables, each one of which is identified by two things:

- **The name** of the **array**
- The position of the **item on the list.**

FIXED VERSUS DYNAMIC ARRAYS

Fixed arrays are memory allocation stays the same for the whole time the program runs.

Dynamic arrays we can change size on the fly while the program is running.

To create a local array, use Private statement in a procedure

```
Dim x (10) as Integer
```

- **To create a module-level array use Private statement in declaration section of module.**
Private x (10) as Integer
- To create public array, use Public statement in the declaration section of a form
Public x (10) as Integer
- In fixed-sized arrays upper bound of the array should be given compulsorily.
- Upper bound is the upper limit for the size of the array.
- Lower bound of the array is given explicitly using the "**To**" keyword.

Example:

```
Dim x (1 To 10) as Integer .
```

- The Ubound function returns the upper bound of an array.
- The Lbound function returns the lower bound of an array.

Example

```
Dim a (30)
```

```
x=Lbound (a)
```

Debug. Print x 'this will display 0 in debug window

How do change memory space for a dynamic array:

The ReDim statement use to the, change the memory space for inside a function or procedure.

Example:

```
Private name() As String Private Sub  
NameofprocedureQ  
  
    Dim Number As Integer  
  
    Number = CInt(Input Box("How many items?"))  
  
    ReDim name(Number) As String  
  
End Sub
```

PRESERVING INFORMATION IN A DYNAMIC ARRAY:

- When ReDim statement is used the previous array and its contents are destroyed.VB resets the values to empty value (for variant array), to Zero (for numeric arrays), to Zero-length string (for string arrays), or to nothing (for arrays of objects).
- To expand the size of array without destroying the data's that are entered already
- "Preserve" keyword is used.
- **The** statement is

```
ReDim Preservesaleval (12, 9)
```

- In case of single dimensional dynamic array we can enlarge the size of array by one clement without losing the values of existing elements using Ubound function.

```
ReDim Preserve x (Ubound (x) +1)
```

LOCAL ARRAYS:

To setup up a local dynamic array, use the appropriate ReDim statement in the function or procedure without first declaring the array in the Declarations section of the form.

As with a local fixed array, no other function or procedure can see the information stored in a local array. In particular, VB allocates space for that array *only* while the function or

procedure is active and reclaims the space as soon as the function or procedure ends. Here's an example using a procedure.

Example:-

```
Private SubProcedureNameO

    Dim DynamicLocalArrayf) as String, Temp As Integer

    Temp = CInt(InputBox ("How many items?"))

    ReDimDynamicLocalArray(Tempi    As
String )

    End Sub
```

Static Arrays:

As with ordinary static variables, it is occasionally useful to have an array whose contents remain the same no matter how often the function or procedure is used. By analogy with static variables, this kind of array is usually called a *static* array. To set up a static array inside a function or procedure, you use the Static keyword:

Example:-

```
Private FunctionFunctionNameO As String

    Static Errand(13) As String

    End Sub
```

One-Dimensional Arrays with Index Ranges:

Visual Basic has a command that eliminates the **zeroth entry in all** one-dimensional arrays dimensioned in a module or form It is the Option Base 1 statement. This statement is used in the Declarations section of a form **or** module, **and it** affects **all** one-dimensional arrays in the module. All **new one-dimensional arrays** dimensioned **in** that form or module will **begin with** item 1.

```
Dim Sales$, I As Integers

    Static SalesInYear(I 7) As Single

    For I =0 TO 17
```

```
Sales$=Input:Box( "Enter the sales in year" + Str$(1980 +1))
```

```
SalesInyear(I) = Cur(Sales$)
```

```
Next I
```

However, this program requires 18 additions (one **for** each pass through the loop) and is more complicated to code. This situation, where you want to go **from the** beginning of a range of indexes to the end of the range is so common **that** Visual **Basic** enhanced the original **BASIC** language by allowing **subscript ranges**,

Instead of writing

```
DimSalesInYear(17)
```

you can now write

```
DimSalesInYear(1980 To 1997)
```

Ex:- _ . ;

```
Static salesinyear( 1980 to 1997) As single
```

```
Dim I As integer,sales$
```

```
For I =1980 to 1997
```

```
Sales$-InputBoxC'enter the Sales in year"+Str$(I))SalesInYear(i) = VCCur (sales$)
```

```
Next I
```

Assigning Arrays

One of the more useful features added to **VB6** is **the** ability to make **array** assignments. Suppose you **have a dynamic** array called MyErrands and another array called Your Errands that hold the **same type of information**. (**For example**, both are string **arrays**.) **Then the line**

```
MyErrands = YourErrands
```

will automatically change the size of the dynamic array named MyErrands to be the size of the array YourErrands and copy all the information from YourErrands into the corresponding slots in the MyErrands array.

For example, in this code

```
ReDim M(1 To 10) As String
Dim foo(3 To 37) As String
Print "Lower bound of M is "&LBound(M)
Print "Upper bound of M is " &UBound(M)
Print "Assigning the foo array to the M array"
M == foo
Print "Lower bound of M is now " &LBound(M)
Print "Upper bound of M is now " &UBound(M)
```

Output:

```
Lower bound of M is 1
Upper bound of M is 10
Assigning the foo array to the M array
Lower bound of M is now 3
Upper bound of M is now 37
```

The Array Function

The array function are used to store an array in a variant. If you store an array **in a** variant, use the ordinary index to get at it. It is a little less than elegant **to** store arrays **in** variants, but this technique does get around the limitation that only allows dynamic arrays on the left side of an assignment statement. For example, this technique gives **you a quick** way to swap the contents **of two** non dynamic arrays, as shown **here**:

Dim I As Long

```
Dim A(1 To 20000) As Long
Dim B(1 To 20000) As Long
For I = 1 To 20000
```

```
    A(I)=I
```

```
    B(I)=2*I
```

```
Next I
```

Dim Array1 As Variant, Array2 As Variant, Temp As Variant

Array1 = A()

Array2 = B()

Temp = Array1

Array1 = Array2

Array2 = Temp

Print "The third entry in Array1 is now" &Array1(3)

Output:

The third entry in Array1 is now 6

ARRAYS WITH MORE THAN ONE DIMENSION

Arrays that have more than one dimension are called "multidimensional arrays".

```
Static MultTable(1 To 12, 1 To 12) As Integer
```

```
Dim I As Integer, J As Integer
```

```
For I = 1 To 12
```

```
    For J = 1 To 12
```

```
        MultTable(I,J)=I*J
```

NextJ

NextI

To compute the number of items in a multidimensional array, multiply the number of entries. The dimension statement for the two-dimensional array that \ used here sets aside 144 elements.

The convention is to refer to the first entry as giving the number of rows **and the** second as giving the number of columns.

Visual Basic allows you up to 60 dimensions with the Dim statement and 8 with the ReDim statement. A statement like

```
Dim LargeArray%(2,2,2,2,2,2,2,2)
```

would set aside either $2^8 = 256$ or $3^8 = 6,561$ entries (depending on whether an Option Base 1 statement has been processed). But you almost never see more than four dimensions in a program, and even a three-dimensional array is uncommon. We use index ranges in multidimensional arrays as well. For example, the following

would give you a sales table for the months in the years 1990 to 1997:

```
Dim SalesTab(1 To 12, 1990 To 1997)
```

Finally, note that you can use **ReDim** for multidimensional arrays in exactly **the** same way as **with one-dimensional** arrays.

For example:

```
ReDim LargeArray%(2,2,2,2,2,2,2,2)
```

Since you can have index ranges in multidimensional arrays, you obviously need versions of LBound and UBound. The commands

```
LBound(NameOfArray, I)
```

```
UBound(NameOfArray, I)
```

Give the lower and upper bounds for the 1'th dimension of the array. (For a one-dimensional array the I is optional, as you have already seen.) Therefore,


```
Dim Test%(1 To 5,6 To 10,7  
To 12) Print LBound(Test%, 2)
```

gives a 6 and

```
Print UBound(Test%, 3)
```

gives a 12.

Using Lists and Arrays with Functions and Procedures

Visual Basic has an extraordinary facility for using lists and arrays in procedures and Functions, Unlike many languages, it's easy to send any size list or array to a procedure. Arrays are *always* passed by reference. This means any changes you make to the array or **to** the entries in the array will persist after VB leaves **the function** or procedure.

To send an array parameter to a procedure or function, just use the name of the array followed by opening and closing parentheses in the list of parameters.

For example,

The Array Function

The array function are used **to** store **an array** in a variant. **If you** store an array **in** a variant, use the ordinary index to get at it. It is a little less than elegant to store arrays in variants, but this technique does get around the limitation that only allows dynamic arrays **on** the left side of an assignment statement. **For** example, this technique gives you a **quick way to swap the contents of two non dynamic arrays**, as shown here:

```
Dim I As Long
```

```
Dim A(I To 20000) As Long Him B.(I  
To 20000) As Long For I = 1 To  
20000
```

```
A(I)-I
```

```
B(I)=2*I Next I
```

```
Dim Array1 As Variant, Array2 As Variant, Temp As Variant
Array1 = AO Array2 = BQ Temp = Array1
Array1 == Array2 Array2 = Temp
Print "The third entry in Array1 is now" & Array1(3)
```

Output:

The third entry in Array1 is now 6

ARRAYS WITH MORE THAN ONE DIMENSION

Arrays that have more than one dimension are called "multidimensional arrays".

```
Static MultTable(1 To 12, 1 To 12) As Integer
Dim I As
```

```
Integer, J As Integer
```

```
For I=1 To 12 For J=1 To 12
```

```
    MultTable(I,J)=I*J
```

```
Next J . Next I
```

To compute the number of items in a multidimensional array, multiply the number of entries. The dimension statement for the two-dimensional array that I used here sets aside 144 elements.

The convention is to refer to the first entry as giving the number of rows and **the second** as giving **the number** of columns.

Visual Basic allows you up to 60 dimensions with the Dim statement and 8 with the ReDim statement. A statement like

```
Dim LargeArray%(2,2,2,2,2,2,2,2)
```

would set aside either $2^5 = 256$ or $3^8 = 6,561$ entries (depending on whether an Option Base 1 statement has been processed). But you almost never see more than four dimensions in a program, and even a three-dimensional array is uncommon. We use index ranges in multidimensional arrays as well. For example, the following

would give you a sales table for the months in the years 1990 to 1997:

```
Dim SalesTable(1 To 12, 1990 To 1997)
```

Finally, note that you can use ReDim for multidimensional arrays in exactly the same way as with one-dimensional arrays. For example:

ReDimLargeArray%(2,2,2,2,2,2,2) Since you can have index ranges in multidimensional arrays, you obviously need versions of LBound and UBound.

The commands

LKound(NameOfArray, J)

UKound(NameOfArray, I)

give the lower and upper bounds for the I'th dimension of the array. (For a one-dimensional array the I is optional, as you have already seen.) Therefore,

```
Dim Test%(1 To 5,6 To 10,7 To 12) Print
```

LBound(Test%,2) gives a 6 and

```
PrintUBound(Test%,3)
```

gives a **12**.

Using Lists and Arrays with Functions and Procedures

Visual **Basic** has an extraordinary facility for using lists and arrays in procedures and functions. Unlike many languages, it's easy to send any size list or array to a procedure. Arrays are *always* passed by reference. This means any changes you make **to** (he array or to the entries **in** the array will persist after **VB** leaves the function **or** procedure.

To send an array parameter to a procedure or function, just use the name **of the** array followed by opening and closing parentheses in the list of parameters.

For example,

Assume that List# is a one-dimensional array of double-precision variables, i Array\$ is a two-dimensional string array, and BigArray% is a three-dimensional array of integers. Then,

```
Private Sub Example(List# (), Array$(), BigArray%(), X%)
```

would allow this Example procedure **to** use (and change) a list of **double-precision** variables, an array of strings, a three-dimensional array **of** integers, and a final integer variable. Note that,

just as with variables, list and array parameters are placeholders; **they** have no independent existence. To call the procedure, you might have a fragment like this:

```
Dim popChange#(50), CityState$(3.10). TotalPop%/o(2,2,2)
```

Now,

```
Example PopChange#(), CityState$,TotalPop%(), XI# .
```

would call the Example procedure by sending **it** the current location (passed by **reference**) of the three arrays and the integer variable. And just as before, since **the** compiler **known** where the variable ,list. or array is located .It can change the contents steps for **write the** function procedure,

- Function FmdMaximum(ListQ)

Start at the top of the list

If an entry is bigger than the current max "swap it"

- Until you finish the **list**
- **Set the** value of the function to the final "Max"

Ex:

```
Function FindMax(A() As Single)As Single
```

```
Dim Start As Integer, Finish As Integer, I As Integer
```

```
Dim Max As Single
```

```
Start=LBOUND(A)
```

```
Finish-UBOUND(A)
```

```
Max=A(Start)
```

```
For I=Start To Finish
```

```
IfA(I)> Max Then Max=A(I)
```

```
Next I
```

FindMax=Max

End Function,

The New Array-Based String

Handling Functions

The first of the new array-based String functions added is the Join function. This takes an array of strings and makes a new string out of all entries. It used to join together the individual strings and it even works faster.

Example:

```
Dim MyNieces(1 to 3)
MyNieces(1)="Shara"
MyNieces(2) = "Rebecca"
MyNieces(3)="Alysa"
MsgBox Join(MyNieces) & "are the joy of my life."
```

The Split Function

Split function is used to split the text in more than one part.

Example:

```
"SachinRomesh Tendulkar"
```

The split function can take this string and return an array of three strings:

First array entry = "Sachin"

Second array entry = "Romesh"

Third array entry = "Tendulkar"

Example program:

```
Private Sub Form_Load
```

```
    Dim Teststr As String
```

```
    Dim A() As String
```

```
    Dim I As Integer
```

```
    Teststr = "SachinRamesh Tendulkar"
```

```
    A = Split(Teststr)
```

```
    For I = LBound(A) to UBound(A)
```

```
        MsgBox A(I)
```

```
    Next End
```

```
End Sub
```

The filter Function:

The filter function takes an array of strings and returns a new array by including only those entries that satisfy the filter.

```
FilteredNewsGroups = Filter (NamesOfNewsGroups."alt.", False)
```

Now the FilteredNewsGroups array contains exactly what **you want**.

The full syntax for filter function:

Filter (inputstrings, value[, include[, compare]])

Records (User defined Types)

Record is a collection **of** mixed variable. **It** usually mixes different kinds **of** numbers and strings.

Declaration;

Type SamInfo

Name as String

Salary as **Long**

Empno as Integer

End Type

This defines the type, Now to make a single variable of "type" SamInfo,

Private Youname As SamInfo

Now you use a dot to isolate the parts of this record:

YoLirname.Name = "Raja"

Yourname.Salary- 10000

Youname. Empno == 101

We can need more than one record created as,

Dim CompanyRecord (1 to 75) as SamInfo.

..... **IV – UNIT COMPLETED**

UNIT - V

Accessing Database Files: Visual basic and Database Files - Using Data Control - Viewing a Database File - Navigating the Database in Code - Using List Boxes and Combo boxes as Data-Bound Controls.

FUNDAMENTALS OF GRAPHICS:

Windows tells the display adapter how to display the image. VB graphics statements depends on the driver programs windows uses to control the screen and printer. Windows has to do a lot to manage a graphics environment, and this forces trade-offs. For example, unless we set AutoRedraw property to True so that VB saves a copy of the object in memory, we will have to manage the redrawing of graphics yourself.

The effects of setting the AutoRedraw property to True are slightly different for forms and picture boxes.

- For resizable form, VB saves a copy of the entire form. Thus when we enlarge the form, no graphics information is lost. This option requires by far the most memory, but id your graphics do not currently fit on a form, but will when the form is enlarged, choose this option.
- For a picture box, VB saves an image only as large as the current size of the box. Nothing new will appear even if the box is enlarged later.

Features of the AutoRedrawProperty :

We can change AutoRedraw to False while a program is running. Then we clear the object by using the Cls method. Whatever we drew before we changed the AutoRedraw property will remain, but everything that was drawn after the switch will disappear.

```
Private sub Form_load()  
    AutoRedraw = True  
    Print " welcome to III – CS "  
End sub
```

Now , for the click procedure, add

```
Private sub Form_Click()  
    AutoRedraw = False  
    Cls  
    Print : print : print  
End sub
```

Finally, the Double_click procedure is simply :


```

Private sub Form_Dblclick( )
    Cls
End sub

```

The Refresh method :

We can use the Refresh method when working with graphics. This method applies to both forms and controls. What calling this method does is force and immediate refresh of the form or control. Whenever VB processes an object. Refresh statement, it will redraw the object immediately and generate the Paint event, if the object supports this feature.

Saving Pictures :

VB makes it easy to save the pictures we have drawn to a form or picture box.

syntax:

SavePictureobjectName.Image, Felename

The operating system uses the Image property to identify the picture in the form or picture box. If we leave off ObjectName, then, as usual, VB uses the current form

Syntax:

SavePicture Image, Filename

SCREEN SCALES :

The default scale for forms and picture boxes uses twips – that rather strange scale that is “1/20 of printers point “. While this is a great scale for our printouts, it can be less than ideal for displays. The default size for a form on an ordinary 14-inch VGA monitor running 640 X 480 is roughly 7,485 twips long by 4,425 twips wide. Since a twip is 1/1440 of an inch when printed, this default form size is roughly 5 inches by 3 inches if we use the PrintForm method on a screen of 640 X 480 resolution.

Coordinates	location
(0,0)	Top left corner
(7485,0)	Top right corner
(0,4425)	Bottom left corner
(7485,4425)	Bottom right corner
(3742,2212)	Roughly the center

Other Screen Scales :

Scale Mode	Units
1	Twips
2	Points
3	Pixels
4	Charaters

5	Inches
6	Millimeters
7	centimeters

Once we set the ScaleMode property to one of these new values, we can read off the size of the drawing area, which is the area inside the form or the picture box or the printable area on a piece of paper in our printer.

Custom Scales :

The screen is normally numbered with (0,0) as the top left corner. This is obviously inconvenient for drawing tables, charts, graphs, and other mathematical shapes. In most of these situations, we want the coordinates to decrease as we move from top to bottom and increase as we move from left to right.

The scale method sets up new coordinates for forms and picture boxes that we can use any of the graphics methods. For example,

Scale (-320,240) – (320, -240)

Sets up a new coordinate system with the coordinates of the top left corner being (-320,240) and the bottom right corner being (320, -240). After this method, the four corners are described in a clockwise order, starting from the top left.

(-320,240)
(320,240)
(320,-240)
(-320,-240)

Now 0,0 is roughly in the center of the screen.

Another way to set up Custom scales :

The Scale method is the simplest way to set up a custom scale, but there is one better way that occasionally may be useful. We can specify the coordinates of the top left corner and how VB should measure the vertical and horizontal scales. We do all this by using combinations of the ScaleLeft, ScaleTop, ScaleWidth and scaleHeight properties. For example,

Object .ScaleLeft = 100
Object .ScaleTop = 500
Object .ScaleHeight = 250
Object .ScaleWidth = 250

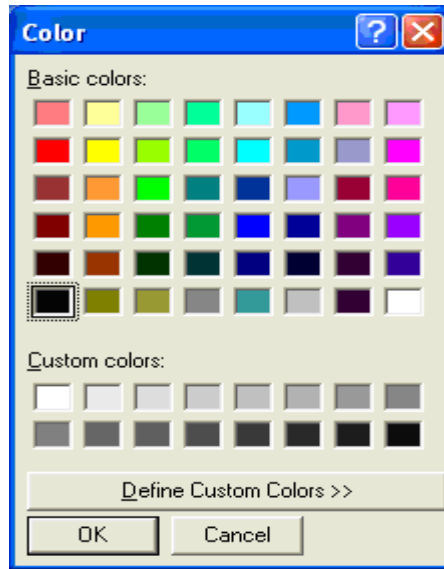
THE LINE AND SHAPE CONTROLS :

We can quickly display simple lines and shapes or print them on a printer with the Line and Shape controls.

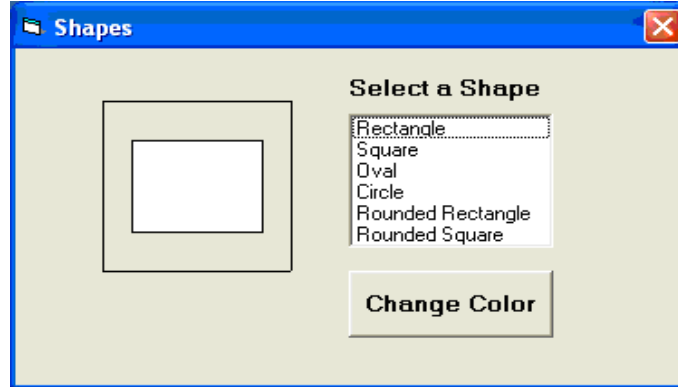
The Code

```
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
```

The color dialog box



The Interface



The Shape control :

The Shape control has 20 properties. Usually, we change them dynamically with code while the application is running. The most important properties for the Shape control at design time.

Shape

This determines the type of shape we get, there are six possible settings.

Setting of Shape property	Effect
---------------------------	--------

0	Rectangle (default)
1	Square
2	Oval
3	Circle
4	Rounded rectangle
5	Rounded square

BackColor:

It determines whether the background of the shape is transparent. The default value is 1, which gives an opaque border. BackColor fills the shape and obscures what is behind it. Set it to 0 if we want to see through the shape to what is behind it.

BorderWidth :

It determines the thickness of the line. It is measured in pixels and can range from 0 to 8192.

BorderStyle :

This is for shape controls has 6 possible settings. Having no border prevents the control from being visible, unless we modify the FillStyle and FillColor properties.

Setting of BorderStyle property	Effect
0	No border
1	Solid
2	Dashed line
3	Dotted line
4	Dash-dot line
5	Dash-dot-dot line

FillColor,

FillStyle :

FillColor determines the color used to fill the shape in the manner set by the FillStyle property. We can set the FillColor property in the same way as setting any color property, either directly via hexadecimal code or by using the color palette.

Setting of FillStyle property	Effect
0	Solid
1	Transparent
2	Horizontal line

3	Vertical line
4	Upward diagonal
5	Downward diagonal
6	Cross
7	Diagonal cross

The Line control :

The Line control has 15 properties. Usually, we change them dynamically with code while the application is running. The most important properties for the Line control at design time are the BorderWidth property and the BorderStyle property. BorderWidth determines the thickness of the line. It is measured in pixels and can range from 0 to 8192.

The most important properties at run time for the Line control are the X1, Y1, X2, and Y2 properties. These govern where the edges of the line appear. The X1 sets the horizontal position of the left end of the line. Y1 sets vertical position of the left-hand end point. The X2 and Y2 work similarly for the right end of the line.screen

GRAPHICS VIA CODE :

If we want to draw a few shapes on the screen, there is no use any of the graphical methods.

Colors :

First step is to decide what colors we want. If we don't specify a color, VB uses the foreground color of the object for all the graphics methods. The way is use RGB function is,

RGB (AmountOfRed, Amount Of Green, AmountOfBlue)

Where the amount of colors is an integer between 0 to 255. If we want to use color scheme from QuickBASIC, use the QBColor function.

QBColor(ColorCode)

Code	Color	Code	Color
0	Black	8	Gray
1	Clue	9	Light blue
2	Green	10	Light green

3	Cyan	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	brown	14	yellow
7	White	15	High-intensity white

Pixel Control :

We can change the scale of our screen by using pixels.

pSet (Col, Row) [, ColorCode]

we need to do replace the parameters with the values we want. The value of the first entry determines the column and the second determines row. For example

```
pSet (3722, 2212)
```

would return on the center pixel on a standard 14-inch VGA screen

LINES AND BOXES :

VB comes with a rich supply of graphics tools. Usually called graphics primitives, that allow to plot such geometric figures as lines, boxes, circles, ellipse and wedges with single statements.

```
Line (StartCol, StartRow) – (EndCol, EndRow), ColorCode
```

Which gives a line connection the two points with the given coordinates, using the color specified by Color Code.for example,

```
Private sub Form_click ( )
AutoReDraw = true
Line (ScaleWidth / 2,0) – (ScaleWidth/2, ScaleHeight + 1000)
Line (0, Scale Height) – (ScaeWidth, scaleHeight)
End sub
```

Last point referenced :

VB keeps track of where it stopped plotting. This location is usually called the last point referenced (LPR), and the values of the CurrentX and CurrentY variables store this information. If we are continuing a line from the last point referenced, VB allows us to omit the LPR in the Lile method.

For example :

```
Line – (160, 90)
```

Grid Graphics :

We want to draw a rocket ship, can read off the coordinates from the diagram, it is easy to write the following fragment.

```
Scale (0, 0) – (25, 20)
Line (6, 20) – (14, 20)
Line – (12, 18)
Line – (12, 9)
Line – (10, 6)
Line – (8, 9)
Line – (8, 18)
Line – (6, 20)
```

DrawWidth, DrawStyle :

When we draw on the printer or the screen by using the pSet or Line method, VB uses dots that are normally drawn one pixel wide. If we need to change the width of points or lines, use the DrawWidth property.

Object .Drawwidth = Size %

The theoretical maximum size for DrawWidth is 32, 767.

Setting of DrawStyle property	Effect
0	Solid
1	Dashed line
2	Dotted line
3	Dash-dot-dash-dot pattern
4	Dash-dot-dot pattern
5	Transparent
6	Inside solid

Boxes :

A modification of the Line method lets we draw rectangle. The statement

Line (FirstCol, FirstRow) – (secCol, SecRow), CCode, B)

Draws a rectangle in the given color code whose opposite corners are given by FirstCOL, FirstRow and SecCol, SecRow. For example,

```
Private sub Form_Load ( )
  Show
  Dim I as integer
  Scale (0, 0) – (639, 479)
  For I = 1 to 65 step 5
    Line (5 * i) – (639 – 5 * I, 479 – i), , b
  Next i
End sub
```


Filled Boxes :

We can arrange for the line method to give a filled box as well. We need to do is use BF rather than B, and we get a filled box.

```
Line (FirstCol, FirstRow) – (SecCol, SecRow), CCode, BF
```

Will yield a solid rectangle whose opposite corners are given by FirstCol, FirstRow and SecCol, SecRow. For example,

```
Dim I as integer
```

```
Scale (0, 0) – (639, 479)
```

```
For I = 1 to 65 step 5
```

```
Line (5 * i) – (639 – 5 * I, 479 – i), CCode , BF
```

```
Next i
```

FillStyle, FillColor :

Boxes are usually empty or solid but VB allows seven different patterns to fill boxes. To do this we need to change the Fillstyle property of the form or picture box.

Setting of FillStyle property	Effect
0	Solid
1	Empty
2	Horizontal line
3	Vertical line
4	Upward diagonal
5	Downward diagonal
6	Cross
7	Diagonal cross

Once we have changed the FillStyle property from its transparent default. We can use FillColor to set the color used for FillStyle.

```
Object .FillColor = ColorCode
```

CIRCLES, ELLIPSES, AND PIE CHARTS :

Circle in VB gives its center and radius. The following fragment draws a circle of radius 0.5 units starting at the center of the screen.

```
Scale (-1, 1) – (1, _1)
```

```
Circle (0, 0), .5
```

The last pointer referenced after a circle method is always the center of the circle. We can add a color code to the circle method. For example.

```
Circle (0, ), .5, CCode
```

Would draw a circle of radius 0.5 in the color code indicated here by the variable CCode. The following program shows the circle method, which produces the nested circles .

```
Private sub Form_load ()
Dim I as Single, CCode as Single
Windowstate = 2
Show
Scale (-1, 1) – (1, _1)
For I = 0.1 to 0.7 step 0.05
CCode = 16 *rnd
  Circle (0, 0 ), I, CCode
Next i
End
sub
```

Ellipses :

We can convert the circle drawing method to an Ellipse drawing command by adding one or more option.

```
Circle [step] (XCenter, YCenter), radius, , , , aspect
```

The four commas must be there, even if we are not using the color code and angle options that we saw earlier.

```
Scale (-1, 1) – (1, _1)
Circle (-.4, 0), .5, , -pi/4, -3*pi/4
Circle (.4, 0), .5 , pi/4, 3*pi/4
```

MOUSE EVENT PROCEDURES :

There are 3 mouse event procedures available.

Controls recognise a mouse event only when mouse pointer is inside the control. All mouse event procedures take the same form and use the same parameters.

ObjectName_MouseEvent (Button As Integer, Shift As Integer, X As Single, Y As Single)

The MouseUp and MouseDownEvents :

These procedures start up a new project, double_click to open the Code window and move to the MouseDown event procedure for the form.

```
Private sub Form_MouseDown(Button As Integer, Shift As Integer, X as Single, Y As Single)
    Circle (X,Y), 75
End sub
```

This simple event procedure uses the positioning information passed by X and Y. each time we click a mouse button, a small circle is centered exactly where we clicked – namely.

Using the Button Argument :

However we want to make some circles filled and some empty. One way to do this is to use added information given by the Button argument. The Button argument uses the lowest three bits of the value of the integer. They are

Name	Event that caused it
MouseDown	User clicks one of the mouse buttons
MouseUp	User releases a mouse button
MouseMove	User moves the mouse pointer to a control

Button	Constant	Value of button argument
Left	vbLeftButton	1
Right	vbRightButton	2
Middle	vbMiddleButton	4

VB tells us about one button for the MouseUp / MouseDown combination. We cannot detect if both the left and right buttons are down simultaneously.

Combining the Keyboard and the Mouse :

We can also detect if the user presses one of the SHIFT, CTRL, and ALT keys while simultaneously working with the mouse. The trick to detect this user action is to use the Shift argument in the MouseUp or MouseDown event procedure.

Action	Constant	Bit Set and Value
SHIFT key down	vbShiftMask	Bit 0 : value = 1
CTRL key down	vbCtrlMask	Bit 1 : value = 2
ALT key down	vbAltMask	Bit 2 : value = 4
SHIFT + CTRL keys down	vbShiftMask + vbCtrlMask	Bit 0 and 1 : value = 3
SHIFT + ALT keys down	vbShiftMask + vbAltMask	Bit 0 and 2: value = 5
CTRL + ALT keys down	vbCtrlMask + vbAltMask	Bit 1 and 2 : value = 6
SHIFT + CTRL + ALT keys down	vbShiftMask + vbCtrlMask + vbAltMask	Bit 0, 1 and 2 : value = 7

Making pop-up menus :

VB controls have their own pop-up menus. It is especially important when using a control for the first time to check whether it has a built in pop-up menu, add it to a blank form and right-click on the control while the program is running.

Here are the steps to add the pop-up capability.

- The pop-up menu will not show the top level entry in the menu.
- If we don't want to see the pop-up menu on the main menu bar, we must set the Visible property of the top-level menu entry to False.

Simply add code in the MouseDown event for the control's MouseDown event that follows.

```
Private sub TheControl_MouseDown (button As Integer, Shift As integer, A As Single, Y As Single)
    If Button = vbRightButton then
        Me.PopUpMenuMenuName
    End sub
```

The MouseMoveevent :

VB calls the MouseMove event procedure whenever the user moves the mouse. It works, start a new project and enter the following code.

```
Private sub Form_MouseMove (Button As integer, Shift As integer, X As Single, Y As Single)
    DrawWidth = 3
    pSet (X,Y)
End sub
```

DRAGGING AND DROPPING OPERATIONS :

To move a control as we are designing the interface in our VB project, we hold down a mouse button and then move the mouse pointer to where we want the control to end up. A gray outline of the control moves with the mouse pointer.

The MS windows documentation calls moving an object with the mouse button depressed **dragging** and calls the release of the mouse button **dropping**.

There are two types of dragging.

1. Manual Dragging
2. Automatic Dragging

For example, to allow dragging and dropping to move the single command button around the form by using this code.

```
Private sub Form_DragDrop (Source As Control, X As Single, Y As Single )
    Source . Move X, Y
End sub
```

The type of source parameter is a control. We can refer to its properties and methods by using the dot notation. The following code is used for apply a method or setting a property.

If TypeOf Control Is.....

Or

If TypeName (control) =
Statement.

The events, methods and properties used for dragging and dropping.

Item	description
DragMode property	Allows automatic dragging(value=1) or manual dragging(value=0)
DragIcon property	Set this to change from the gray rectangle to a custom icon when dragging
DragDrop event	Associated with the target of the operation, generated when the source is dropped on the target control
DragOver event	Associated with any control the source control passes over during dragging
Drag method	Starts or stops dragging when DragMode is set to manual.

Manual Dragging :

If we have left value of the DragMode property at its default of 0, then we must use the Drag method to allow dragging of the control.

Syntax :

Control . Drag TypeOfAction

The TypeOfAction is an integer value from 0 to 2. That is

<i>Control.Drag 0</i>	Cancel dragging
<i>Control.Drag 1</i>	Begin dragging
<i>Control.Drag 2</i>	Drop dragging

The DragOverevent :

All VB objects except menus and timers will detect if a control is passing over them. This event monitors the path a control takes while being dragged. We can the background color of the control being passed over. That event procedure is,

```
Private sub Form_DragOver (Source As Control, X As Single, Y As Single, State As Integer )
```

Where the X and Y parameters are CurrentX and CurrentY values. The State parameter has 3 values.

Value of State	Description
0	Source is now inside target
1	Source is just left of target
2	Source moved inside target

To create the drawing pad, you need to insert a common dialog control, a picture box, four text boxes , six command buttons and the necessary labels. The function of the common dialog control is to assist the users to choose colors. The text boxes are for the user to enter the coordinates and the picture box is to display the pictures drawn.

The method to draw a straight line is **Line**, and the syntax is as follows:

Picture1.Line (x1, y1)-(x2, y2), color

where picture1 is the picture box, (x1,y1) is the coordinates of the starting point, (x2, y2) is the ending point and color understandably is the color of the line.

The syntax to draw a non-solid rectangle is

Picture1.Line (x1, y1)-(x2, y2), color, B

The syntax to draw a solid rectangle is

Picture1.Line (x1, y1)-(x2, y2), color, BF

The syntax to draw a circle is

Picture1.Circle (x3, y3), r, color

Where (x3, y3) is the center of the circle and r is the radius.

if you wish to draw a solid circle and fill it with the selected color, then add two more lines to the above syntax:

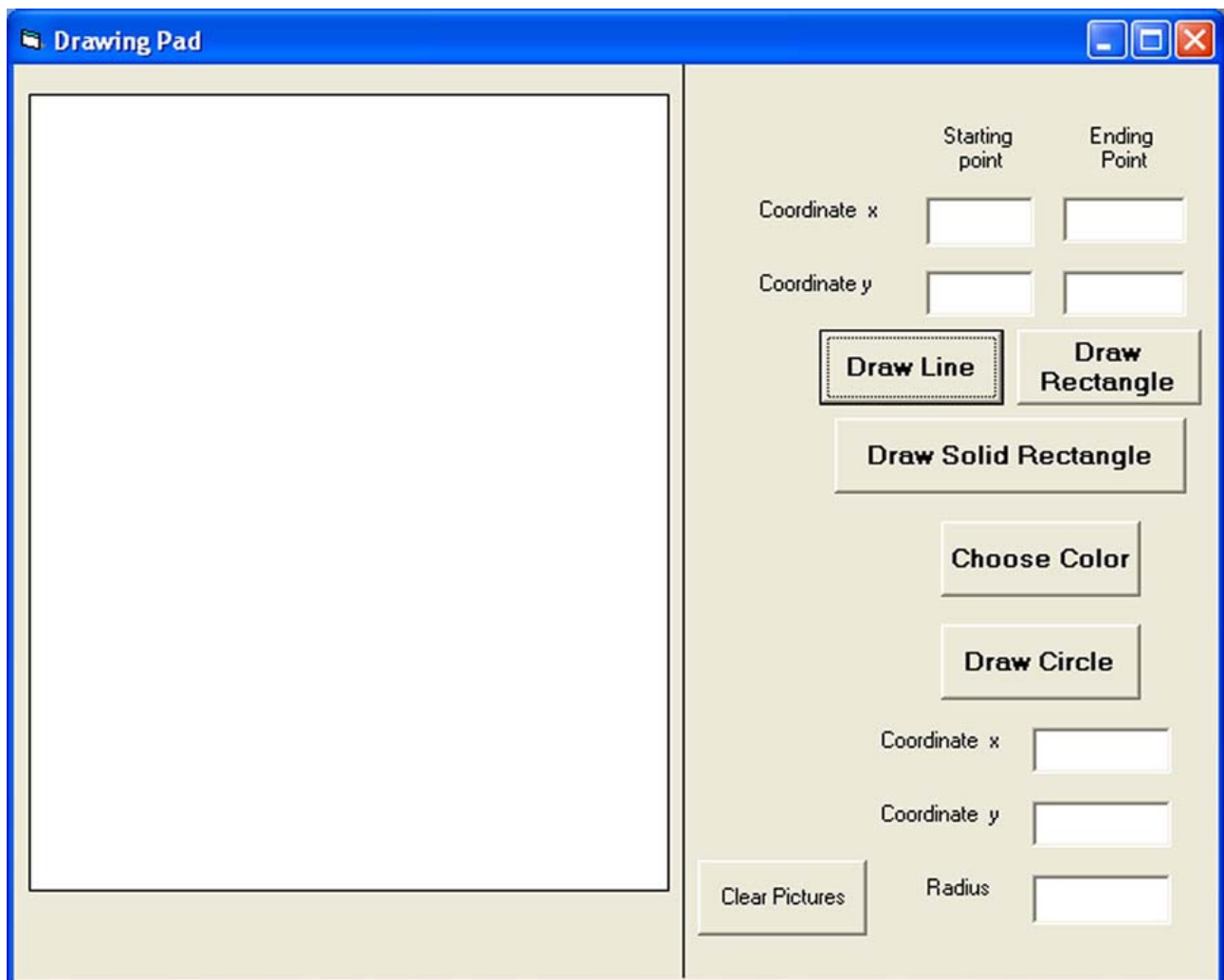
Picture1.FillStyle = vbSolid

Picture1.FillColor = color

The syntax to clear the picture is

Picture1.Cls

The Interface



```
Private Sub cmd_Rectangle_Click()  
x1 = Text1.Text  
y1 = Text2.Text  
x2 = Text3.Text  
y2 = Text4.Text  
Picture1.Line (x1, y1)-(x2, y2), color, B  
End Sub
```



```

Private Sub cmd_Color_Click()
CommonDialog1.Flags = &H1&
CommonDialog1.ShowColor
color = CommonDialog1.color
End Sub

Private Sub cmd_Circle_Click()
On Error GoToaddvalues
x3 = Text5.Text
y3 = Text6.Text
r = Text7.Text

Picture1.FillStyle = vbSolid
Picture1.FillColor = color
Picture1.Circle (x3, y3), r, color
Exit Sub

addvalues:
MsgBox ("Please fill in the coordinates ,the radius and the color")

End Sub

Private Sub Command5_Click()
Picture1.Cls
End Sub

Private Sub cmd_SolidRect_Click()
x1 = Text1.Text
y1 = Text2.Text
x2 = Text3.Text
y2 = Text4.Text
Picture1.Line (x1, y1)-(x2, y2), color, BF
End Sub

```

FILE COMMANDS :

VB has 6 commands that interact with the OS, that handle files and drives. We use these commands with a string or string variable.

MkDir "TESTDIR"

Would add a subdirectory called TESTDIR to the current directory. The line

MkDir" C:\TESTDIR"

Would add the subdirectory to the root directory of the C drive.

The command that handles files also accept the normal file handling.

For example,

Kill “ *.*”

Deletes all the files in the current directory.

Command	Function
ChDrive	Changes the logged drive for the underlying OS
ChDir	Changes the default directory
MkDir	Makes a new Directory
RmDir	Remove a directory
Name	Changes the name of a file or moves a file from one directory to another on the same drive
Kill	Deletes a file from a disk

File handling functions :

There are Four types of functions.

1. The FileCopy function
2. The FileDateTime function
3. The GetAttr function
4. The SetAttr function

The FileCopyfunction :

It copies a file from the source path to another path by the following syntax.

FileCopy source, destination

The FileDateTImefunction :

It returns the date and time a file was created or last modified by the following syntax.

FileDateTime (pathname)

The GetAttrfunction :

It returns an integer. Using masking techniques to get the individual bits, we can determine the how the various attributes are set.

GetAttr (pathname) As Integer

The SetAttrfunction :

It sets attribute information for files. Using the same bit values given, we can change the various attributes.

SetAttrpathname, attributes

SEQUENTIAL FILES :

When the sequential files is that of recording information on a cassette tape. These are analogous to easy tasks for a cassette recorder, such as recording an album on a blank tape, will be easy.

To avoid unnecessary work, use a sequential file.

- Rarely make changes within the file.
- Process the file's information from start to finish, without needing to constantly jump around
- Add information to the file at the end of the file

Some common operations on a cassette tape and the analogous operations on a sequential text file called TEST in the currently active directory.

Operation	VB equivalent
Rewind the tape, press and pause	Open "TEST" for input as #1
Rewind the tape, press record and pause	Open "TEST" for output as #1
Press stop	Close #1

The LOF command :

Once a file is open, we can use the VB "length of file" command LOF() to learn how large the file is, instead of using the FileLen command.

```
Sub Form_Click ()
    Open "Test 1" for Output As #1
    Open "Test 2" for Output As #2
    Print LOF(1)
    Print LOF(2)
    Close
End sub
```

The FreeFilecommand :

If we are writing all of the code for the program, we can keep track of the file identifier numbers. The way to find an unused file identifier is with the command FreeFile. The value of FreeFile is always the next unused file ID number.

```
FileNumber = FreeFile
```

RANDOM ACCESS FILES :

Random access files are stored on a disk in such a way they have much the same advantages and disadvantages as the song collector's tapes. We can access to individual pieces of information, but only at same cost.

In random access file, however, the notion of a record is built in. a random-access file is a special disk file arranged by records. This lets we immediately move to the 15th record without having to pass through the 14 before it.

When we first set up a random-access file, we specify the maximum length for each record. when we enter data for an individual record, we can, of course, put in less information than this preset limit , but we can never put in more. We close a file opened for random access by using the Close command followed by the file ID number.

Category	Size
Author	20
Title	30
Subject	15
Publisher	20
Miscellaneous	13

FILE SYSTEM CONTROLS :

VB allows users to select a new drive, see the hierarchical directory structure of a disk. If we want to tell the underlying operating system to change drives or directories as the result of a mouse click by a user. Our code checks what the user has done to the drive list box and passes this information on to the file list box. The changes in the directory list box are passed on to the file list box.

File List Boxes :

A file list box defaults to displaying the files in the current directory. As with any list box, we can control the position, size, color, and font characteristics at design time or via code. Most of the properties of a list box are identical to those of ordinary list boxes.

For example, as with all list boxes, when the number of items can't fit the current size of the control, VB automatically adds vertical scroll bars. This lets the user move through the list of files using the scroll bars. We can set the size, position, or font properties of the file list boxes via the properties window or via code, as needed. Similarly, file list boxes can respond to all the events that list boxes can detect.

Directory List Boxes :

This displays the directory structure of the current drive. The current directory shows up as an open file folder. Sub directories of the current directory are shown as closed folders, and directories above the current directory are shown as non-shaded open folders.

The list property for a directory list box works a little differently than it does for the file list boxes. While subdirectories of the current directory are numbered from 0 to listCount-1, VB uses negative indexes for the current directory and its parent and grandparent directories.

Drive List Boxes :

Unlike file and directory list boxes, drive list boxes are pull-down boxes. Drive list boxes begin by displaying the current drive, and then when the user clicks on the arrow, VB pulls down a list of all valid drives.

The key property for a drive list box is the drive property, which can be used to return or reset the current drive. For example, to synchronize a drive list box with a directory list box, all we need is code that looks like this,

```
Sub drvBox_change( )  
    dirBox.Path = drvBox.Drive  
End sub
```

On the other hand, if we also want to change the logged drive that the underlying operating system is using, enter this code :

```
Sub drvBox_change( )  
    dirBox.Path = drvBox.Drive  
    ChDrivedrvBox.Drive  
End sub
```

Trying All the file controls Together :

When we have all three file system controls on a form like the one shown below, we have to communicate the changes among the controls in order to have VB show what the user wants to see.

For example, if the user selects a new drive, VB activates the change event procedure for the drive box. The n the following occurs.

1. The change event procedure for the drive box assigns the Drive property to the directory box's path property.
2. This changes the display in the directory list box by triggering the change event procedure for the directory list box.
3. Inside the change event procedure, we assign the path property to the file list box's path property. This updates the file list box.

In this project, you need to insert a **ComboBox**, a **DriveListBox**, a **DirListBox**, a **TextBox** and a **FileListBox** into your form. We will briefly explain again the function of each of the above controls.

- **ComboBox**- to display and enable selection of different type of files.
- **DriveListBox**- to allow selection of different drives available on your PC.
- **DirListBox** - To display directories
- **TextBox** - To display selected files
- **FileListBox**- To display files that are available

Relevant codes must be written to coordinate all the above controls so that the application can work properly. The program should flow in the following logical way:

Step 1: The user chooses the type of files he wants to play.

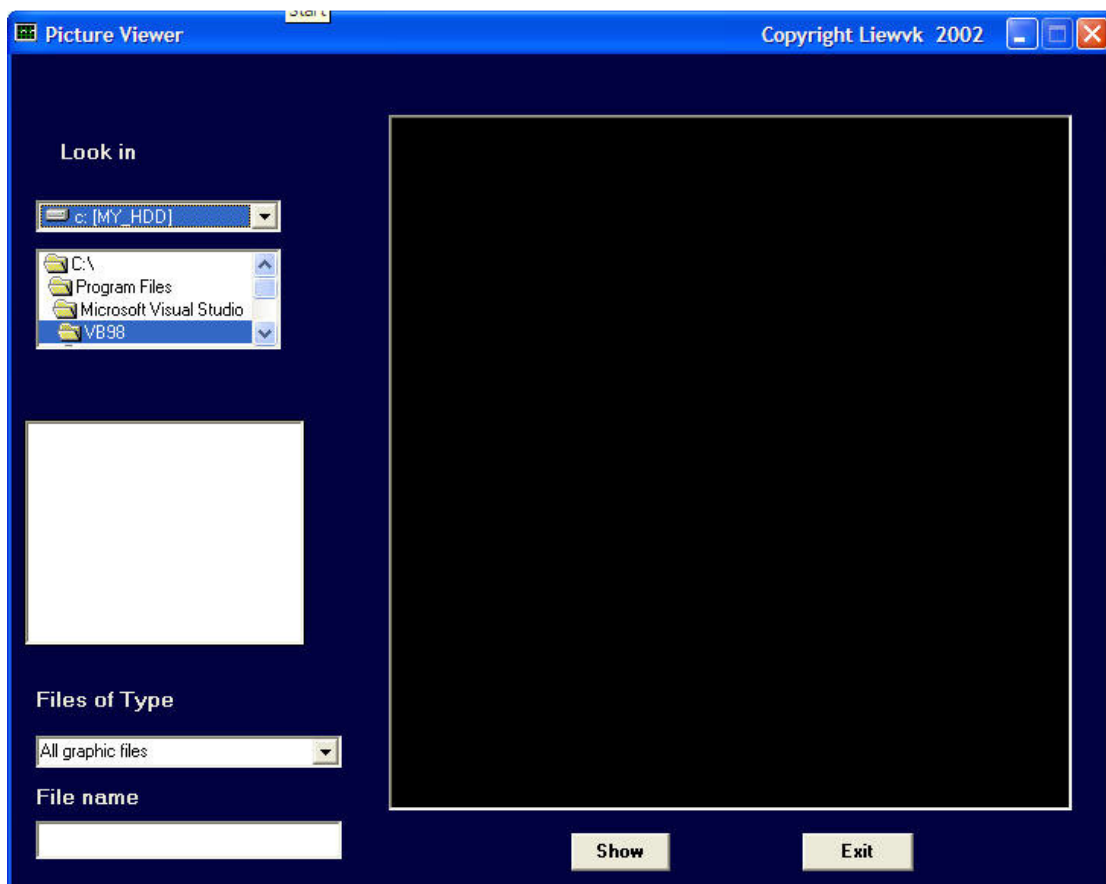
Step2:The user selects the drive that might contains the relevant graphic files.

Step 3:The user looks into directories and subdirectories for the files specified in step1. The files should be displayed in the FileListBox.

Step 4: The user selects the files from the FileListBox and click the Show button.

Step 5: The user clicks on Exit button to end the application.

The Interface



The Code

```
Private Sub Form_Load()  
    'To center the player  
    Left = (Screen.Width - Width) \ 2  
    Top = (Screen.Height - Height)\2  
  
    Combo1.Text = "All graphic files"  
    Combo1.AddItem "All graphic files"  
    Combo1.AddItem "All files"  
End Sub  
  
Private Sub Combo1_Change()  
    If ListIndex = 0 Then  
        File1.Pattern = ("*.bmp;*.wmf;*.jpg;*.gif")  
    Else  
        File1.Pattern = ("*.*)" )  
    End If  
End Sub  
'Specific the types of files to load  
Private Sub Dir1_Change()  
    File1.Path = Dir1.Path  
    File1.Pattern = ("*.bmp;*.wmf;*.jpg;*.gif")  
End Sub  
'Changing Drives  
Private Sub Drive1_Change()  
    Dir1.Path = Drive1.Drive  
End Sub  
Private Sub File1_Click()  
    If Combo1.ListIndex = 0 Then  
        File1.Pattern = ("*.bmp;*.wmf;*.jpg;*.gif")  
    Else  
        File1.Pattern = ("*.*)" )  
    End If  
  
    If Right(File1.Path, 1) <> "\" Then  
        filenam = File1.Path + "\" + File1.FileName  
    Else
```

```
filenam = File1.Path + File1.FileName
End If
Text1.Text = filenam
End Sub
```

```
Private Sub show_Click()
If Right(File1.Path, 1) <> "\" Then
filenam = File1.Path + "\" + File1.FileName
Else
filenam = File1.Path + File1.FileName
End If
'To load the picture into the picture box
picture1.Picture = LoadPicture(filenam)
End Sub
```

THE FILE SYSTEM OBJECTS :

The FileSystemObject has rather a lot of built in properties and many methods associated with it, the basic idea of how to use it is pretty simple many methods associated with it, the basic idea of how to use it pretty simple.

- We will always need to add the reference to the scripting library.
- We will always need to make a new “FileSystemObject” as our first step.
- Once we have a new FileSystemObject, it has a “Drives” collection that tells us about the individual drives on the user machines.
- The Drives collection can give us a “folders” collection that tells us about the folders on a drive.
- The folders collection can have another folders collection inside of it for its subfolders and a files collection for the files in the folder.
- We will have many ways of getting at individual drives, folders, or files , and of reading off their properties

Main properties of a Drive Object :

- AvailableSpace
- DriveLetter
- DriveType
- FileSystem
- FreeSpace
- IsReady

- Path
- RootFolder
- SerialNumber
- ShareName
- TotalSize
- VolumeName

Most common methods of the FileSystemObject :

CopyFile source, destination [overwrite] :

Another way to copy a file. The overwrite flag allows us to choose to overwrite an existing file with that name or not.

CopyFolder source, destination [overwrite] :

A way to copy a folder or directory that will automatically copy all subdirectories and sub-subdirectories as well.

CreateFolder (foldername) :

Creates a folder by name.

DeleteFile filespec[,force] :

Another way to delete a file. If the optional force parameter is true, this will delete read-only file.

DeleteFolder folderspec[,force] :

A way to delete a folder or directory.

Creating and Reading files using Common Dialog Box

This example uses the common dialog box to create and read the text file, which is much easier than the previous examples. Many operations are handled by the common dialog box.

The following is the program

```
Dim linetext As String
Private Sub open_Click()
CommonDialog1.Filter = "Text files (*.txt)*.txt"
CommonDialog1.ShowOpen

If CommonDialog1.FileName <> "" Then
Open CommonDialog1.FileName For Input As #1
Do
```

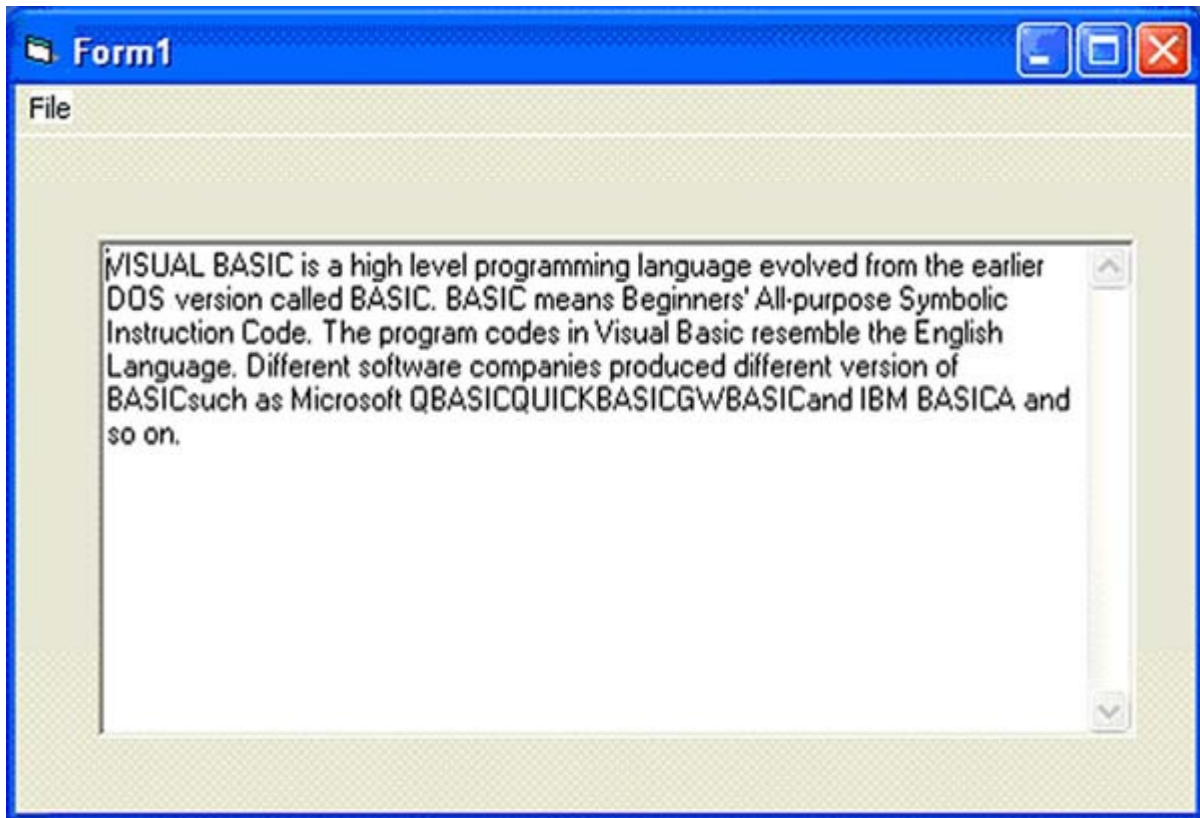
```

Input #1, linetext
Text1.Text = Text1.Text &linetext
Loop Until EOF(1)
End If
Close #1
End Sub
Private Sub save_Click()
CommonDialog1.Filter = "Text files (*.txt)|*.txt"
CommonDialog1.ShowSave
If CommonDialog1.FileName <> "" Then
Open CommonDialog1.FileName For Output As #1
Print #1, Text1.Text
Close #1
End If
End Sub

```

The syntax **CommonDialog1.Filter = "Text files (*.txt)|*.txt"** ensures that only the textfile is read or saved .The statement **CommonDialog1.ShowOpen** is to display the open file dialog box and the statement **CommonDialog1.ShowSave** is to display the save file dialog box. **Text1.Text = Text1.Text &linetext** is to read the data and display them in the Text1 textbox

The Output window is shown below:



THE CLIPBOARD :

VB uses the clipboard for its cut-and-paste editing feature, and we can use the clipboard together with the properties given in the section “selecting Text in VB” to implement features in our projects.

It can hold only one piece of the same kind of data at a time. If we send new information of the same format to the clipboard, we wipe out what was there before. Sometimes, however we will want to make sure that the clipboard is completely free before working with it. To do this, add a line of code inside,

Clipboard . Clear

This applies the clear method to the predefined clipboard object. If we need to send text to and from the clipboard, use the two additional methods.

Clipboard.setText :

This method is used in the following form.

Clipboard.SetTextstringData

This send the string information contained in the variable or string expression StringData to the clipboard, wiping out whatever text was there.

Clipboard.GetText :

It takes a copy of the text currently stored in the clipboard. Because the text contents of the clipboard remain intact until we explicitly clear the clipboard or send new text to it. the form is,

Destination = Clipboard.GetText

THE WITH STATEMENT :

We can use the With statement for getting at the parts of a record. For example

```
With YourName
    .Name = "sudha"
    .Salary = 10000
End With
```

We can even nest with statements if one of the components of a record is itself a record

```
Dim MyStatsAsExpandedVitalInfo
With MyStatus
    .Name = "ziyaudeen"
    With .Salary
        .SalInJan = 1000
        .SalInFeb = 2000
        .
        .
        .
    End With
End With
```

We can also use With statement with properties of objects.

```
With txtbox
    .Height = 2000
    .Width = 2000
    .Text = "AVS COLLEGE "
End With
```

ENUMS :

In visual basic, we can assign successive integers to the variable by using Enums or enumerated constants.

```
EnumDaysOfTheWeek
    Sunday = 1
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
```

Now visual basic automatically assigns successive integers to the days of the week, that is Monday = 2, Tuesday = 3, and so on.

After they have been defined, Enums can be used as the parameters for functions and procedures, so they can make code much clearer.

```
Function Salary (x As DaysOfTheWeek) As single
    Select Case X
        Case Saturday, Sunday
            Salary = 1.5
        Case Else
            Salary = 1.0
    End Select
End Function
```

CONTROL ARRAYS :

Some time we can use the same control name more than once while designing a VB application or use CTRL+C, CTRL+V to copy an existing control from one place on the form to another, VB asks you whether you really want to create a control array by popping up the message box. Then click the yes button.

If we assign any number to the Index property of a control at design time, VB automatically creates a control array. We can create a control array without having to use two controls at design time. We can have up to 255 elements in a control array, but that would be very unusual, as it would waste too many Windows resources.

Working with a control array :

Suppose we want to work with the Change procedure for one of the two elements of the txtMoney text box control array. When we move to the Code window, by double clicking one of these text boxes from the control array. As with arrays , the index parameter is the key to the smooth functioning of control arrays.

Add the following code to the event procedure template.

```
Private Sub txtMoney_Change (Index As Integer)
    If Index = 0 then
        MsgBox "Sudha"
    Else
        MsgBox "Jaya"
    End If
End Sub
```

Now , when we type in one of the text boxes :

1. Visual Basic calls this event procedure
2. VB passes the index parameter of the control we typed in to the procedure.

The event procedure can use the index to determine what to do. When we type in the text box with the Index property of zero, VB activates the If clause inside this event procedure and so displays text telling us which box we typed in. otherwise, VB processes the Else clause. The If-Then-Else, combined with the index parameter, lets the event procedure determine where we typed.

Adding and Removing Controls in a Control array :

We can add a control to a control array at design time, also we can remove it by changing the control name or deleting the control at design time. Once create a control array, we must change all the names of all the controls in the control array before VB will let us eliminate the Index property.

The Load statement :

Once we have created a control array at design time, we can add controls while the application is running. To do this, we use a method called Load statement.

Load txtMoney (I)

where I will be the index of the element.

When adding controls to a control array at run time, start up a new project and add the following code to the Form_Load event procedure.

```
Private Sub Form_Load ( )
    Dim I As Integer
    For I = 2 to 5
        Load txtMoney (I)
        txtMoney (I).text = "Text box #" + Str$(I)
    Next I
End Sub
```

If you run this program, VB loads a new element of a control array, the object is invisible, the visible property of the new control is set to False. We have to change this to true. And all other properties of new control are copied from the object that has the lowest index in the array. Even if you modify the preceding Form_Load procedure to read.

```
Private Sub Form_Load ( )
    For I = 2 to 5
        Load txtMoney (I)
        txtMoney (I).Text = " Text box #" + Str$(I)
        txtMoney (I).Visible = true
    Next I
End Sub
```

We will see the text box with index number 5. This is because the Left and Top properties start out the same for all four of the newly created controls. The Left and Top properties determine where we see the control, that is newly created controls in a control array default to being stacked one on top of the other.

The unload statement :

We can use the Unload statement to remove any element of a control array that we added at run time via the Load command. We cannot use the Unload statement to remove original elements of a control array that were created at design time. For example,

```
Private Sub Form_Click ( )
    Static I As Integer
    If I < 4 Then
        Unload txtMoney (I+2)
        I = I + 1
    Else
        Exit Sub
    End If
End Sub
```

Each click on an empty place in the form removes the next controls in the control array. It will not remove two elements in the control array because of the line

```
If I < 4 then .....
```

This is necessary because we can only load or unload an element of a control array once. If we try to load or unload a control array element twice in succession, VB gives a run-time error.

Control array

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

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

You create a control and then assign a numeric, non-negative value to its Index property; you have thus created a control array with just one element.

You create two controls of the same class and assign them an identical Name property. Visual Basic shows a dialog box warning you that there's already a control with that name and asks whether you want to create a control array. Click on the Yes button.

You select a control on the form, press Ctrl+C to copy it to the clipboard, and then press Ctrl+V to paste a new instance of the control, which has the same Name property as the original one. Visual Basic shows the warning mentioned in the previous bullet.

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

Controls that belong to the same control array share the same set of event procedures; this often dramatically reduces the amount of code you have to write to respond to a user's actions.

You can dynamically add new elements to a control array at run time; in other words, you can effectively create new controls that didn't exist at design time.

Elements of control arrays consume fewer resources than regular controls and tend to produce smaller executables. Besides, Visual Basic forms can host up to 256 different control names, but a control array counts as one against this number. In other words, control arrays let you effectively overcome this limit.

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

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

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

Sharing Event Procedures

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

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

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

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



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

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

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

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

Creating Controls at Run Time

Control arrays can be created at run time using the statements

- `Load object (Index %)`
- `Unload object (Index %)`

Where `object` is the name of the control to add or delete from the control array. `Index %` is the value of the index in the array. The control array to be added must be an element of the existing array created at design time with an index value of 0. When a new element of a control array is loaded, most of the property settings are copied from the lowest existing element in the array.

Following example illustrates the use of the control array.

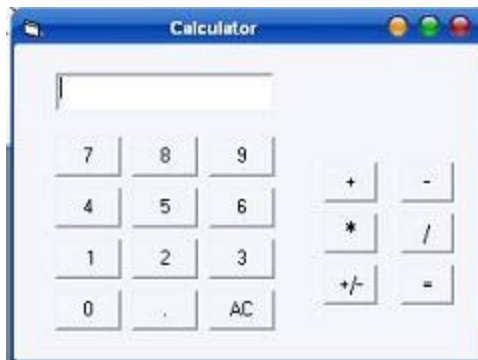
* Open a Standard EXE project and save the Form as `Calculator.frm` and save the Project as `Calculator.vbp`.

* Design the form as shown below.

Object	Property	Setting
Form	Caption	Calculator
	Name	frmCalculator
CommandButton	Caption	1
	Name	cmd
	Index	0

CommandButton	Caption Name Index	2 cmd 1
CommandButton	Caption Name Index	3 cmd 2
CommandButton	Caption Name Index	4 cmd 3
CommandButton	Caption Name Index	5 cmd 4
CommandButton	Caption Name Index	6 cmd 5
CommandButton	Caption Name Index	7 cmd 6
CommandButton	Caption Name Index	8 cmd 7
CommandButton	Caption Name Index	9 cmd 8
CommandButton	Caption	0

	Name	cmd
	Index	10
CommandButton	Caption	.
	Name	cmd
	Index	11
CommandButton	Caption	AC
	Name	cmdAC
CommandButton	Caption	+
	Name	cmdPlus
CommandButton	Caption	-
	Name	cmdMinus
CommandButton	Caption	*
	Name	cmdMultiply
CommandButton	Caption	/
	Name	cmdDivide
CommandButton	Caption	+/-
	Name	cmdNeg
TextBox	Name	txtDisplay
	Text	(empty)
CommandButton	Caption	=
	Name	cmdEqual



The following variables are declared inside the general declaration

```
Dim Current As Double
Dim Previous As Double
Dim Choice As String
Dim Result As Double
```

The following code is entered in the cmd_Click() (Control Array) event procedure

```
Private Sub cmd_Click(Index As Integer)
txtDisplay.Text = txtDisplay.Text&cmd(Index).Caption
'&is the concatenation operator
Current = Val(txtDisplay.Text)
End Sub
```

The following code is entered in the cmdAC_Click () event procedure

```
Private Sub cmdAC_Click()
Current = Previous = 0
txtDisplay.Text = ""
End Sub
```

The below code is entered in the cmdNeg_Click() procedure

```
Private Sub cmdNeg_Click()
Current = -Current
txtDisplay.Text = Current
End Sub
```

The following code is entered in the click events of the cmdPlus, cmdMinus, cmdMultiply, cmdDevide controls respectively.

```
Private Sub cmdDevide_Click()
txtDisplay.Text = ""
Previous = Current
Current = 0
Choice = "/"
End Sub
```

```
Private Sub cmdMinus_Click()
txtDisplay.Text = ""
Previous = Current
Current = 0
Choice = "-"
End Sub
```

```
Private Sub cmdMultiply_Click()  
txtDisplay.Text = ""  
Previous = Current  
Current = 0  
Choice = "*"   
End Sub
```

```
Private Sub cmdPlus_Click()  
txtDisplay.Text = ""  
Previous = Current  
Current = 0  
Choice = "+"   
End Sub
```

To print the result on the text box, the following code is entered in the cmdEqual_Click () event procedure.

```
Private Sub cmdEqual_Click()  
  
Select Case Choice  
  
Case "+"  
Result = Previous + Current  
txtDisplay.Text = Result  
Case "-"  
Result = Previous - Current  
txtDisplay.Text = Result  
Case "*"  
Result = Previous * Current  
txtDisplay.Text = Result  
Case "/"  
Result = Previous / Current  
txtDisplay.Text = Result  
End Select  
  
Current = Result  
  
End Sub
```

Save and run the project. On clicking digits of user's choice and an operator button, the output appears.

Iterating on the Items of a Control Array

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

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

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

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

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

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

LIST AND COMBO BOXES :

We can give the list of values to the List box. VB automatically adds vertical scroll bars whenever the list box is too small for all the items it contains. We might want this application to let the user select a data by number of elements rather than by scrolling through the list. To allow users to input data as well as make choices from a list, use combo box.

There are two types of combo boxes, and which one we get depends on the value of the Style property.

- If the value of the Style property is set to the default value of 0, we get a combo box with an arrow. If the user clicks the arrow, then drop-down list of choices given in the box.

- If the value is 1, the user sees the combo box with the list already dropped down.

In both cases, the user still has a text area to enter information. There is one other possible choice,

- If the value of the Style property of a combo box is 2, we will see a pull-down list box, the text area will disappear.

There are also two types of list boxes, and working with the Style property lets we determine if the box shows little check boxes next to the items.

Sorting Lists and Combo Boxes :

The items in List or Combo box can be sorted in ASCII order by simply setting the sorted property to True.

Manipulating the Items on a List or Combo box :

We usually add or remove items from a list or combo box while the project is running. Use the AddItem method to add an item to a list or combo box. The syntax is,

Listname.AddItem item [index]

ListName is the control name of the list or combo box, andItem is a String. If the sorted property is True for the list or combo box, then Item goes where ANSII order places it. If it is false, VB places Item either at the position determined by the Index Parameter or at the end of the list when we don't specify the index.

The Index parameter must be an integer, and value of 0 (not 1) means we are the beginning of the list. The Index position has no effect on where the item is added to the list if the sorted property of the list box is set to true.

Other common List and Combo box properties :

List :

This is a property of both list and combo boxes, and it is for all practical purposes like a zero-based string array that contains all the items on the list.

1stTheBox (0)
1stTheBox (1)
1stTheBox (2)

It gives us another way to initialize the items in the list or combo box instead of doing it at run time.

ListCount:

This is a property of both list and combo boxes, and it gives the number of items on the list. Since the Index property starts at 0, in order to analyze the contents of the list using a For-Next loop, write the limits of the loop as,

For I% = 0 to 1stBoxName.ListCount - 1

Text :

It gives the currently selected item stored as a string. This, of course, changes as the user moves through the list box. This changes as the user moves through the list box. If we need the numeric equivalent of this string, just apply the correct conversion function (CInt, CSng, etc).

Columns and MultiSelect :

The columns property controls the number of columns in a list box. If the value is 0, we get a normal single-column list box with vertical scrolling. If the value is 1, we get a single column list box with horizontal scrolling. If the value is greater than 1, we get multiple columns.

The Multiselect property controls whether the user can select more than one item from the list. There are three possible values for this property.

Type of Selection	Value	How It works
No multiselection allowed	0	
Simple multiselection	1	Use ordinary windows techniques and mouse dragging to select more items.
Extended multiselection	2	SHIFT+mouse click extends the selection to include all list items between the current selection and the location of the click. Pressing and clicking the mouse selects or deselects an item in the list.

List Box Events :

List Boxes respond to 12 events, the most two important events for list boxes are the Click and Double-click events. An important Windows convention is that clicking on an item selects the item but does not choose the item. This is reserved for double – clicking on the item. We don't write code for the Click event procedure for a list box but only for the Double-click event procedure.

Combo Box Events :

Combo boxes respond to many of the same events as like list boxes. We can analyze which keys were pressed or whether the box has received or lost the focus. The first event is the Change event. It occurs only for pull-down combo boxes and simple combo boxes (Style = 0 or 1). VB does not generate event if we are using a pull-down list box (Style = 2).

THE FLEX GRID CONTROL :

The flex grid control build spreadsheet like feature into project or display tabular information neatly and efficiently. we can use a flex grid control larger magic square than the control array of label allowed you to do earlier. Moreover. The flex grid version works faster and uses fewer windows resources than a control array.

This control display a rectangular grid of rows and columns at design time. How many rows and columns we can see at design time depends both on the number of rows and columns we have set and the current size of the grid. Each grid member is usually called a cell. Cells are hold text, bitmaps, or icons, and we can even have some cells holding text and other holding graphics.

Numbers must be translated back and forth using the right conversion and Str(\$) functions or the Variant data type must be used with text boxes. Users can move from cell to cell by using the arrow keys or the mouse. VB handles such movement automatically. Users move the control around the grid.

Users can work with contiguous groups of cells in the grid, usually called regions, by clicking a cell and dragging the mouse or by pressing SHIFT plus an arrow key to select the region. Once a region is selected, code can be used to analyze or change the contents.

General Properties :

Cols, Rows :

These properties determine the number of rows and columns in the grid. The default values for each of these properties is 2, but we can reset them in code or via the properties window. They must be integers, and the syntax is,

```
GridName.cols = NumOfCols%  
GridName.Rows = NumOfRows%
```

We can also add an optional form name, for example

```
frmDisplay.Cols = 10
```

Col, Row :

These properties set or return the row and column for the currently selected cell inside the grid. These are only available at run time. Use their values to determine where inside the grid the user is. Since both Col and Row start out at zero, the top left corner cell has Col value 0 and Row value 0.

ColPosition, RowPosition :

These are used to move whole rows and columns around in your grid. The syntax is,

```
GridName.ColPosition (number)[ = value]  
GridName.RowPosition (number) [ = value]
```

ColWidth, RowHeight :

These are specify the width of a specific column or height of a specific row. They can only be set via code. Both are measured in twips. The syntax is

```
GridName.Colwidth(ColNumber%) = width%  
GridName.RowHeight(RowNumber%) = Height%
```

Text, TextMatrix :

The Text property sets or returns the text inside the current cell. The TextMatrix property, which has the following syntax,

```
Textmatrix (rowindex, colindex) = [string]
```

We set or retrieve the text in an arbitrary cell without needing to change the Row and Col properties.

CellLeft, CellWidth, CellTop, and CellHeight :

These properties give the size and location of a cell relative to its container. We can use these extensively in the section on adding an editing capability to grid.

ColAlignment :

There are three possible settings for data inside a column. We can left- justify (value = 0), right-justify (value = 1), or center the text (value = 2). The syntax is

```
GridName.ColAlignment(Index%) = Settings%
```

GridLines, ScrollBars :

These are used to control whether grid lines and scroll bars appear. The default is to show grid lines and to have both horizontal and vertical scroll bars.

LeftCol, TopRow :

These are used to control which are the leftmost column and highest row displayed from a grid. It can be set in code, and the syntax for both of these properties is similar.

```
GridName.LeftCol = LeftmostCol%  
GridName.TopRow = HighestRow%
```

Sorting a Grid :

One of the most powerful features of the flex grid is its ability to sort rows according to the columns we select. If we change the Sort property at design time, the grids starts out sorted. If we change it at run time, VB sorts the rows of the grid immediately after it processes the statement. The syntax for the sort property takes the form,

```
FlexGridName.Sort = Value
```

Constant	Value	Description
flexSortNone	0	None
flexSortGenericAscending	1	Sort in ascending order
flexSortGenericDescending	2	Sort in descending order
flexSortNumericAscending	3	Ascending but converts strings to numbers
flexSortNumericDescending	4	Descending but converts strings to numbers
flexSortStringNoCaseAscending	5	Ascending but case-insensitive
flexSortStringNoCaseDescending	6	Descending but case-insensitive
flexSortStringAscending	7	Ascending but case - sensitive
flexSortStringDescending	8	Descending but case - sensitive
Custom	9	Uses the compare event to compare rows

Events and Methods for Grid Controls :

EnterCell, LeaveCell :

EnterCell is triggered when the user clicks inside a cell that is different from the one currently selected. LeaveCell is triggered right before the active cell changes to a new one.

RowColChange :

This event is also triggered when the current cell changes. It occurs after LeaveCell and EnterCell.

SelChange :

This event is activated when the selected region changes, either because the user has moved around in the grid or the code has directly changed one of the properties given.

Compare Event :

This event is used to order the columns any way. When we set the sort property to 9, VB will automatically trigger this event. The syntax is

Private Sub object_Compare(row1 As Integer, row2 As Integer, cmp As Integer)

Change the value of the cmp parameter in order to tell VB which row to consider as being less than the other row. There are three possible ways to change the cmp parameter. They are,

Cmp setting	Description
-1	If row1 should appear before row2 when VB to sort the grid
0	If both rows are equal or the order in which they appear is irrelevant
1	If row1 should appear after row2 when VB to sort the grid.

CODE MODULES : GLOBAL PROCEDURES AND GLOBAL VARIABLES :

We can reuse the code, by store the procedures and functions in separate modules rather than leaving them attached to a form. A standard code module is where we put code that we want to be accessible to all code in a project. It has no visual components. They are also useful for reusing code.

The Sub and Function procedures in code modules default so that they are available the whole project. The buzzword is that they have global scope. This is placement of a procedure or of a declaration of a variable or constant determines which parts of the project can use it.

VB automatically puts the keyword private in front of any event procedures attached to a form, we should use the keyword private for any Sub and Function procedures that are attached to a form. And the public, any code attached to a form that marked Public can be used by simply prefixing it with the form's name.

public variables :

We make a variable attached to a form into a global variable visible to every part of a project, use a statement of the form.

Public VariableNameAsVariableType

Eg:

Public Rate As Single

Scope of procedures :

Code modules can also have module-level variables that are visible only to the code attached to that module. To make a variable a global variable requires the Public keyword or leaving off the Public access modifier.

If we use the ordinary Private (Dim) declaration syntax that we have been using for code attached to a form in the declarations section of a code module. When we use a Sub or Function procedure inside another procedure, VB follow these steps.

- VB first looks at procedures attached to the current form or module
- If the procedure is not found in the current form or module, VB looks at all code modules attached to the project.

Adding or Removing Existing Code Module :

To add an existing code module, open the Project menu and choose the Add Module option. Choose the Existing tab on the dialog box that opens, this gives a standard File Open dialog box asking for the name of the file. To add code from another file to an existing module, choose Insert | File and then work with the dialog box that pops up.

To remove a code or from module from a project, follow these steps.

- Open the project explorer window by clicking on any part that's visible or by choosing View | project explorer.
- Select the code module or form module we want to remove
- Either choose remove from the right-click menu, or choose the remove item from the project menu.

THE DO EVENTS FUNCTION AND SUB MAIN :

Windows maintains a list of pending events in what is called an Event Queue. The DoEvents statement passes control to windows, and VB gets control back only after the operating system has finished processing the events in its queue.

Eg :

```
Private Sub Form_Load ( )  
    Show  
    Do  
  
    Loop  
End Sub
```

If you run this program, the Exit button doesn't end the program, it doesn't close the form. It is called Tight Loop.

But by using DoEvents statement we can change the above code,

```
Private Sub
    Show
    Do
        DoEvents
    Loop
End Sub
```

The form will close normally when we click on the Exit button.

SUB MAIN :

DoEvents can actually be used as a function, in this case it returns the number of loaded forms. There is no startup form, we have to manage the code for loading all the forms our self. This is done inside a special Sub procedure called Main that may be attached to any code module. We can have only one Sub Main procedure in any project.

```
Sub Main ( )
    frmSplash.Show
    frmSplash.Refresh
    frmSplash.Show
End Sub
```

Once we have set Sub Main as the startup object, VB doesn't load any forms automatically. We have to write the code for this using the Load and show keywords.

Sub Main and DoEvents :

We can use aDoEvents as a function. It returns the number of loaded forms. The framework for this takes the following form.

```
Sub Main ( )
    frmSplash.Show
    frmSplash.Refresh
    Do While DoEvents ( )
        frmSplash.Show
    Loop
End Sub
```

Here the End statement will only be reached when there are no more loaded forms.

Idle Time :

A loop that is processed only when no events are occurring is called an idle loop. Idle loops are written inside the Sub Main.

```
Sub Main ( )
    Load StartUpForm
    Do While DoEvents ( )

    Loop
End Sub
```

Again, this kind of code will continue to execute as long as there are any loaded forms.

ACCESSING WINDOWS FUNCTIONS :

We can create a stand alone applications, include .dll file to run it. A dynamic link library like this one contains specialized functions that a windows program can call on as needed. Windows itself can be thought of as an interlocking set of DLLs containing hundreds of specialized functions. These are called Application Programming Interface (API) functions.

Using an API statement requires a special type of declaration that is much more complicated than the ones used for variables, these are called Declare statements. As a first example of using an API call, consider the plight of left-handed users of windows. They might like to swap the left and right mouse buttons via the windows control panel.

Before we can do this, we need to find out if the buttons have been swapped. The only way to do this is by using a windows API function called GetSystemMetrics. To use this function, add the following Declare statements to the general section of a Code module.

```
Declare Function GetSystemMetrics Lib "users32" Alias _
    "GetSystemMetrics" (ByVal nIndex As Long) As Long
```

As Declare statement indicates, we send this function a long integer value that tells the function what information we want reported back. This function returns a long integer that we can analyze.

Mouse swap:

```
Sub Form_Load
    Const SM_SWAPBUTTON = 23
    If GetSystemMetrics (SM_SWAPBUTTON) Then
        MsgBox "Mouse buttons switched "
    End If
End Sub
```

There are two possibilities for the general form of the Declare statement. For a Sub program in a DLL, use

```
[Public | Private ] Declare Sub name Lib "libname" [Alias_"aliasname"]([arglist])
```

For a function, use

[Public|Private] Declare Sub name Lib "libname" [Alias_"aliasname"]([arglist])[As type]

Where the Lib keyword should be just bookkeeping – it tells VB that a DLL is being called. The libname argument is the name of the DLL that contains the procedure we will be calling.

To use the API Viewer, follow these steps:

- Open the File menu in the API Viewer and choose the file we want to look at.
- After the text file loads, choose what part of the API we need to look at from the API type list box
- Choose the item we want by scrolling through the Available Items list box.
- Click on the Copy button to place the item in the clipboard
- Move the insertion point inside our code window where we want the item to appear, and choose Edit|Paste to copy the item from the clipboard

ERROR TRAPPING :

If we want to prevent fatal errors, the command activates error trapping within a given procedure is,

On Error GoTo...

Where the three dots stand for the label that defines the error trap. The labeled code must be in the current procedure. We cannot jump out of a procedure using an On Error GoTo command.

The On Error GoTo command can occur anywhere in an event, Sub, or Function procedure. Usually, the error-trapping code is inside that procedure. The only exception to this is when one procedure has been called by another.

Once start error trapping with the On Error GoTo command, a run-time error will no longer bomb the program. It should transfer control to a piece of code that identifies the problem and, if possible, fix it.

If the error can be corrected, the Resume statement takes back to the statement that caused the error in the first place. However, we cannot correct an error if we don't know why it happened. We identify the problem by means of either the Err function or the Err object. This gives an integer that can assign to a variable.

For example, if we write

`ErrorNumber = Err.Number`

The value of the variable ErrorNumber is the error number. VB can identify more than 80 run-time errors.

Error Code	Explanation
57	Device
68	Device unavailable
482	A general printer error. VB reports the error whenever the printer driver returns an error code.
483	The printer driver does not support the property

More on the Resume Statement :

A variant on the Resume statement lets bypass the statement that may have caused the problem. If we use

Resume Next

VB begins processing at the statement following the one that caused the error. We can use

OnError Resume Next

To automatically bypass any code that causes an error. We can also resume execution at any line of code that has been previously identified with a label. For this, use

Resume Label

It is unusual to have labels in VB except in connection with error trapping.

The Erl (Error Line) function :

The error handling function Erl is used to find the line that caused the error, and VB is not stopping the program at that time. We can do the following.

- Add line numbers before every statement in the procedure
- Add a Debug.PrintErl statement inside the error trap.

When developing a program, we may want to test how our error handler works. VB includes the statement

Error (Errorcodenumber)

Which, when processed, makes VB behave as if the error described by the given error number has actually occurred. This makes it easier to develop the trap.

Disabling Error Trapping :

If we are confident that we will no longer need an error trap, we can disable error trapping with the statement,

On Error GoTo 0

Similarly, we can change which error trap is in effect by using another On Error GoTo statement. Be sure to have an Exit command between to decide where to go.

CREATING AN OBJECT IN VB :

We can build objects in 3 ways, that is,

1. by adding custom properties to an existing form and then using that form as a class for new instances of the form.
2. By using a special type of module called a class module. It have the advantages that they can be compiled separately and used by other windows applications.
3. To create a custom control.

In all three ways gives the same ideas for adding properties to these classes. The following basic things are needed to do with a new property.

- We want to get its current value
- We want to assign a new value to it.

For the first situation, we use a special type of procedure called a Property get procedure. For the second, we use a Property Let procedure.

For example, suppose we want to add a custom property to a form that will tell whether a form named frmNeedsToBeCentered is centered, and also to center it if we set the property to true. We should need to do.

Set up a Private variable in the declarations section of the form.

```
Private IsCenteredAs Boolean
```

Then add the following procedures to the form

```
Public Property Let Center (X As Boolean)
    IsCentered = X
    If X then
        Me.Move (Screen.Width - Me.Width) / 2
        - Screen.Height - Me.Height) / 2
    End If
End Sub
```

The first line of the code uses the Private variable to store the current value of the property. Now we can use a line of code like

```
Me.Center = true
```

to center the form. From another form or code module, we can use a line of code like this,

```
frmneedsToBeCentered.Center = True
```

it might be useful to know whether a form is centered. For this we ne need to use a Property Get Procedure that returns a Boolean.

```
Public Property Get Center ( ) As Boolean
    Center = Iscentered
End Sub
```

This picks up the current value of the IsCentered Private variable that we are using to hold information about the current value of the property.

BUILDING YOUR OWN CLASSES :

We can add custom properties to a form and then use them as templates for new objects, the most common way to build a new class for new objects in VB is to use a class module. A class module object contains the code for the custom properties and methods that objects defined from it will have.

We can then create new instances of the class from any other module or form. A class module cannot have a visible interface of its own. Each module we create gives the naturally enough, a single class for building new instances of that class.

Once we have a class module, we use the New keyword to create new instances of it. Eg, if FirstClass is the name of a class module in our project, we would create an instance as follows.

Dim AnInstance As new FirstClass

We use the Property procedures to define the properties of our class and use Public Sub and Public Function procedures for its methods.

Creating a New Class Module :

We can create a new class module at design time by choosing Project\Add Class Module. Each class module can respond to only two events.

- Initialize
- Terminate

They are triggered when someone creates an instance of our class or terminates it.

..... **V – UNIT COMPLETED**