CONTENTS

.NET PROGRAMMING

SYLLABUS

UNIT I

Essential Visual basic.Net, Operators - Conditional statements - Loops - Procedure - Scope - Exception handling.

UNIT II

Window forms: Msgbox - Input box, Events_Textboxes - Rich Text Boxes - Latch and link labels - Buttons - Check Boxes Radio Buttons - Panels and group Boxes.

UNIT III

List Boxes - Checked Boxes - Combo boxes - Picture boxes - Menus - Built-in dialog boxes - Printing image lists - Trees and views Text box - Status and Progress Bars and Tab Controls.

UNIT IV

Object Oriented - Inheritance - Graphics and File Handling - Validation Controls - Calendar - Ad Rotator - HTML Controls.

UNIT V

Data - Access - With ADO.NET - Binding Controls to Data Bases - Handling database in coding - Database Access in Web Application.

UNIT I

LESSON 1

VISUAL BASIC.NET

CONTENTS

1.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the history of programming languages
- Create a Visual Basic.NET Windows-based application
- Know how to manage the windows in the IDE

1.1 INTRODUCTION

Visual Basic.Net is an advanced version of Visual Basic 6.0 with 7.0 as the version number, which is seldom used. It looks completely different from all its previous versions though it has all the features of previous versions too still maintaining the essence and beauty of its predecessors. The most visible difference is that everything in Visual Basic.Net is Net enabled.

In 1964 a language was invented for beginner programmers. This language was named BASIC (Beginner's all purpose Symbolic Instruction Code). This language aimed at ease of learning. This led to the foundation of Visual Basic. Microsoft then developed Microsoft BASIC in 1970s. Then came Quick BASIC, which was the most popular version of BASIC on the PC. Microsoft then came out with another successor of BASIC i.e. BASIC with Graphical User Interface named as Visual Basic 1.0 , which was DOS based. But, later on it moved to Windows. With time different features were added to Visual Basic and so new versions came in the market. Each new feature was more powerful and with more features than previous one making it more capable development tool. Visual Basic 4.0 was released with the capability to create components called COM. Version 5.0 came out with the capability to create either compiled or interpreted versions of programs. Prior versions were interpreted.

Visual Basic.NET is Microsoft's latest release of Visual Basic and has been redesigned from the ground up to build.NET applications, including .NET assemblies, class inheritance, web applications, and web services. Visual Basic.NET also includes CLR support for protocols such as XML, HTTP, and SOAP for promoting of loosely coupled applications.

1.2 VISUAL BASIC.NET AS A PROGRAMMING LANGUAGE

Programs are directions given to a computer accomplish a given task. A Brief History of Programming Languages is as follow:

1.2.1 Machine Language

This was early language. It was difficult to program and all the more difficult to debug or modify. This language has following features:

- 0 (for on switch)
- 1 (for off switch)
- Tedious and prone to error

1.2.2 Assembly Language

Assembly was next generation language after machine language. Its features were:

- More advanced than machine languages
- Mnemonics (memory aids). *Examples:* MUL b1, ax
- Require an assembler
- Still difficult to program and modify.

1.2.3 High Level Languages

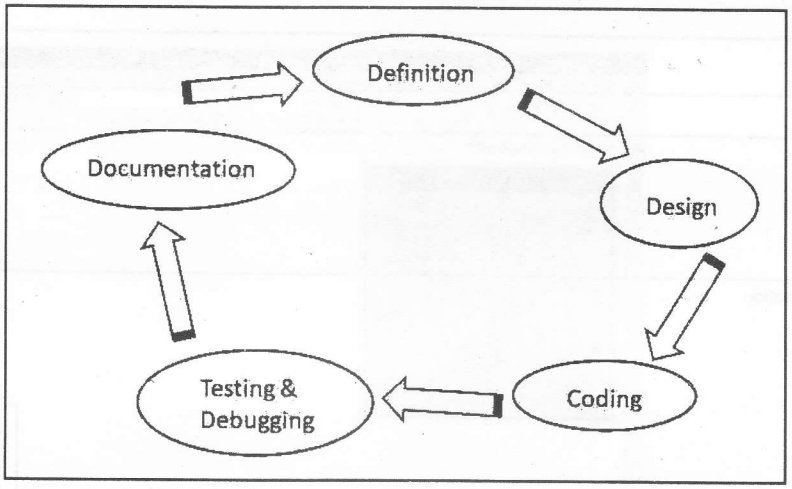
High level language brought respite to the programmers as it was easy to program, maintain and modify. The features of type of language were:

- Require Interpreter or compiler
- Interpreter translates high-level into machine code line-by-line while program is running
- Compiler translates entire program before program has run
- English like syntax
- Allow for object-oriented programming
- Programmer focuses on using objects to achieve the program's goal.

Visual Basic.Net is a strong, comfortable, easy to understand object oriented, event driven programming language. Microsoft Visual Basic.Net comes with Visual Studio.Net. Visual Studio.Net is available in an Academic Edition, a Professional Edition, an Enterprise Developer Edition and an Enterprise Architect Edition. The standard Edition of VB.Net can also be purchased separately by itself without any other .Net language.

1.3 DEVELOPING APPLICATIONS IN VB.NET

Application is wider term than a program. An application is a group of programs for some specific domain or task. Application development is complete process consisting of a number of steps, which combine together to make what is called the project life cycle as shown in the following figure:



1.3.1 Definition

As very rightly said problem is half solved if properly defined. It's a very significant step in application development as if the problem is stated with complete clarity then only it can have a proper solution.

1.3.2 Design

Designing an application is yet another important step in application development. Good time spent in designing can save you of problem that might occur at later stage and most importantly of debugging.

Design phase consists of creating detailed design description like designing user interface, planning properties of each object in the interface, technical definitions, functional definitions and final layouts.

1.3.3 Coding

Though half of the battle is won till now, yet coding is yet another milestone to be achieved properly. Writing code is not an easy task. It consists of defining user interfaces, setting the properties of each component of the interface and finally writing the basic code. Coding is done at three levels:

- *Class level* – Class modules contain class and data.
- *Event handling* – Event procedures or subprograms are written to capture specific events.
- *Standard code modules* – Global subprograms which can be included at any level.

1.3.4 Testing and Debugging

Once coding is complete, next process is testing to find out whether the code is working as desired or not and if not debug and fix the bugs. There are different methods of testing which are adopted. Testing is done a number of times not to miss any bugs. Generally testing should be done with "what if" method. Once errors are identified, cause of error is investigated and corrected.

Documentation

Documentation is needed at each phase of development and even during implementation in the form of manuals and online help. Other features like tool tips, "what is this?" helps can also be used. Illustrated document can be in printable form or an electronic ReadMe text file or an HTML PDF format file.

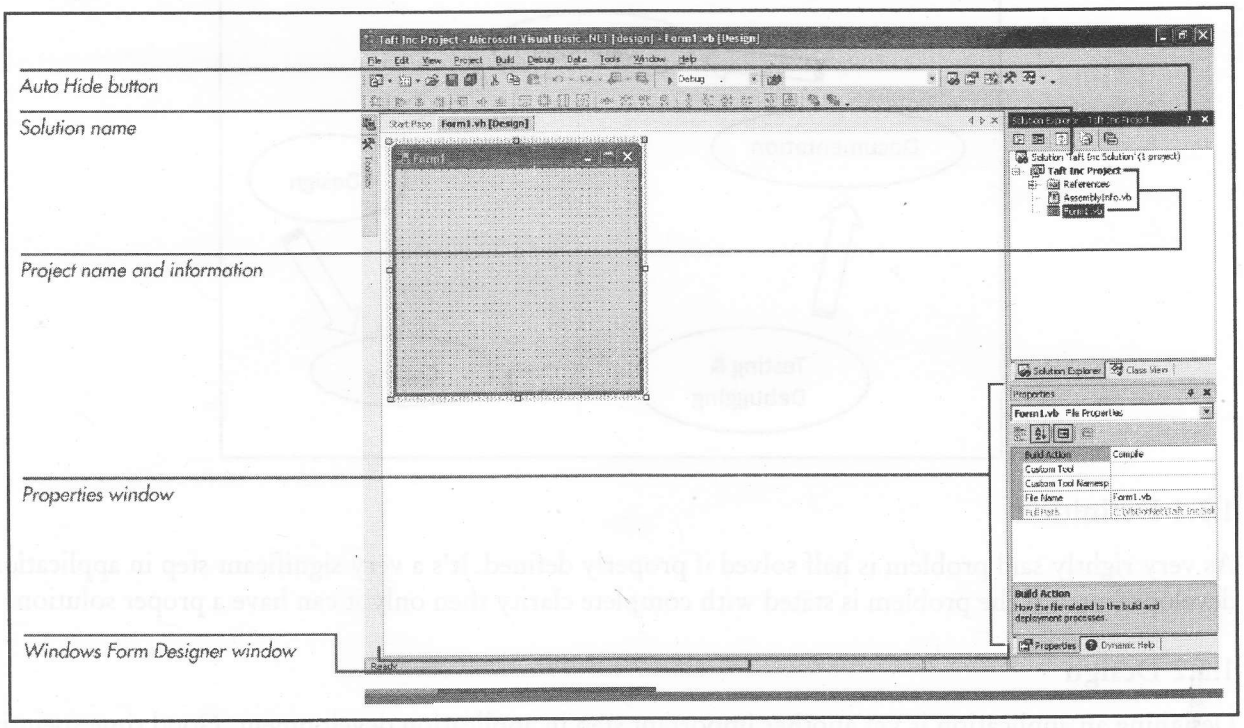


Figure 1.1: Solution and Project Created by Visual Studio.Net

1.4 VISUAL BASIC.NET USED FOR WRITING WINDOWS APPLICATIONS

Visual Basic.Net has a very strong user interface which makes it suitable to write user-friendly applications, that too in an easy and comfortable environment. Figure 1.2 below shows basic structure of Visual Basic .net solution applications. As shown in the figure a solution consists of a number of projects, which on the other hand consists of number of projects. Each project consists of a file.

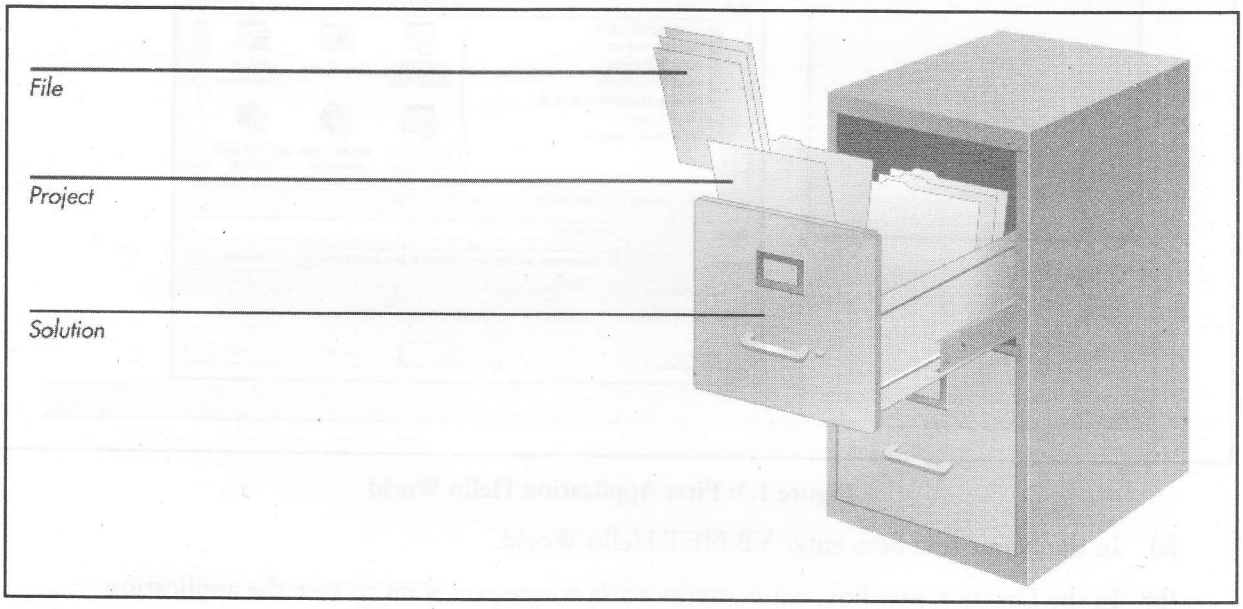


Figure 1.2: Visual Basic.Net Project Window

1.4.1 Lets Start Programming: A Sample Project

Now let's get introduced to Visual Basic .NET. The purpose of this section is not to teach you Visual Basic .NET, but to introduce you to it. You can create applications easily with Visual Studio .NET and application templates.

Lets create our first sample project. The following steps guide you through creating a Visual Basic .NET application:

1. Start Visual Studio .NET.
2. From the File menu, select New and then select Project.
3. In the New Project window, select the Visual Basic Projects folder.
4. Select the Windows Application project icon.
5. Before pressing OK, select the project name and location. Type the following values in the Name and Location text boxes (Figure 1.3):

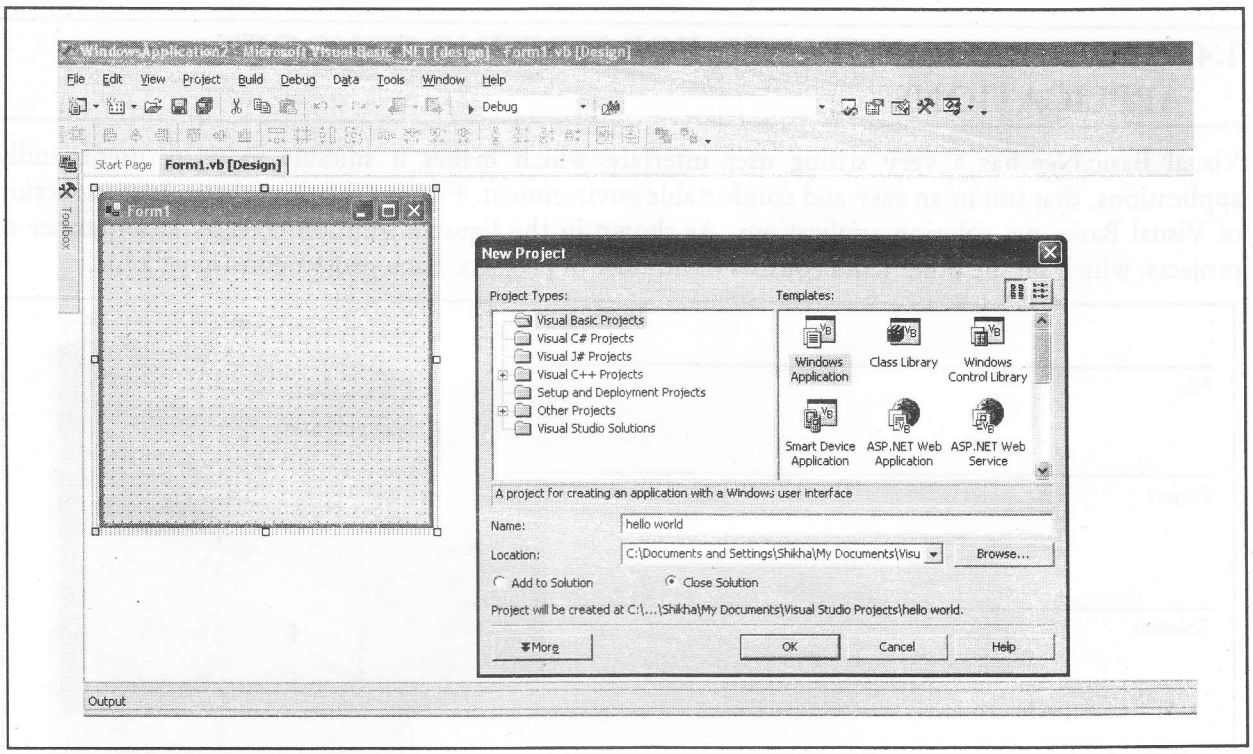


Figure 1.3: First Application Hello World

(a) In the Name text box, enter VB.NET Hello World.

(b) In the Location text box, enter any location where you want to save the application.

Creating the VB.NET "Hello World" sample.

6. Press OK. The Visual Basic .NET project, supporting files, and references are created.

Note that if you create a new project and use a directory with the project name after defining the location, you will end up with a directory structure that seems to duplicate the project name within it.

1.4.2 Adding "Hello World" Code

Now you'll add "Hello World" to the text area of the control bar:

1. Right-click on the Form1.vb object in the Solution Explorer window and select View Code.
2. You will first need to expand the "Windows Form Designer generated code" region. Under the comment, "'Add any initialization after the InitializeComponent() call" add the following code:

   Me.text = "Hello World"

   Your screen should look like Figure 1.4.

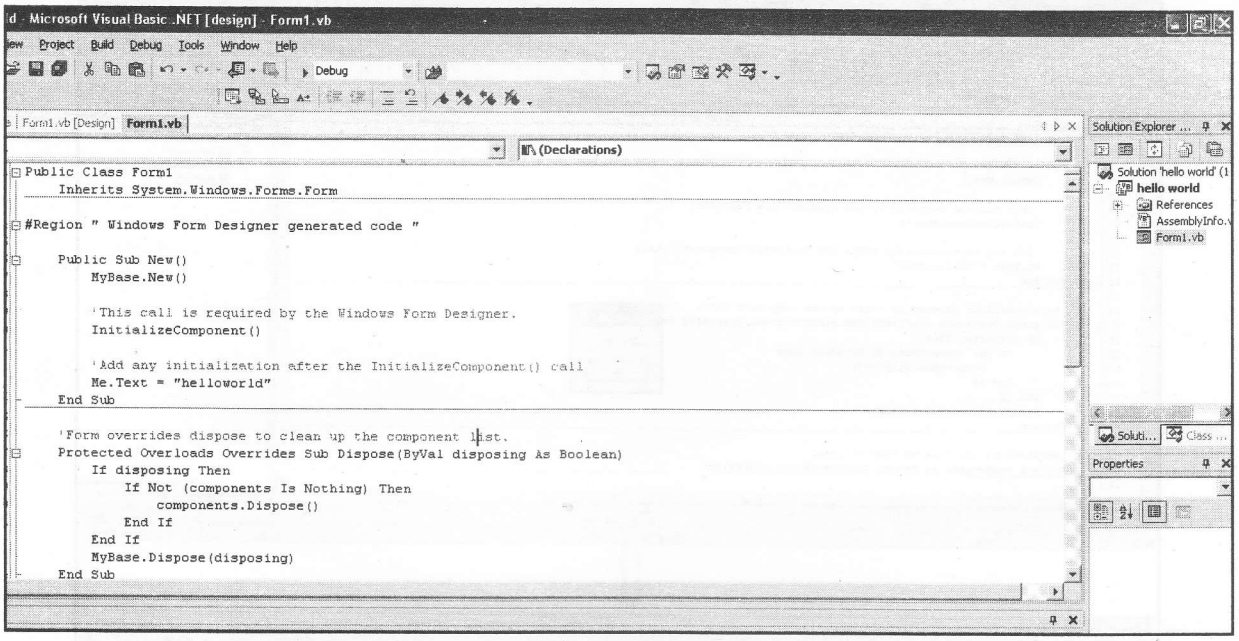


Figure 1.4: The Results of Adding the "Hello World" Code

3. Building the Project

To check for errors, build the Visual Basic project by selecting Build from the menu bar and then select Build again. When you run the build, an output screen appears at the bottom of Visual Studio. If there are no problems, you should see the output shown in Figure 1.5.

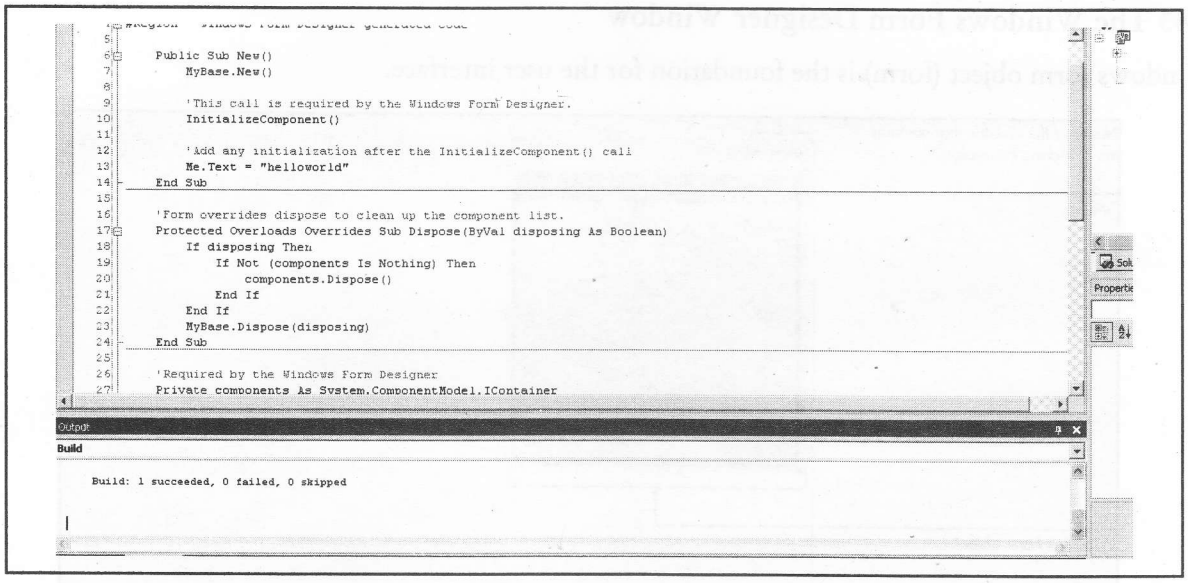


Figure 1.5: Compilation

4. Running the "Hello World" Application

To run your new windows application, select Debug from the menu bar and then select Run.

You will see the result shown in Figure 1.6.

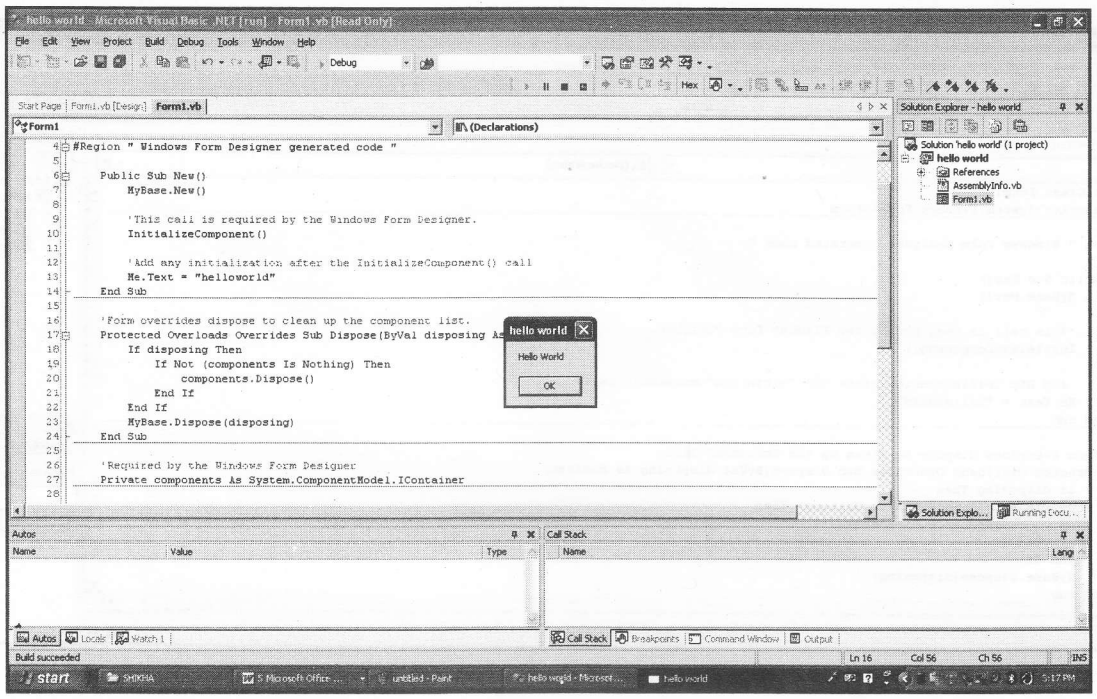


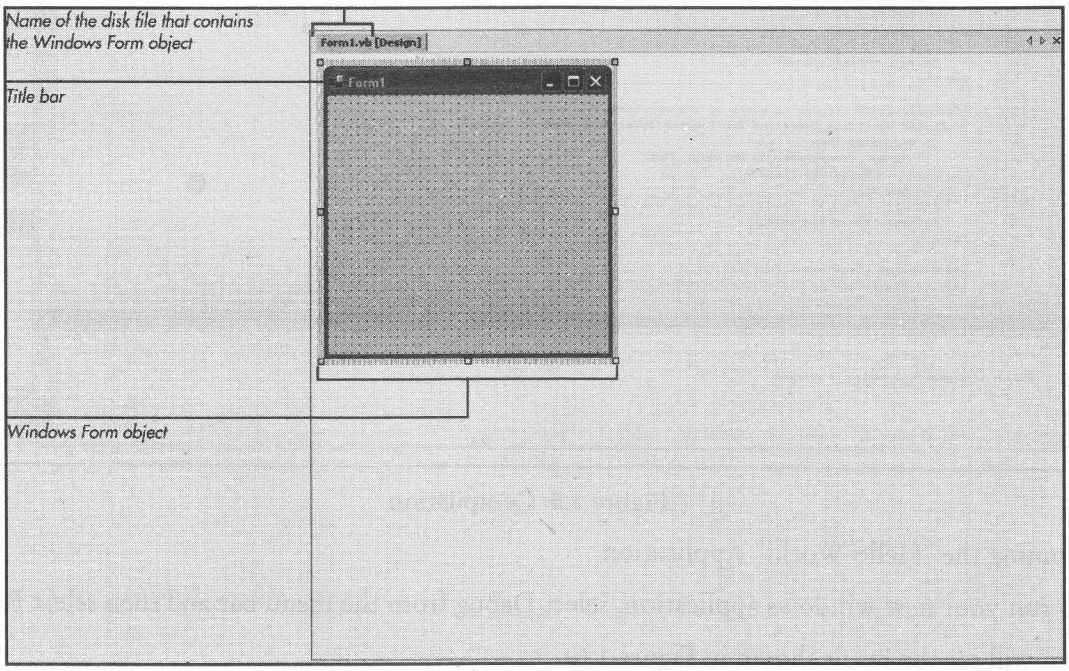
Figure 1.6: Successful Execution of Hello World

5. Press OK

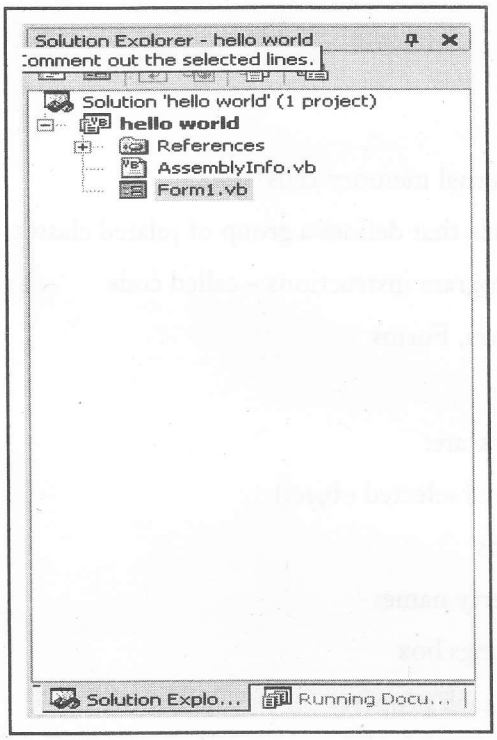
Congratulations! You have now created a .NET Windows Application.

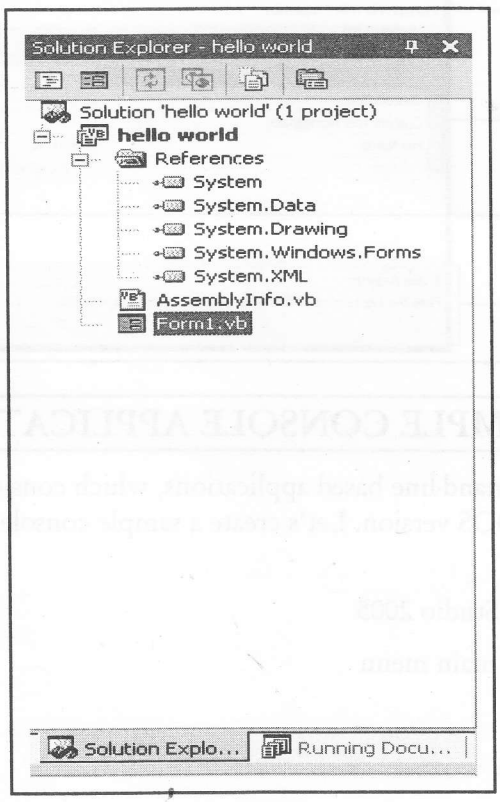
1.43 The Windows Form Designer Window

Windows form object (form) is the foundation for the user interface.



Solution Explorer Window





It contains:

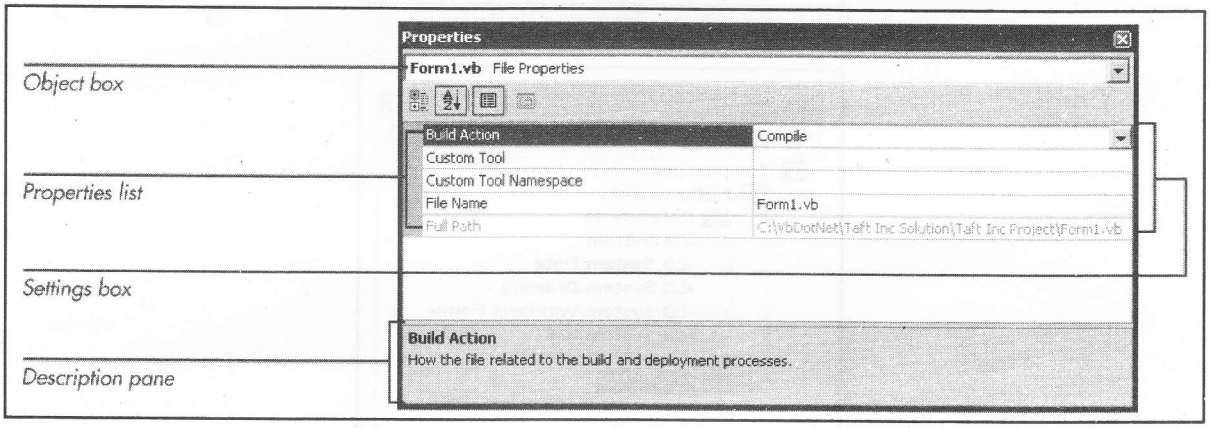
- Projects
- References
- Addresses of memory cells
- Namespace – block of internal memory cells
  Contains code that defines a group of related classes
- Source files – Contains program instructions – called code

Examples: Assembly information, Forms

The Properties Window

The components of this window are:

- Object box (contains name of selected object)
- Properties list has 2 columns
- Left column displays property names
- Right column displays Settings box
- Contains current value for each property



1.5 CREATING A SAMPLE CONSOLE APPLICATION IN VB.NET

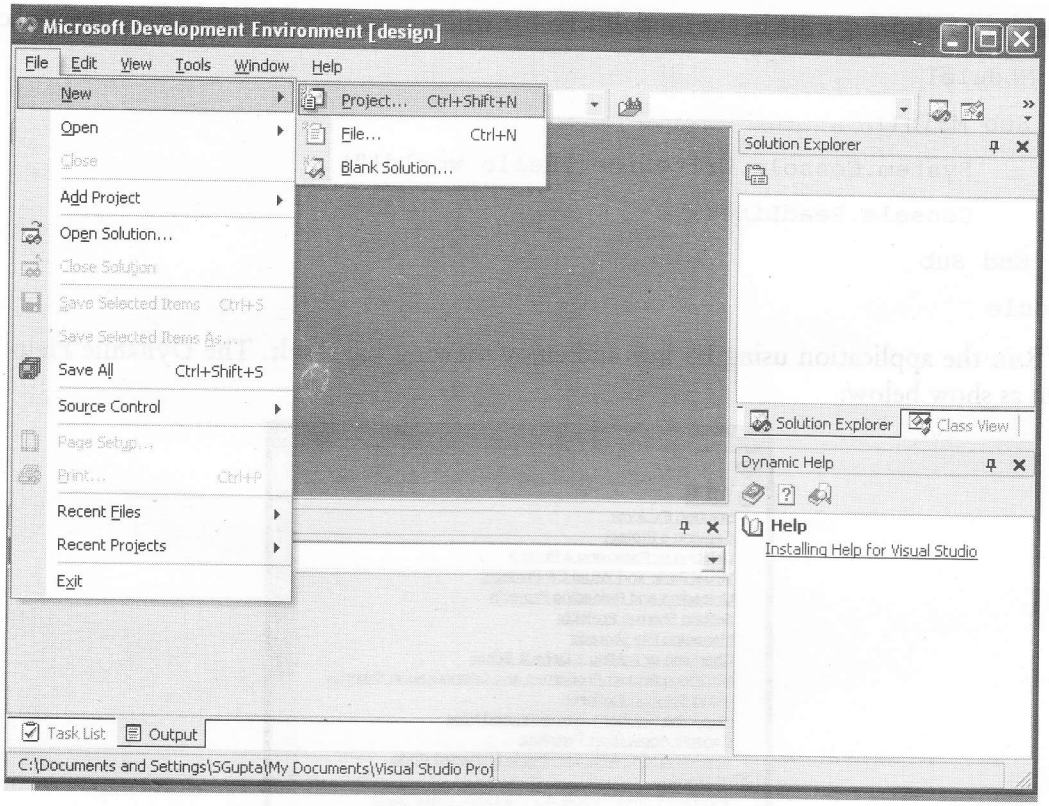
Console applications are command-line based applications, which consider the console as input/output medium and are executed in DOS version. Let's create a sample console application following the steps mentioned below:

Step 1: Open Microsoft Visual Studio 2005

Step 2: Click on File option of main menu

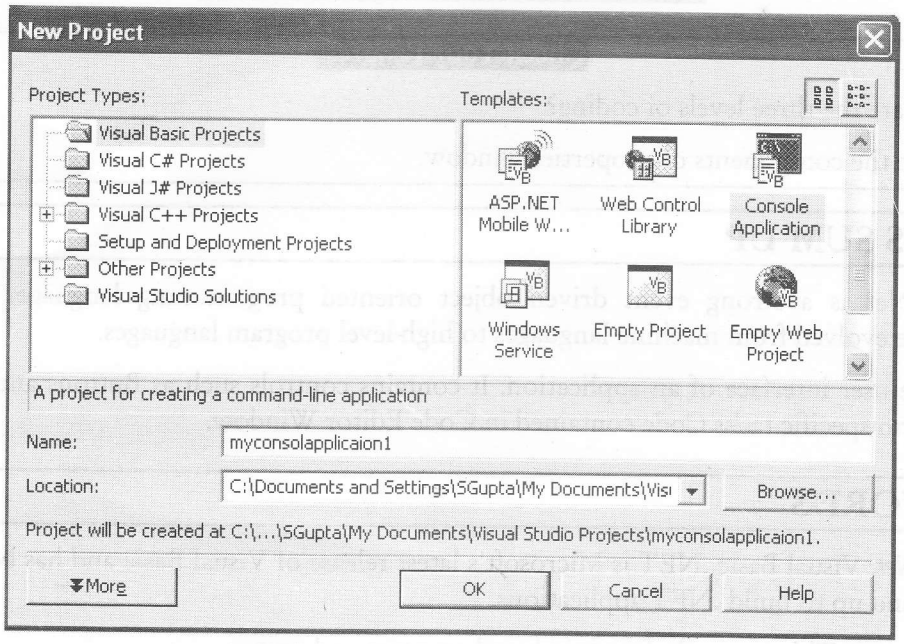
Step 3: Select New option

Step 4: Click on Project option



Step 5: Select Visual Basic from the dialog Box which appears

Step 6: Select Console Application from the Templates pane



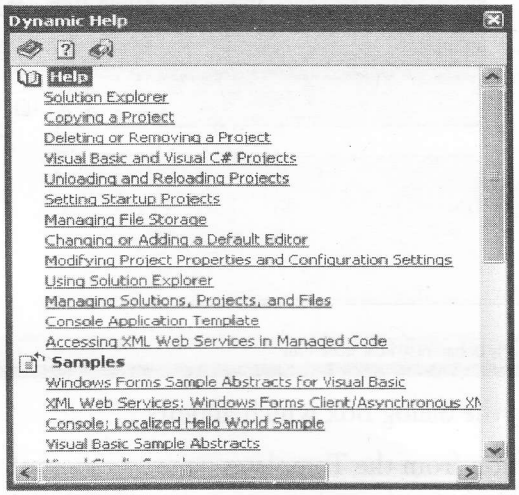
Step 7: Type the name of your application as myConsoleApplication1

Step 8: Click on OK button.

Step 9: Add the following code in the module1.vb file which is included by default in the application.

```
Module Module1
    Sub Main()
        System.Console.WriteLine("Hello world!")
        Console.ReadLine()
    End Sub
End Module
```

Step 10: Run the application using F5 key and enjoy viewing the result. The Dynamic Help Window will open as show below:



Check Your Progress

1. What are the three levels of coding?
2. Define the components of properties window.

1.6 LET US SUM UP

Visual Basic.Net is a strong event driven object oriented programming language. Programming languages have evolved from machine languages to high-level program languages.

Forms are the user interface of an application. It contains controls such as Buttons and Labels. Have code to perform specific tasks Code contained in Code Editor Window.

1.7 KEYWORDS

Visual Basic.Net: Visual Basic .NET is Microsoft's latest release of Visual Basic and has been redesigned from the ground up to build .NET applications.

Machine Language: This was early language based on zeros and ones.

Assembly Language: Assembly was next generation language after machine language based on symbols.

High Level Languages: High level language is English-like language for the programmers.

Documentation: Descriptive information about the application.

1.8 QUESTIONS FOR DISCUSSION

1. Explain the process of evolution of language.
2. Draw a diagram to depict project life cycle and explain each phase.
3. Explain the features of Visual Basic.Net.

Check Your Progress: Modal Answers

1. The three levels of coding are defined as below:
   (a) ***Class level:*** Class modules contain class and data.
   (b) ***Event handling:*** Event procedures or subprograms are written to capture specific events.
   (c) ***Standard code modules:*** Global subprograms which can be included at any level.
2. The components of this window are:
   (a) Object box (contains name of selected object)
   (b) Properties list has 2 columns
   (c) Left column displays property names
   (d) Right column displays Settings box
   (e) Contains current value for each property.

1.9 SUGGESTED READINGS

Sanjeev Sharma, *Visual Basic 6*, Excel Books

A. Chakraborti et al., *Microsoft.Net Framework*, PHI, 2002

M. Reynolds et al., *.Net Enterprise*, Wrox/SPD, 2002

Vikas Gupta, *.Net Programming*, Dreamtech Publication

Mackenzie Sharkey, *Teach yourself Visual Basic.Net in 21 days*, SAMS Techmedia

Bible, Bill Evjen, Jason Beres et al., *Visual Basic.Net Programming*, Wiley India

LESSON

2

OPERATORS, CONDITIONAL STATEMENTS AND LOOPS

CONTENTS

2.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Understand arithmetic and assignment operators
- Write an if...then...else statement
- Write code that uses comparison operators and logical operators
- Explain a nested selection structure
- Code an if/elseif/else selection structure
- Include a case selection structure
- Include the repetition structure
- Write a for...next statement
- Write a do...loop statement

2.1 INTRODUCTION

In this lesson, we will discuss the operators used in visual basic.net and how to program VB.Net code that can make decision when it process input from the users, and control the program flow in the process.

2.2 OPERATORS

Visual Basic.Net comes with many built-in operators that allow us to manipulate data. An operator performs a function on one or more operands. For example, we add two variables with the "+" addition operator and store the result in a third variable with the "=" assignment operator.

For Example, Lets suppose we have declared 3 variables myint1, myint2 and myint3 as integers. I have to add the values of myint1 (20) myint2 (40). The sample code will be written as below:

```
myint1 = 20
myint2 = 40
myint3 = myint1 + myint2
```

The two variables above are called operands operated by two operators "+" and "=".

2.2.1 Arithmetic Operators

Visual Basic .NET provides a basic set of operators to perform simple arithmetic.

- + Addition
- - Subtraction
- * Multiplication
- / Division
- \ Integer division
- Mod Remainder Division

- ^ Exponentiation
- & String concatenation

Operation	Result
10 + 3	13
10 - 3	7
10 * 3	30
10 / 3	3.3
10 \ 3	3
10 Mod 3	1
10 ^ 3	1000
"10" & "3"	103

Example code:

```
Option Strict On
 Imports System
 Module Module1

 Sub Main( )

 Dim myintvar1 As Integer = 10
 Dim myintvar2 As Integer = 3

 Dim myinvar3 as integer

 Myintvar3 = myintvar1^myintvar2

 Console.WriteLine("result is ", myintvar3)
 End Sub
 End Module
```

Division Operators

There are more types of division than the one denoted by the "/" symbol. There is also integer division and remainder division.

This is the most commonly used form of division and is denoted by the "/" operator.

```
Option Strict On
 Imports System
 Module Module1

 Sub Main( )

 Dim myintvar1 As Integer = 10
 Dim myintvar2 As Integer = 3
 Dim myinvar3 as single
' ( we must use the Single class to have decimals)
```

Contd...

```
Myintvar3 = myintvar1 / myintvar2
Console.WriteLine("result is ", myintvar3)

End Sub
End Module
```

Integer Division: This divides two numbers, and gives the result without the remainder if the quotient is a decimal. Example:

```
Option Strict On
 Imports System
 Module Module1

 Sub Main( )

 Dim myintvar1 As Integer = 10
 Dim myintvar2 As Integer = 3
 Dim myinvar3 as single
' ( we must use the Single class to have decimals)

 Myintvar3 = myintvar1 / myintvar2
 Console.WriteLine("result is ", myintvar3)

 End Sub

 End Module
```

Remainder Division: This divides two numbers, and gives the result's remainder if the quotient is a decimal. This is denoted by the operator "Mod." Example:

```
Option Strict On
 Imports System
 Module Module1

 Sub Main( )

 Dim myintvar1 As Integer = 10
 Dim myintvar2 As Integer = 3
 Dim myinvar3 as single
' ( we must use the Single class to have decimals)

 Myintvar3 = myintvar1 / myintvar2
 Console.WriteLine("result is ", myintvar3)

 End Sub

 End Module
```

2.2.2 Operator Precedence

The order in which operations are performed determines the result. Consider the expression 3+4*2. What is the result? If addition is done first, the result is 14. However, if the multiplication is done first, the result is 11. The hierarchy of operations or order of precedence, in arithmetic expression from highest to lowest is:

1. Any operation inside parentheses
2. Exponentiation
3. Multiplication and division
4. Integer division
5. Modulus
6. Addition and subtraction.

In the previous example, multiplication is performed as the addition, yielding a result of 11. To change the order of evaluation, use parentheses. The expression (3+4)*2 will yield 14 as a result. One set of parentheses perhaps be used inside another set. In that case the parentheses are said to be nested.

Multiple operations at the same level like multiplication and division or addition and subtraction are performed left to right. For example 16+3-7 produce 12 as its result. First operation is performed is 16+3 which produces 19 and then 19-7 is performed which gives 12.

2.2.3 Comparison and Logical Operators

The Conditional Operators allow you to refine what you are testing for. Instead of saying "If X is equal to Y", you can specify whether it's greater than, less than, and a whole lot more.

Table 8.4: Conditional and Logical Operators

Operator	Meaning

>	This symbol means Is Greater Than
	Example:
	If mynumber1 > mynumber2 Then
	Console.WriteLine("mynumber1 is greater")
	End If

<	This symbol means Is Less Than
	Example:
	If mynumber1 < mynumber2 Then
	Console.WriteLine("mynumber1 is greater")
	End If

> =	This symbol means Is Greater Than or Equal To
	Example:
	If mynumber1 > = mynumber2 Then
	Console.WriteLine("mynumber1 is greater or equal")
	End If

< =	This symbol means Is Less Than or Equal To
	Example:
	If mynumber1 < = mynumber2 Then
	Console.WriteLine("mynumber1 is greater or equal")
	End If

Contd...

And	You can combine the logical operators with the word And.
	Example:
	If number > 5 And number < 15 Then
	MsgBox "Greater than 5 And Less than 15"
	End If

Or	You can combine the logical operators with the word Or.
	Example:
	If number > 5 Or number < 15 Then
	MsgBox "Greater than 5 Or Less than 15"
	End If

< >	This symbols mean Is Not Equal to, and are used
	Example:
	If number1 < > number2 Then
	MsgBox "number1 is not equal to number2"
	End If

2.2.4 Assignment Operator

Statement of the form variable = variable operator expression has assignment operator Visual Basic.NET provides several assignment operators for abbreviating assignment statements. For example, the statement

value = value + 3

can be abbreviated with the addition assignment operator (+ =) as

value + = 3

The + = operator adds the value of the right operand to the value of the left operand and stores the result in the left operand's variable. Any statement of the form

variable = variable operator expression

where operator is one of the binary operators +, -, *, ^, &, / or \, can be written in the form

variable operator= expression

There is a huge list of combined assignment operator symbols. All the symbols =, + =, -=, *=, /=, \=, ^= and &= are operators, though they are not included in operator-precedence tables.

```
Module Module1

 Sub Main()
 Dim MYINTVAR as Integer
 MYINTVAR= 0
 MYINTVAR+= 10
 Console.WriteLine("MYINTVAR+= 10 yields " & MYINTVAR)
 MYINTVAR-= 5
 Console.WriteLine("MYINTVAR-=5 yields " & MYINTVAR)
 MYINTVAR*= 3
 Console.WriteLine("MYINTVAR*= 3 yields " & MYINTVAR)
```

Contd...

```
MYINTVAR/= 5
Console.WriteLine("MYINTVAR/= 5 yields " & MYINTVAR)
MYINTVAR^= 2
Console.WriteLine("MYINTVAR= 2 yields " & MYINTVAR)

End Sub

End Module
```

2.3 CONDITIONAL STATEMENTS

2.3.1 If - Then

Life is full of choices (isn't it so nice??). We look around and what do we observe? We are constantly making decisions every day. We leave home and if it is raining, we take our raincoats along. Our mother might say, go to the market and buy 5 kg potatoes if potatoes cost less than 10 rupees a kilo.

In programming terms, we can say this: 'if' it is raining, 'then' put on the raincoat. Similarly, 'if' potatoes are < Rs. 10/kg, 'then' buy 5 kgs. The underlying concept is this: 'if' a condition is 'true' 'then' we will take some action. The same approach can also be applied to programming. We can specify a condition to VB. 'If' the condition given by us is true, 'then' all of the statements following the 'if' will be executed, otherwise VB will jump over those statements. For example, the following lines of code

Code Listing if.1

```
IF text1.TEXT = "Shreyas" THEN
MSGBOX "Hello" & "Shreyas"
END IF
```

will display a message box with the message "Hello Shreyas" printed on it, 'if' Text1 contains the text 'Shreyas'. If Text1 does not contain the text Shreyas, the message box will NOT be displayed. Take another example. Suppose there are two integer variables, named int_a and int_b. Analyze the code segment that follows.

Code Listing if.2

```
IF int_a > int_b THEN
Text2.TEXT = int_a
END IF
```

The value of variable int_a will be printed in Text2, only if the value of variable int_a is greater than variable int_b. Otherwise, Text 2 will stay as it is.

General Syntax of If-Then

This brings us to the general syntax of the If - Then statement, which is as follows:

```
If condition Then
......
'execute all of these statements if the condition is true
......
```

```
......
'if condition is false none of these statements will be executed
......
End If
```

2.3.2 If - Then - Else

In our discussion of if-then, we have so far considered only one option. That is,

if int_a > int_b, then print int_a

But what if we want to print int_b if 'int_a > int_b' is not true? The simple if-then does not offer another option, i.e., it does something if the condition is true, but does nothing if the condition is false. This extra option is offered by if-then-else, a modification of the simple if-then we have just seen. To see this in action, we will modify the Code List if.1 as follows:

Code Listing if.3

```
IF text1.Text = "Shreyas" THEN
MSGBOX "Hello" & "Shreyas"
ELSE
MSGBOX "Hello" & "Mr/Miss No Name"
END IF
```

In this code, VB will check for the contents of Text1. If the text entered in "Text1" is "Shreyas", the message box "Hello Shreyas" will be displayed. If the text entered is not "Shreyas" but anything else, the message box "Hello Mr/Miss No Name" will be displayed. Modifying the Code List if.2,

Code Listing if.4

```
IF int_a > int_b THEN
text1.TEXT = int_a
ELSE
text1.TEXT = int_b
END IF
```

This will display value of the variable int_a in Text1 if the condition [int_a > int_b] is true. Otherwise, it will display value of the variable int_b in Text1.

General Syntax and Working of If-Then-Else

This code segment shown below will work this way, depending on whether the condition following 'if' is true or false:

- *Condition is true:* All of the statements following 'then' will be executed, until VB comes across the keyword 'else'. Once 'else' is reached, VB will 'jump over' all of the statements between 'else' and 'end if' and come out of the 'if - then - else' block. It will then continue the normal program execution after the 'end if' statement.
- *Condition is false:* None of the statements following 'then' will be executed. However, all of the statements following 'else' will be executed, until the time VB comes across the keywords 'end if'.

After 'end if' is read, the 'if - then - else' block will terminate. VB will then continue the normal execution of the program after the 'end if' statement.

The general syntax of if-then-else statement is as follows:

```
If condition Then
......
......
'execute all of these statements
'starting from "then"
'ending at "else"
'if the condition is true -------> a
......
Else ----------------------
......
......
'execute all of these statements
'starting from "else"
'ending at "end if"
'if the condition is false
......
End If ---------> b
```

2.3.3 Multiple (Nested) If - Then Statements

The code segment we have used in discussing if - then - else works fine if one of the numbers is greater than the other one. But what if both the numbers are equal? Suppose we gave the value of

int_a = 10

and

int_b = 10

what will be the output of Code List if.4? The result will be ..... you guessed it right, 10! The Code List if.4 checks for [int_a > int_b] , i.e. 10 > 10. Since this condition is not true, i.e. false, the statements following 'else' will be executed, which print the value of int_b which, in the present case, is 10. As is obvious, if...then...else is most effective when there are only two ways to go. In the Code Listing if.4, the possible outcomes are -

int_a > int_b or int_a < int_b or int_a = int_b

As we can clearly see now, a simple if..then..else will be ineffective in the case just mentioned above, since, this situation has three possible outcomes. We therefore need three blocks to effectively deal with the three different outcomes. Hence, we need another block in addition to the 'then' and 'else'

block that 'if..then..else' offers. We can get this option by adding another if..then block 'inside' the existing if..then..else block., like this.

Code Listing if.5

```
IF int_a > int_b THEN
text1.TEXT = int_a     'printed when a > b --- 1
ELSE
IF int_a < int_b THEN  '--- 2
text1.TEXT = B     'printed when a < b --- 3
ELSE
text1.TEXT = "both are equal"     '--- 4
END IF
END IF
```

Observe from Code Listing if.5 that for every if, a corresponding end if has to be used. The 'inner' if..then is represented by block 'I', the 'outer' if..then by 'O'. The code runs as follows:

- This code will first check for [int_a > int_b]. If the condition is true, the program goes to line number '1', prints the value of 'int_a' in Text1, and skips over all of the other lines. It then comes out of the if-then blocks.
- If int_a > int_b is false, the program goes to line '2' and checks for [int_a < int_b]. If this condition is true, the program goes to line '3' and prints the value of 'int_b' in Text1. It then comes out of the if-then blocks.
- However, if this condition [int_a < int_b] is also false, then it is obvious that both the numbers are equal. Hence, the program switches to line number 4 and prints the message 'both are equal' in Text1.

It is important to realize the fact that the inner block 'I' must be completely inside the outer block 'O'. If this is not done, we will get wrong results.

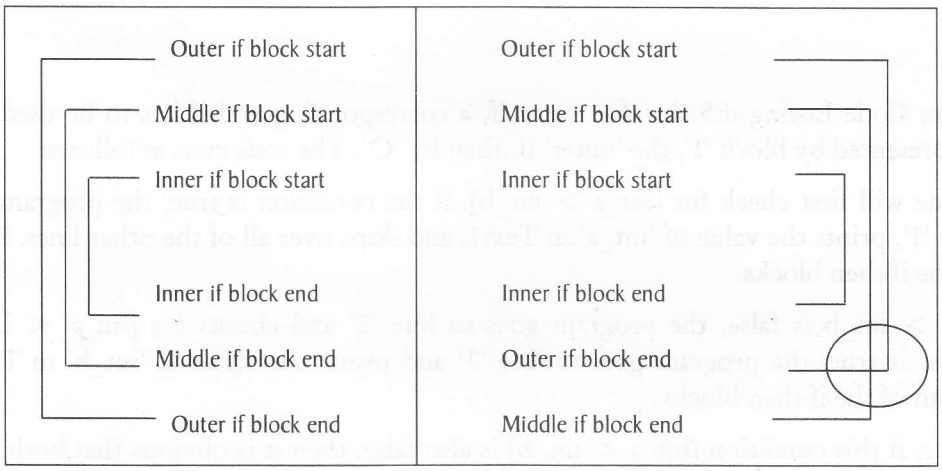
General Syntax of Multiple If-Then-Else

The general syntax for multiple if-then-else statements is as follows:

```
IF  condition1 THEN
    ......
    ......
    'execute all of these statements, ending at "else"
......
ELSE
IF condition2 THEN
    'execute all of these lines, ending at "else"
    'if the condition2 is true
    ......
```

```
ELSE
    ......
    'execute all of these statements
    'ending at "end if"
    'if the condition2 is false
END IF 'inner if-then-else block ends
END IF 'outer if-then-else block ends
```

As explained above, the inner block must be completely contained by the outer block. In well-designed nested if-then blocks, if we draw lines showing the boundaries of the various blocks, it would look like the left-side figure below:



As seen in the figure above, all of the block lines are complete and none of the block lines are intersecting one another. However, in the figure on the right, the outer and middle block lines are intersecting towards the bottom-right of the figure. In this situation, we get run-time errors that are difficult to handle.

2.3.4 If - Then - Else if Statement

If - then performs fine, but as the levels of nesting increase, the number of *if* keywords and the associated *end if* keywords also increases. While the increase in if..*end if* keywords is not a problem for the computer, it is a bit tough for us as programmers. We have to keep a close track of the number of times we have used *if* so that we close with an equal number of *end if* statements. VB offers a useful feature to overcome this problem. Instead of adopting this approach where we have three '*end - if*' commands (since we have three '*if*' commands too), we can combine the two statements of block 'a' and also of block 'b' into a single statement, like this

```
If condition1 Then
...
Elseif condition2 Then 'Else and If of block "a" combined
...
Elseif condition3 Then 'Else and If of block "b" combined
```

```
...
Else
...
End If
```

The advantages of this approach are obvious. It makes our code look more neat. We can more easily follow the flow of logic in the program. At the same time, we have the benefit of typing 'end..if' only once, even though we have checked for multiple conditions in the code. This saves time spent on typing.

Having come this far, we will now modify our earlier Code Listing if.5 as follows, using the Elseif keyword:

Code Listing if.6

```
IF int_a > int_b THEN '- - - - I
text1.TEXT = int_a    'printed when int_a > int_b
ELSEIF int_a < int_b THEN '- - - - II
text1.TEXT = int_b    'printed when int_a < int_b
ELSE
text1.TEXT = "BOTH ARE EQUAL"    'printed when both numbers are equal
END IF
```

You can see in the code that we have used 'If' twice (on lines marked as I and II), but 'End If' has been used only once.

Let us make another piece of code. We will accept the name of a colour in a textbox (Txt_col) and then click on a command button (Cmd_col) to change the background colour of the VB form as follows:

- If the user enters 'green', we change the colour to green.
- If the user enters 'red' we change the colour to red.
- If there is any other value, we change the colour to blue.

Code Listing if.7

```
PRIVATE SUB cmd_col_CLICK( )
IF txt_col.TEXT = "green" THEN
form1.BACKCOLOR = VBGREEN
ELSEIF txt_col.TEXT = "red" THEN
form1.BACKCOLOR = VBRED
ELSE
form1.BACKCOLOR = VBBLUE
END IF
END SUB
```

2.3.5 Select..Case Statement

Apart from If..then statements, another programming construct performs the task of conditional branching as good - maybe even better. This is the select..case statement. We'll now re-make the Code listing if.3 using the same logic, but with select..case. We will write the code as follows:

Code Listing select.1

```
1.  SELECT CASE text1.TEXT
2.  CASE = "Shreyas"    'IF text1 IS "Shreyas"
3.  MSGBOX "Hello" & "Shreyas"
4.  CASE ELSE  'IF text1 IS NOT "Shreyas"
5.  MSGBOX "Hello Mr/Miss No Name"
6.  END SELECT
```

This code starts by reading the value of text1.Text, as per the line no.1. It then checks the value of text1, matching it with "Shreyas", as per line no. 2. The program will now take two different routes, depending on whether [Text1 = "Shreyas"] or not:

- If the value matches "Shreyas", line no.3 is executed, displaying a message box with the given message. The program then skips over line nos. 4 and 5, and comes to line no. 6. This line terminates the select..case block. Thus, line no. 3 acts as the then option of if..then..else.
- If the value does NOT match "Shreyas", line no.3 is skipped and program continues at line no. 4. This line acts as the else of if..then..else. Line no. 5 tells what to do when the else part - case else - is to be executed.

Select..case, thus, offers a choice of multiple branching, depending on which condition happens to be true. The condition is checked whenever the 'Case = ..' statement is encountered. If the condition is found to be true, the program executes all of the statements following the current 'Case = ..' upto the next 'Case' statement.

General Syntax of Select-Case

A general syntax for the Select - Case statement, thus, is as follows:

```
Select Case variable name/control name
Case = value1  'check for condition1
....
....     execute these statements
....     if condition1 is true
....
Case = value2  'check for condition2
....
....     execute these statements
....     if condition2 is true
....
```

```
Case = value3 'check for condition3
....
....   execute these statements
....   if condition3 is true
....
Case Else
....
....   execute these statements when
....   all other conditions are false
....
End Select
```

We will now use Select..Case to modify our Code Listing if.7 we have made earlier.

Code Listing select.2

```
1.  SELECT CASE var1   ' read the value of the variable var1
2.  CASE = "green" ' if var1 is green
3.  form1.BACKCOLOR = VBGREEN
4.  MSGBOX "Green Colour"
5.  CASE = "red"   ' if var1 is red
6.  form1.BACKCOLOR = VBRED
7.  MSGBOX "Red Colour"
8.  CASE ELSE  ' if var1 is neither green nor red
9.  form1.BACKCOLOR = VBBLUE
10. MSGBOX "DEFAULT COLOUR Blue"
11. END SELECT
```

This code selects the value of a variable 'var1' at line 1 [Select Case var1] and then checks it at the following line numbers:

- line 2 [Case = "green"]
- line 5 [Case = "red"] and
- line 8 [Case Else]

The program then executes one of the three paths depending on the value of var1:

- *'Var1' is "green":* This value becomes true at line no. 2 resulting in execution of line nos. 3 and 4 only, i.e. all lines between [Case = "green"] and [Case = "red"]. The form's backcolor changes to green. An appropriate message box is also displayed. The program then skips over all lines from line no. 5 up to line no. 10. Line no. 11 ends the Select..Case block.
- *Var1' is "red":* This value becomes true at line no. 5 resulting in execution of line nos. 6 and 7 only, i.e., all lines between [Case = "red"] and [Case Else]. The form's backcolor changes to red.

An appropriate message box is also displayed. The program then skips over all lines from line no. 8 up to line no. 10. Line no. 11 ends the Select..Case block.

- *'Var1' is NOT "red" or "green":* This value becomes true at line no. 8 resulting in execution of line nos. 9 and 10 only, i.e., all the lines between [Case Else] and [End Select]. The form's backcolor changes to blue. An appropriate message box is also displayed. The program then encounters Line no. 11, which ends the Select..Case block.

2.3.6 Nested Select-case

Similar to the concept of nesting we have seen in if..then..else, Select..Case can also be nested. An example will make this clear to us. Suppose we have a program which accepts a student's category and status. The program checks the category and then mentions the government grant for the students based on the following criteria:

- General Category - No grant
- OBC category - Rs. 1,500
- SC/ST category - Rs. 2,500 if status = 'bpl'
- SC/ST category - Rs. 2,000 if status = 'ric'

Code Listing select.3

```
1.  SELECT CASE txt_categ.TEXT
2.  CASE IS = "GEN"
3.  MSGBOX "Sorry, no grant for you."    'if the code is gen
4.  CASE IS = "OBC"
5.  MSGBOX "Your government grant is Rs. 1500." 'if the code is obc
6.  CASE IS = "SC", "ST"  'if case is either sc or st
7.  SELECT CASE txt_status.TEXT
8.  CASE IS = "bpl"
9.  MSGBOX "Your grant is Rs. 2500."     'if the values are sc/stand bpl
10. CASE IS = "ric"
11. MSGBOX " Your grant is Rs. 2000."    'if the values are sc/st and ric
12. END SELECT
13. CASE ELSE
14. MSGBOX "Sorry, the category code is invalid"
15. END SELECT
```

Here, the code runs as follows:

1. The line nos. from 7 to 12 mark the 'nested block'.
2. Line no. 1 selects the text in Txt_categ text box.
3. Line no. 2 checks if the text is 'gen'. If true, line no. 3 is executed, and the remaining lines are all skipped.

4. Line no. 4 checks if the text is 'obc'. If true, line no. 5 is executed, and the remaining lines are all skipped.
5. The line no. 6 checks if the text is 'sc' or 'st'. If true, the 'nested' Select-Case block is started.
   (a) Line no. 7 selects the text in Txt_status textbox.
   (b) Line no. 8 checks if 'text' in Txt_status is 'bpl'. If true, line no. 9 is executed. Remaining lines of inner and outer block are skipped.
   (c) Line no. 10 checks if 'text' in Txt_status is 'ric'. If true, line no. 11 is executed, and the remaining lines are all skipped.
   (d) Line no. 12 terminates the inner select...case block.
6. Line no. 13 is reached if the user has entered a wrong category, i.e., the condition at line no. 2, 4, and 6 is 'false'.

   Case = 'po'

   and press enter. VB will automatically convert this line to

   Case Is = 'po'

   This way, we will type less and avoid any possible spelling errors.

2.3.7 Select-case With Multiple Options (OR)

We will see another aspect of the 'Select-Case' statement, if we observe the line no. 6 above, i.e.

CASE IS = 'sc','st'

this line checks for two possible values of Txt_Categ, 'sc' or 'st'. This line is just like saying

IF txt_categ.TEXT = 'sc' or txt_categ.TEXT = 'st' THEN

Thus, if we have to check for one out of two or more values, we just type (Case Is =) and then put all the values one after the other, separated by a comma. For example, the code list given below checks for city names given in a text box (Txt_city) and then prints the state to which the city belongs.

Code List select.4

```
1.  SELECT CASE txt_city.TEXT
2.  CASE IS = "BHOPAL" , "ujjain", "gwalior"
3.  MSGBOX "M.P. STATE"
4.  CASE IS = "MUMBAI" , "thane" , "nasik"
5.  MSGBOX "MAHARASHTRA STATE"
6.  CASE IS = "LUCKNOW" , "ALLAHABAD" , "MEERUT"
7.  MSGBOX "U.P. STATE"
8.  CASE ELSE
9.  MSGBOX "The state is NOT known"
10. END SELECT
```

Here, the code runs as follows:

1. Line no. 1 selects the text in txt_city textbox.
2. Line no. 2 checks text of txt_city. If it is either 'bhopal' or 'ujjain' or 'gwalior', line no. 3 is executed. The other lines are by-passed.
3. Line no. 4 checks the text of txt_city. If the value is 'mumbai' or 'thane' or 'nasik', line no. 5 is executed, by-passing all other lines of code.
4. Line no. 6 checks text of txt_city. If it is 'lucknow' or 'allahabad' or 'meerut', line no. 7 is executed. Remaining code lines are by-passed.
5. Line no. 8 is reached only if the line nos. 2, 4, and 6 don't match the value of txt_city, and then line no. 9 is executed.
6. Line no. 9 prints an error message for the user.
7. Line no. 10, of course, terminates the Select..Case block.

2.3.8 Select-case with Multiple Options (AND)

We will now see another aspect of the 'Select-Case' statement. Consider the situation where we are preparing a marksheet and we need to calculate the division. *As is expected, the criteria will be:*

- 80% upto 100% - honours
- 60% upto 79.9% - 1st division
- 45% upto 59.9% - 2nd division
- 33% upto 44.9% - 3rd division
- any other value - fail

The code for the problem will be as follows:

Code List select.5

```
1.  SELECT CASE CINT(txt_perc.TEXT)  'convert into nearest integer value
2.  CASE 80 to 100 'if perc is between 80 and 100
3.  MSGBOX "HONOURS DIVISION"
4.  CASE 60 TO 79  'if perc is between 60 and 79
5.  MSGBOX "1ST DIVISION"
6.  CASE 45 TO 59  'if perc is between 45 and 59
7.  MSGBOX "2ND DIVISION"
8.  CASE 33 TO 44  'if perc is between 33 and 44
9.  MSGBOX "3RD DIVISION"
10. CASE ELSE  'if perc is not in any range mentioned above
11. MSGBOX "FAIL"
12. END SELECT
```

Here, the code runs as follows:

1. Line no. 1 selects the text in txt_perc textbox and converts it to the nearest integer value (e.g. 79.6 to 80, 79.4 to 79, and so on.).
2. Line no. 2 checks text of txt_perc. If between 80 and 100, line no. 3 is executed. The other lines are by-passed.
3. Line no. 4 checks text of txt_perc. If between 60 and 79, line no. 5 is executed, by-passing all other lines of code.
4. Line no. 6 checks text of txt_perc. If between 45 and 59, line no. 7 is executed. Remaining code lines are by-passed.
5. Line no. 8 checks text of txt_perc. If between 33 and 44, line no. 9 is executed. Other lines are by-passed.
6. Line no. 10 is reached if all other criteria of 'percentage' are not fulfilled.
7. Line no. 11 prints a messagebox with the message 'Fail'.
8. Line no. 12 terminates the Select..Case block.

This is a situation similar to the "and" in "if..then..else", i.e,

- If the percentage is between 80 and 100, then give the message "honours".
- If the percentage is between 60 and 79, then give the message "1st division", and so on.

2.4 LOOPS (ITERATIVE STATEMENTS)

At times, we might need to execute a group of statements repeatedly. That is, we want to execute them over and over again. To consider some simple examples:

- Preparing the results of an exam: we start from the first student and continue with the same set of rules upto the last student.
- Preparation of bills in a telephone company: we take the first customer and continue with the billing process upto the last customer in our list.
- Preparing a bill in a shop: we start processing the order from the first product to the last.
- On the popular TV game show KBC: questions are repeatedly asked till the time the participant opts out or gives a wrong answer.

As is obvious, we are repeating a group of statements, applying similar conditions and rules at every stage. This repeated execution of statements can be achieved through what we call as 'looping' or 'iterative' statements. We will now see all of these statements.

2.4.1 For-Next Loop

The general syntax of the For-Next Loop is as follows:

```
FOR i = initial value TO final value'line 1
    'All of these statements
    'will be repeated as long
```

```
    'as the numeric value
    'of the variable 'i' is
    'between the initial value
    'and the final value
NEXT i  'line 8
```

where i can be any integer variable. In For-Next terminology, 'i' is known as an index variable. The code starts by initializing the value of i, as given in the line [For i = initial value to final value]. All of the statements upto line no. 8 (Next i) are then executed.

At line number 8, two important things happen:

1. The value of i is incremented by 1.
2. "Next i" then checks if the value of i falls in the given range on line 1 (initial value and final value)

If the incremented value of i falls within the range, all the lines between 1 and 8 are executed once again. Once again, the program comes to line 8. Once again the process happening at line number 8 is repeated, as we have just discussed.

If the incremented value falls between the range, all of the lines between 1 and 8 are again executed. This process continues till the time the value of i is between the initial and the final value. That is, the lines between 1 and 8 are repeatedly executed. This is what we call as the 'loop'.

Once incremented value of 'i' does NOT fall in the given range, execution of lines between 1 and 8 is stopped. Thus, the For-Next loop is terminated. The program will then start processing statements written after "Next i" statement.

Let us say we want to print a sequence of numbers from 1 to 3. Our program will look like this:

Code Listing for.1

```
1.  FOR i = 1 TO 3 'define initial and final values
2.  PRINT i
3.  NEXT i  'check if loop is to be repeated
4.  MSGBOX " For - Next loop has ended"
```

Here, the code runs as follows:

1. Line no. 1 starts the loop, assigning initial of 1 for i and a final value of 3.
2. Line no. 2 prints the value of i on the screen, i.e., 1.
3. Line no. 3 "Next i" increments the value of 'i' by 1: 'i' becomes 2.
4. The line "Next i" checks if 'i' falls in the range 1 to 3 as specified in line no. 1. The current value of 'i', i.e., 2, is between 1 and 3. Hence, line no. 2 is repeated.
5. Line no. 2 again prints the value of 'i' on the screen, i.e., 2.
6. Line no. 3 is reached again. "Next i" again increments 'i' by 1. So, 'i' now becomes 3. The current value is then checked at line no.1. Since the current value of i is between 1 and 3, line no. 2 is repeated.

7. Line no. 2 again prints the value of 'i' on the screen, i.e., 3.
8. Line no. 3 is reached again. "Next i" again increments 'i' by 1. So, 'i' now becomes 4. The current value of 'i' is 4, but 4 is NOT between 1 and 3. Thus, line no. 2 is not repeated. The For..Next loop is terminated, and further printing of the value of 'i' is stopped. Now the program comes to line 4.
9. Line no. 4 gives a message box indicating that the loop has terminated. The program stops.

Take another example. This time we will print a name on the screen four times. The code segment is as follows:

Code Listing for.2

```
1. FOR i = 1 TO 4 'define initial and final values
2. PRINT "Mamta Puri"
3. NEXT i 'check if loop is to be repeated
4. MSGBOX "Name printing task is over"
```

Here, the code runs as follows:

1. Line no. 1 starts the for loop. It assigns an initial of 1 for the index variable i and a final value of 4.
2. Line no. 2 prints 'Mamta Puri' on the screen (1st time).
3. Line no. 3 "Next i" increments the value of 'i' by 1: 'i' becomes 2. The line "Next i"
4. Then checks if 'i' falls in the range 1 to 4 as given in line no. 1. The current value of i, i.e., 2, is between 1 and hence, line no. 2 is repeated.
5. Line no. 2 again prints 'Mamta Puri' on the screen (2nd time).
6. Line no. 3 is reached again. "Next i" again increments 'i' by 1. So, 'i' now becomes 3. The current value is then checked at line no.1. Since the current value of 'i' is between 1 and 4, line no. 2 is repeated.
7. Line no. 2 again prints 'Mamta Puri' on the screen (3rd time).
8. Line no. 3 is reached again. "Next i" again increments 'i' by 1. So, 'i' now becomes 4. The current value of 'i' is 4 and 4 is between 1 and 4. Thus, line no. 2 is repeated.
9. Line no. 2 again prints 'Mamta Puri' on the screen (4th time).
10. Line no. 3 is reached again. "Next i" again increments 'i' by 1. So, 'i' now becomes 5. The current value of 'i' is 5, but 5 is NOT between 1 and 4. Thus, line no. 2 is not repeated. The loop is terminated. The program shifts to line no. 4.
11. Line no. 4 gives a message box on the screen. The program then stops.

2.4.2 For-Next with Step

So far, we have always increased the value of the index variable by 1. But what if we want to increase it by 2, or by 5? What if we want to decrease it by 1, or by 5? This option is provided by the Step keyword. The general syntax of For-Next now becomes

```
FOR i = initial value TO final value     STEP j
'everytime the statement
   'NEXT I is reached
   'i will change by
   'the value given in
   'STEP j
NEXT i
```

where 'j' is an integer variable, either positive or negative. If we want to decrease the value of " i " at every step, 'j' should be a negative integer.

Let us now use Step to make a program that generates the multiples of 2 upto the number 10, i.e., the output should be: 2,4,6,8,10.

Code Listing for.3

```
1. FOR i = 2 TO 10 STEP 2    'increase i by 2 at every iteration
2. PRINT i
3. NEXT i 'check if loop is to be repeated
4. MSGBOX "loop is over"
```

Here, the code runs as follows:

1. Line no. 1 gives 'i' the initial value of 2.
2. Line no. 2 prints '2' on the screen.
3. Line no. 3 increases the value of 'i' by 2, because the value of Step is 2. This changes the value of i to 4. Line no. 3 then checks if the current value of i, i.e., 4 is between 2 and 10. Since it is, the program shifts to line no. 2 and prints the value of i.
4. As long as the value of i is between 2 and 10, the program will keep shifting to line no.2 from line no. 3. Once 'i' comes out of the range (2 to 10) the for-next loop will terminate.
5. Now line no. 4 will run, informing the user that the loop has terminated.

Thus, after giving the output as a sequence, i.e.,

2 4 6 8 10

the loop terminates.

We can make the current code do the reverse thing - printing even numbers from 10 to 2, in descending order. The output expected, thus, becomes - 10,8,6,4,2. The code is going to be as follows:

Code Listing for.4

```
1. FOR i = 10 TO 2 STEP -2  ' decrease i by 2 at every iteration
   'compare line above with line no. 1 of Code Listing for.3
2. PRINT i
```

```
3.  NEXT i 'check if loop is to be repeated
4.  MSGBOX "loop is over"
```

Here, the code runs as follows:

1. Line no. 1 gives 'i' the initial value of 10.
2. Line no. 2 prints '10' on the screen.
3. Line no. 3 decreases the value of 'i' by 2, because the value of Step is [-2]. This changes the value of 'i' to 8. Line no. 3 then checks if the current value of 'i' [8] is between 10 and 2. Since it is, the program shifts to line no. 2 and prints the value of 'i'.

As long as the value of 'i' is between 10 and 2, the program will keep shifting to line no.2

from line no. 3. Once 'i' comes out of the range [10 to 2] the for-next loop will terminate. Then line no. 4 will be executed.

If we compare Code Listing for.3 and Code Listing for.4, we will notice that only line 1 has been changed. Thus, after giving the output

10 8 6 4 2

the loop terminates.

Here is another program, which accepts five names, one by one, and displays them on a message box. The code is like this:

Code Listing for.5

```
1.  DIM nam AS STRING
2.  FOR i = 1 TO 5 'define initial and final values
3.  nam = INPUTBOX ("Please enter a name")
4.  MSGBOX nam & "is a nice name indeed"
5.  NEXT i 'check if loop is to be repeated
6.  MSGBOX "loop is over"
```

Here, the code runs as follows:

1. Line no. 1 declares a String type of variable 'nam' at line 1.
2. Line no. 2 begins the for-next loop, initializing 'i' to the value 1.
3. Line no. 3 accepts a name and stores it in the variable 'nam'.
4. Line no. 4 displays this name in a message box alongwith a message that should please any user!
5. Line no. 5 increases 'i' by 1 and then checks if 'i' falls between 1 and 5. Since the present value of 'i' does fall in the range, line nos. 3 and 4 are re-executed, again asking for a name and displaying it in a message box.

As long as the value of 'i' stays between 1 and 5, the For..Next loop will continue to accept a name and display it in a message box. Once the value of 'i' goes beyond 5, the loop is ended and line no. 6 is executed.

2.4.4 Nested Loops

For..Next loops can be 'nested', with the same rules for nesting as for others: no crossing-over of the inner and outer loops. Thus, the following loop is valid as it does not have any intersection of the outer and inner For..Next loop.

```
FOR i = 1 TO 5
FOR j = 1 TO 5
    'some processing
    'done here
  NEXT j
NEXT i
```

Whereas, the following loop is invalid, since two loops intersect:

```
  FOR i = 1 TO 5
    FOR j = 1 TO 5
      'some processing
      'done here
    NEXT i
  NEXT j
```

To minimize risk of wrong nesting, just type 'Next' instead of 'Next i', 'Next j', etc. This way, VB will follow a proper nesting by itself.

```
  FOR i = 1 TO 4
    FOR j = 1 TO 4
      'some processing
      'done here
    NEXT
  NEXT
```

By the way, can you guess how many times the inner 'j' loop will run? A total of 16 times! This is because of the following reason:

1. *'i' is 1:* While 'i' stays at 1, 'j' will change in value from 1 to 4. Thus, the 'j' loop will run a total of 4 times when [i=1]. The variable 'i' does not change in value at present because the 'i' loop's [Next I] statement is not reached in this duration. (The program keeps looping between 'For j-Next j' statements.) Once 'j' becomes 5, the 'j' loop ends and the program comes to [Next I] statement. Now, the value of 'i' becomes 2 and the control shifts back to the [For j] statement.
2. *'i' is 2:* The 'j' loop will again change in value from 1 to 4. Thus, the 'j' loop will run a total of 4 times while [i=2]. The variable 'i' does not change in value at present because the 'i' loop's [Next I] statement is not reached in this duration. (The program keeps looping between 'For j-Next j' statements.) Once 'j' becomes 5, the 'j' loop ends and the program comes to [Next I] statement. Now, the value of 'i' becomes 3 and the control shifts back to the [For j] statement.

3. *'i' is 3:* The 'j' loop will again change in value from 1 to 4. The 'j' loop will run a total of four times while [i=3]. Once 'j' becomes 5, the 'j' loop ends and the program comes to [Next I] statement. Now, the value of 'i' becomes 4 and the control shifts back to the [For j] statement.
4. *'i' is 4:* The 'j' loop will again change in value from 1 to 4. The 'j' loop will run a total of four times while [i=4]. Once 'j' becomes 5, the 'j' loop ends and the program comes to [Next I] statement. Now, the value of 'i' becomes 5. Since the value of 'i' is no longer between the 'i' loop condition [1 to 4], the 'i' loop is terminated. The program comes out of the 'i' loop and stops after [Next I].

As just explained, for every single value of i, j changes value from 1 to 4. Thus, the j loop runs '4' times for every single run of the i loop. The i loop will change from 1 to 4 and then terminate. This means that the i loop will run a total of '4' times. Thus, the program will run a total of 4 * 4 = 16 times. (No. of iterations of 'i' loop * No. of iterations of 'j' loop for each single value of 'i').

2.4.5 Do Loops

Apart from the For-Next loop, VB provides the Do loop as another iterative mechanism. Needless to say, the need of this loop is the same as that of the For-Next loop. This loop is, broadly speaking, of two types:

- Do-While-Loop
- Do-Until-Loop

In both the types, we provide a condition after the keyword "Do-While" or "Do-Until". The repetition of the loop depends on whether the condition is true or false at any given point of time.

2.4.6 Do-While-Loop

In this loop, the loop lines are repeated as long as the given condition is true. Once the condition becomes false, the loop is terminated and the program continues with the statements that appear after the Do-While loop.

The general syntax for the Do-While loop is

```
DO  WHILE  some condition
    'execute some
    'statements repeatedly
    'while the condition
    'is true
LOOP
```

All of the statements written between 'Do-While' and 'Loop' are repeatedly executed till the time the condition mentioned after Do-While is true. That is, the 'Loop' statement shifts the control back to 'Do-While' and the condition is then checked. If the condition is true, the loop is re-executed. Once the condition becomes false, the program stops the repeated loop execution and begins to execute the statements after 'Loop'.

As an example, we will remake our Code Listing for.1 using Do-While along with a command button's click event:

Code Listing do.1

```
1.  i = 1
2.  DO WHILE i < 4 'check if i is less than 4
3.  PRINT i
4.  i = i + 1
5.  LOOP    'check if loop is to be repeated
6.  MSGBOX " Do - While loop over"
```

Here, the code runs as follows:

1. Line no. 1 assigns the initial value for the variable 'i'.
2. Line no. 2 will check the condition [i < 4]. If the condition is true, the program will execute all lines upto line no. 5.
3. Line no. 3 will print the value of 'i' [1].
4. Line no. 4 increases the value of 'i' by 1: 'i' now becomes 2.
5. Line no. 5 shifts the control back to line 2, where the condition [i < 4] will be checked again. With the current value of 'i' [2], this condition is true. Thus, line no. 3 is executed, printing the value of 'i' [2] on the screen. Line no. 4 again increases value of 'i' by 1. So, 'i' is now 3. Line no. 5 shifts the control back to line 1, where the condition [i < 4] will be checked again. As long as the value of 'i' is less than 4, the program will keep on printing and incrementing the value of 'i' and checking whether the loop should be continued or not.

Once 'i' becomes equal to 4, the loop is terminated. The program shifts to line no. 6, displaying a message box on the screen and then stops.

Take another example. As done in Code Listing for.2, we will print a name on the screen four times. This time we'll use Do-While to get the same result, using a command button's Click event:

Code Listing do.2

```
1.  i = 1
2.  DO WHILE i < 5 'check if i is less than 5
3.  PRINT "Taranya Gupta"
4.  i = i + 1
5.  LOOP 'check if loop is to be repeated
6.  MSGBOX "Name printing is over"
```

Here, the code runs as follows:

1. Line no. 1 assigns a value of 1 to the variable 'i'.
2. Line no. 2 compares the value of 'i' with 5. If it is less than 5, the remaining lines are executed. Otherwise, the loop is not executed.

3. Line no. 3 prints 'Taranya Gupta' on the screen.
4. Line no. 4 increases 'i' by 1.
5. Line 5 takes the control back to line 2 and here the condition [i < 5] is checked again. If it is true, line nos 3 and 4 are repeatedly executed. Once the condition [i < 5] becomes false, further execution of the loop is stopped. The program now shifts to line no. 6.
6. Line no. 6 displays a message box on the screen. The program stops.

We'll now get the same results as from Code Listing for.3 seen earlier in the chapter. This time, though, we'll be using Do-While. Again, use a command button's Click as the event for the code. The code is as follows:

Code Listing do.3

```
1. i = 2
2. DO WHILE i < 11    'check if i is less than 11
3. PRINT i
4. i = i + 2
5. LOOP    'check if loop is to be repeated
6. MSGBOX "do while ended"
```

Here, the code runs as follows:

1. Line no. 1 assigns the number 2 to the variable 'i'.
2. Line no. 2 checks if [i < 11]. If it is true, line 3 is executed. If it is not true, all the loop lines are skipped over, and program control shifts to line no. 6.
3. Line no. 3 prints the value of 'i' on the screen.
4. Line no. 4 increases the value of 'i' by 2.
5. Line no. 5 takes the program back to line 2, where the condition [i < 11] is again checked. If the condition is true, line nos. 3 and 4 are re-executed. This means that the program continues with the repetition of the loop as long as the condition [i < 11] is true. Once the condition [i < 11] becomes false, the loop is terminated, and control shifts to the line following line 5.
6. Line no. 6 displays a message box and the program ends.

Thus, after giving the output

2 4 6 8 10

the loop terminates.

Now we will generate the same output in reverse order, i.e., 10,8,6,4,2, with the following code:

Code Listing do.4

```
1. i = 10
2. DO WHILE i > 1 'check if i is greater than 1
3. PRINT i
4. i = i - 2
```

```
5.  LOOP    'check if loop is to be repeated
6.  MSGBOX "loop over"
```

Here, the code runs as follows:

1. Line no. 1 assigns 10 to 'i'.
2. Line 2 checks the condition [i > 1]. Since i is initially 10, this condition is true. Thus, line nos. 3 and 4 are run. After line 4, 'i' decreases by 2, i.e., 'i' drops to 8.
3. Line no. 5 takes the control back to line 2 and again checks for the condition [i > 1]. Since 'i' is still greater than 1, lines 3 and 4 are again executed. This process of repetition continues while 'i' is greater than 1. Once 'i' becomes 0, the condition [i > 1] becomes false, and the repetition of the loop is stopped. The program then continues with any statements written after line 5 (LOOP).
4. Line no. 6 terminates the program after displaying a messagebox.

The output produced by Code List do.4 is:

10 8 6 4 2

2.4.7 Do-Until Loop

With the Do-While loop, the repetitions continue as far as the given condition is "true". That is, the loop will continue "while" the condition is "true". The Do-Until loop is just the reverse of this. In a Do-Until loop, the repetitions take place as far as the given condition is "false". In other words, the loop will continue "until" the condition becomes "true". Once the condition becomes true, the loop is terminated and the program continues with the statements mentioned after Do-Until loop. The general syntax for the Do-Until loop is:

```
DO UNTIL some condition
    'execute some
    'statements repeatedly
    'until the condition
    'becomes true
LOOP
```

All of the statements written between Do-Until and Loop are repeatedly executed as far as the condition mentioned after Do-Until is not true. That is, the Loop statement shifts the control back to Do-until and the condition is then checked. If the condition is not true, the loop is re-executed. Once the condition becomes true, the program stops the repeated loop execution and begins to execute the statements after Loop.

As an example, we will remake our Code Listing do.1 using Do-Until, to get the same result as from code listing do.1 using a button's click event:

Code Listing do.5

```
1.  DO UNTIL i > 4 'continue if i is NOT greater than 4
2.  PRINT i
```

```
3.  i = i + 1
4.  LOOP    'check if loop is to be repeated
5.  MSGBOX " Do - Until loop over"
```

Here, the code runs as follows:

1. Line no 1 will check the condition [i > 4]. If the condition is not true, the program will execute all lines upto line no. 4. If the condition is true, the loop will not be executed.
2. Line no. 2 will print the value of 'i'.
3. Line no. 3 will increase the value of 'i' by 1, which now becomes 2.
4. Line no. 4 will shift control back to line 1, where the condition [i > 4] will be evaluated again. With the current value of 'i' [2], this condition is still not true. Thus, line no. 2 is re-executed, printing the value of 'i' [2] on the screen. Line no. 3 increases value of 'i' by 1 again. This process of repeating the statements (from line no. 1 upto line no. 4) will continue until [i > 4]. Once 'i' becomes greater than 4, the loop will terminate.
5. Line no. 5 displays a message box and the program terminates.

Take another example. This time we will print a name on the screen four times, exactly as we have done in Code Listing for.2 and Code Listing do.2. Now we will use Do-UNTIL to get the same result.

Code Listing do.6

```
1.  i = 1
2.  DO UNTIL i > 4 'continue if i is NOT greater than 5
3.  PRINT "Sam"
4.  i = i + 1
5.  LOOP    'check if loop is to be repeated
6.  MSGBOX "Name printing is over"
```

Here, the code runs as follows:

1. Line no. 1 initializes 'i' to 1.
2. Line no. 2 checks if the current value of 'i' is greater than 5. As the condition is not true, line no. 3 is executed.
3. Line no. 4 now increases 'i' by 1.
4. Line 5 takes the control back to line 2 and here the condition [i > 5] is checked again. If it is not true, the process of repeating the loop continues. Once the condition [i > 5] becomes true, further execution of the loop is stopped. The program now shifts to line no. 6.
5. Line no. 6 displays a message box on the screen and the program stops.

Now we will generate a series of numbers - 2,4,6,8,10. You might recollect that we have already done this using For-Next and Do-While. Now we will be using Do-Until to get the same result. Use the click event of a button for the code:

Code Listing do.7

```
1. i = 2
2. DO UNTIL i > 11    'continue loop if i is NOT greater than 11
3. PRINT i
4. i = i + 2
5. LOOP    'check if loop is to be repeated
```

Here, the code runs as follows:

1. At line no. 1 'i' is assigned value of 2.
2. Line 2 checks whether [i > 11]. Since it is not, Line 3 is executed, printing the value of 'i' on the screen.
3. Line 4 increases the value of 'i' by 2.
4. Line 5 takes the program back to line 2, where the condition [i > 11] is again evaluated. If the condition is true, line nos. 2 and 3 are re-executed. This means that the program continues with the repetition of the loop as long as the condition [i > 11] is not true. Once the condition becomes true, the loop is terminated, and control shifts to the line following line 5.

Thus, after giving the output:

2 4 6 8 10

the loop terminates.

2.4.8 Do-Loop Variants

When we start with Do-While or Do-Until, the condition is checked 'before' the loop begins execution. If the condition is true, the loop will be executed at least once. If the condition is false right from the start, the loop will not at all be executed. There might be cases where we want the loop to be executed at least once. Thus, in such a situation we need to check for the condition at the end of the loop, after at least one execution of the loop. For these situations, we can alter the Do-While and Do-Until statements as follows:

```
DO
'some statements meant
'for repeated execution
LOOP WHILE condition
Or
DO
'some statements meant
'for repeated execution
LOOP UNTIL condition
```

As we can see, the Do-While-Loop and Do-Until-Loop statements have been changed to Do-Loop-While and Do-Loop-Until, respectively. This is the only change we need. The logic of running the

loops remains what we have seen so far. The only change is that the condition is checked after the first execution of the loop has been done. This means that if the condition is false from the very start, the Do/Loop-While and Do/Loop-Until will be executed at least once. This is because the condition is checked at the 'end' of the loop. By this time, the loop has already run at least once. In a similar situation, the Do-While/Loop and Do-Until/Loop will never be executed, since the condition will be checked at the start of the loop.

Exit Command

All of the iterative or looping statements seen so far continue to repeat, depending on the condition. There might be occasions where we want to terminate a loop prematurely, i.e., we don't want the loop to run the full distance. For example, we might come to have a situation in which the loop stops immediately, if the variable has reached a particular value. Further processing of this variable might possibly lead to some errors. In a situation such as this, we would like our loop to stop all further iterations, even if the range of the variable is proper. The 'exit' keyword helps us in such a situation, and offers the facility to prematurely come out of the loop.

For example, take a hypothetical situation. We might have a program in which we are doing some database processing. Once the value of the field 'department' becomes 'marketing', we want to terminate the loop. (this is just a hypothetical situation, I repeat). This can be achieved as follows:

Code Listing exit.1

```
FOR i = 1 TO 8
'do some
'processing here
IF department = "marketing" THEN
EXIT FOR   'end the loop immediately
END IF
'continue processing
NEXT i
```

The code given here works as follows:

- Normally, the For-Next loop above will repeat 8 times.
- The If-Then will check for the value of department at every run of the loop. If the value becomes 'marketing', the loop will be terminated and the lines following NEXT i will now be executed.
- The loop will be terminated even if the For-Next has been run only 6 times. This is because the EXIT FOR statement instructs VB to end the loop prematurely and to come out of it at once.

Check Your Progress

1. What is If-then-else statement? Give example.
2. What is Do-While loop? Give example.

2.5 LET US SUM UP

Three programming structures are sequence, selection, and repetition. Selection structure allows program to make a decision and then select one of 2 paths depending upon the results of that decision. Visual Basic .NET has four selection structures: If, If/Else, If/ElseIf/Else, Case. Selection structures can be "nested" within other selection structures: Outer structure contains primary decision Inner structure contains secondary decision.

- All expressions in a selection structure evaluate to either true or false.
- Repetition (loop) allows program to repeatedly process one or more program instructions.
- Pretest loops test before program instructions.
- Posttest loops test after program instructions and will always be processed at least once.
- All variables in an array have the same name and data type.

2.6 KEYWORDS

Operator: An operator performs a function on one or more operands.

For Loops: For loop is used to process instructions a precise number of times.

For Next Loop: The for Next Loop executes the loop body (the source code within For ...Next code block) to a fixed number of times.

Erase Statement: The Erase statement is used to release memory assigned to array variables.

Repetition Structure: Repetition Structure repeatedly processes a set of instructions until a condition is met.

ReDim Statement: You use the ReDim statement to specify or change the size of one or more dimensions of an array that has already bean declared.

2.7 QUESTIONS FOR DISCUSSION

1. Explain the order of precedence of operators for calculations.
2. Define the following terms:
   (i) Loop
   (ii) Subscript
3. What is a condition?
4. In what situation would a loop be used in a procedure?
5. Explain the difference between the Do Loop and For Next Loop.
6. What are the steps in processing a For Next Loop?

Check Your Progress: Modal Answers

1. If then else statement is defined as a different set of code statements depending on the given condition which is either true or false.

   Example: Let us consider the case where student with marks above 40 is pass and student with marks less than 40 is fail.

```
Private Sub Button1_Click( ByVal sender As System.Object,
 ByVal e As System.EventArgs Handles Button1.Clic
If Val(Textbox1.Text) >=40 Then
 MsgBox("PASS")
    Else
 MsgBox("FAIL")
End If
End Sub
```

2. In this loop, the loop lines are repeated as long as the given condition is true. Once the condition becomes false, the loop is terminated and the program continues with the statements that appear after the Do-While loop.

   Example:

```
Private Sub Button1_Click( ByVal sender As System.Object,
ByVal e As System.EventArgs Handles MyBase.Load
Dim a As Integer
a = 2
Do While a<100
a = a * 2
MsgBox("Product is::" & a)
Loop
End Sub
```

   In the example above, the loop is continued on the arrival of false value according to the given condition, and the loop exits.

2.8 SUGGESTED READINGS

Sanjeev Sharma, *Visual Basic 6*, Excel Books

A. Chakraborti et al., *Microsoft.Net Framework*, PHI, 2002

M. Reynolds et al., *.Net Enterprise*, Wrox/SPD, 2002

Vikas Gupta, *.Net Programming*, Dreamtech Publication

Mackenzie Sharkey, *Teach yourself Visual Basic.Net in 21 days*, SAMS Techmedia

Bible, Bill Evjen, Jason Beres et al., *Visual Basic.Net Programming*, Wiley India

LESSON 3

PROCEDURE, SCOPE, EXCEPTION HANDLING

CONTENTS

3.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Differentiate between a sub-procedure and a function procedure
- Create a sub-procedure
- Explain a procedure that receives information passed to it
- Differentiate between passing data by value and passing data by reference
- Explain function procedure
- Use Try/Catch blocks for error handling
- Define design time, run time, and run time
- Identify syntax errors, run-time errors and logical errors
- Explain the reasons of the various errors
- Know the importance of handling exceptions
- Differentiate between errors and exceptions

3.1 INTRODUCTION

A procedure is a block of program code that performs a specific task. For example, the code for the click event of a Command Button command1 is written within the stub: There are a number of advantages of using procedures; the most important ones are namely: The programmer can put reusable code in a block called procedure without writing the same code a number of times. Procedures are useful in breaking down complex and very big tasks into smaller ones that perform a specific task. In VB, code cannot be written anywhere but in the above four procedures. The advantage of using procedures is that they make applications more readable, since code that is oft repeated, is positioned within them. It would not be wrong to say that without procedures, no application can be written in VB.

When you design and develop a project, errors are bound to occur. Visual Basic.Net also supports structured exception handling, which enables you to identify and remove errors at run time. In Visual Basic.Net, you can create robust and effective exception handlers to improve the performance of your application.

3.2 PROCEDURE

The main logic behind having procedures is to write the code once, and then, call it several times from different places. By writing procedures, the program can be broken down into smaller logical units. It is always easier to debug smaller procedures or blocks or modules, rather than debugging one large program. This approach is referred to as modular approach. By breaking down complex and big tasks into smaller ones, you simplify maintenance required in future.

It is made up of a number of small code segments that are linked in some way.

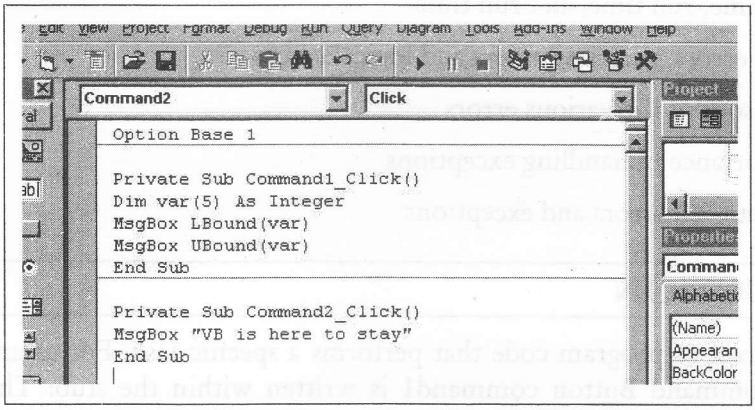
For example, the code for the click event of a Command Button command1 is written within the stub:

```
PRIVATE SUB command1_CLICK( )
...
END SUB
```

As is very obvious, we might be having another command button 'command2' and the code for its click event will be written within

```
PRIVATE SUB command2_CLICK( )
    ...
END SUB
```

Here is a sample picture of the code window of a VB project.



We can clearly see that we have code written for the click event of different command buttons. The different code segments are self-contained and are separated from one another by the lines running across the code window. These code segments perform a particular task. These are what we call procedures. Thus, a procedure is a code segment that performs a particular task and is self-contained.

A procedure offers the advantage of better management of code. For example, a code might be required at multiple locations in a project. It can be pasted at multiple locations, and the output will be the same. Now, any change in the code means all the copies have to be updated. A procedure will help avoid this repetitive work. We can write the code for the procedure at only one place and we can call on this procedure through code. Thus, any changes to be made will be limited to just one location.

The kinds of procedures we have used so far are known as 'event procedures'. This is because they are procedures that are related to an event, and will be executed when a certain event is raised. Thus, we primarily code through event procedures in VB.

Types of Procedures

All the code of Visual Basic applications is written in procedures. There are four different types of procedures in Visual Basic:

- The first is a sub, which does not return a value.
- The second is a Function that returns a value.
- The third is the event handling procedure.
- The fourth is a property procedure that also contains code.

You can choose from two types of general procedures:

1. ***Function Procedure:*** It performs actions and returns a value after performing its assigned task
2. ***Sub Procedure:*** Completes the task but does not return a value

One example can make the difference clear.

Suppose you need to set property values of a set of objects, then you need a sub procedure. On the other hand, when you do any calculations and need the result in one object, then you should use functions.

Both sub-procedures and functions are called the methods in object oriented programming.

3.3 WORKING WITH SUB-PROCEDURE

Let's revisit the definition of sub-procedure.

Sub-procedure is a collection of code that can be invoked from one or more places in an application.

Sun-procedure is processed only when called (invoked) from code. Sub procedure can be invoked within a set of Sub and End Sub statements as given below:

```
Private sub procedure_name(parameterlist)
.....Actions to be performed
End Sub
```

Private indicates procedure can only be used by other procedures in the current form

Sub keyword identifies procedure as a Sub Procedure

Procedure name - name given to procedure

Naming rules same as for naming variables

Parameter list - optional set of memory locations referred to as parameters

ByVal, ByRef specify how parameter is passed (discussed in details later)

The sub Main requires an 'end sub', or else the above stated error is generated. In the world of VB, there is no 'begin' statement, but there is a mandatory 'end' statement.

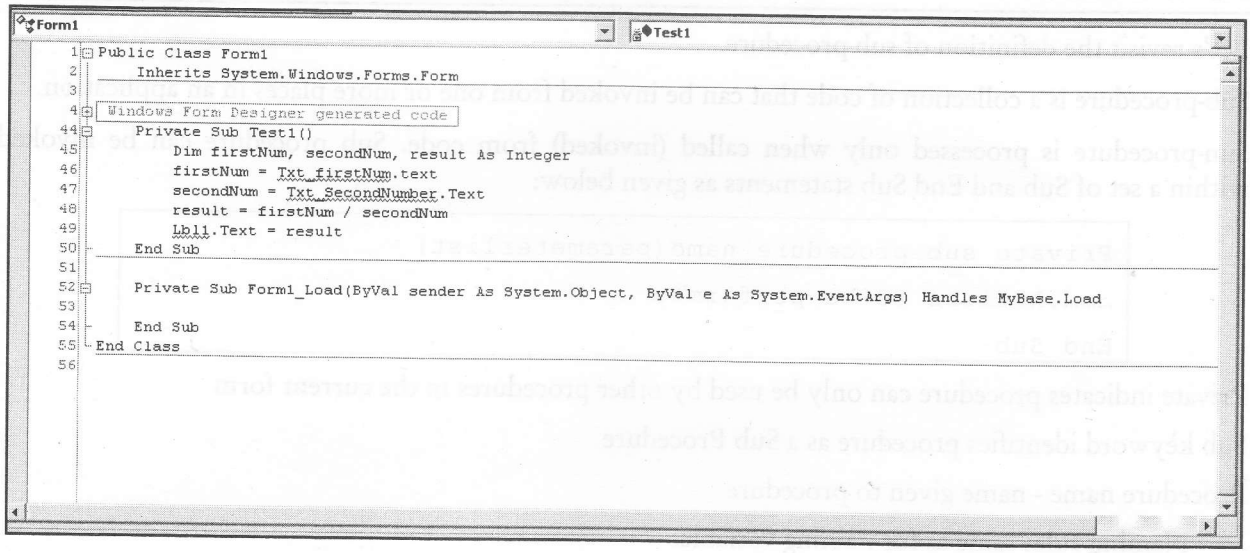
```
new.vb
   class abc
      shared Sub Main()
            call sample
       End sub
       shared sub sample
                ystem.Console.WriteLine("Hello World")
        end sub
End class
   Output
Hello World
```

Parentheses are automatically added to the line Private sub procedure_name, when you type it and press enter. Even End Sub is automatically added to it.

Steps for adding Sub procedure in the Code Editor Window

1. Open the Code Editor Window
2. Click a blank line in the Code Editor window. The blank line can be anywhere between the Windows Form designer generated code and the End Class Statement, but outside of any other Sub or Function.
3. Type Sub procedure header and Enter key. By doing this, the code editor automatically inserts the footer(End Sub) for the procedure.

Example



The code entered is written below:

```
Private Sub Test1
Dim firstNum, secondNum, result As Integer
        firstNum = Txt_FirstNumber.Text
        secondNum = Txt_SecondNumber.Text
       result = firstNum / secondNum
        Lbl1.Text = result
End Sub
```

In the example, Txt_FirstNumber, Txt_SecondNumber & Lbl1 are two text boxes and one label.

3.3.1 Passing Data to a Procedure with Arguments

At times you need to use the value of a variable in one procedure and then in a second procedure that is called from the first. The problem can be solved by using module level procedure, but then variable will be visible to all the other procedures also. The other approach is to declare the variable as local and pass it as parameter to other procedures.

When a sub procedure names an argument, any call to that procedure must supply the argument. In addition, the argument value must be the same data type in both locations, though the name at both locations need not be the same and hence can vary. Multiple arguments can be included in the procedures, though the number of arguments, their sequence and their data types must match in procedure header and the call to the procedure.

Including Parameters in an Independent Sub Procedure:

```
Private sub procedure_name(parameterlist)
.....Actions to be performed
End Sub
.
.
.
Call  procedure_name(parameterlist)
```

- Call statement has an optional argument list
- Comma separated list of arguments passed to procedure being called
- Argument list must agree with number of parameters listed in parameter list in the procedure header
- Data type and position of each parameter must agree with data type and position of corresponding argument.

Passing Variables

When it comes to passing parameters, there are four distinct possibilities to mull over. The parameters can either be 'byval' or 'byref' types, or they may be 'value' or 'reference' types.

- *Pass by value*
  - ❖ Use keyword ByVal
  - ❖ Passes contents of variable to the parameter in receiving procedure but not memory location
  - ❖ Cannot change contents of passing variable.
- *Pass by reference*
  - ❖ Use keyword ByRef
  - ❖ Passes memory address of variable to the receiving procedure
  - ❖ Contents of variable can be changed within receiving procedure
  - ❖ The parameters to a sub or a function must be separated by commas.

The differences between Pass by value and Pass by reference can be described as follows:

- It is easiest to understand the concept of passing parameters by value. Here, any change in the value of the variable in the procedure, is not reflected in the original variable.

- Passing parameters by reference is very distinct from the above. The value contained in the original variable, changes with any change introduced in the variable in the sub. Reference type variables are actually pointers to the original data.
- A 'pass by value' affects only the value of the variable or the member, without affecting the original variable; whereas, a 'pass by ref' changes both, the object, i.e. the original variable it is pointing to, as well as, its members.
- The advantage of 'pass by ref' is that the calling sub can change the value of the parameters. There is no significant overhead in passing a variable either by ref or by value. However, it is advantageous to pass larger value types such as structures by ref, and not by value, or else, a large number of variables will have to be copied on to the stack.

3.4 WORKING WITH FUNCTION PROCEDURES

Function procedure is a block of code that performs a task and returns a value after completing that task. Function has following features:

- Function header has keyword Function instead of Sub;
- Header has "As datatype" clause that specifies data type of value returned by the function:
  - As Decimal - returns a decimal number,
  - As String - returns a string;
- Return statement alerts computer that function has completed task and returns the value contained in its expression;
- There are subtle differences between functions and subs. In a sub, the parameters can have any name, but this is not true of a function. There are two ways of returning values in a function;
- The first is by using the return statement, and
- The second is by simply initializing the name of the function.

A function declaration establishes the name of the function and the number and types of its parameters. A function declaration consists of a return type, a name, and a parameter list. In addition, a function declaration may optionally specify the function's linkage. A declaration informs the compiler of the format and existence of a function prior to its use. A function can be declared several times in a program, provided that all the declarations agree.

If a parameter is assigned the same name as a function, the confusion is bound to be created. This is because, it is an extremely arduous task to ascertain whether the programmer is setting the return value, or is changing the value of the parameter. This error applies to functions only, since subs do not return values.

Optional Keyword

The 'optional' keyword proves to be useful in situations when the user is not required to call the procedure with all its parameters. The parameters that have been overlooked are assigned their default values. However, the sub has no way of identifying the parameters that it has been called with.

3.5 UNDERSTANDING SCOPE AND ACCESSIBILITY

Consider the following VB.Net code:

```
Sub Test( )
 If varx <> 0 Then
  Dim intvar As Integer
  intvar = 1/varx
 End If
MsgBox CStr(intvar)
End Sub
```

In this code, the variable intvar is not recognized outside the block in which it is defined, so the final statement will produce an error.

It is important to note that the lifetime of a local variable is always that of the entire procedure, even if the variable's scope is block-level. This implies that if a block is entered more than once, a block-level variable will retain its value from the previous time the code block was executed.

Variable Scope

The Scope and Lifetime of a Variable:

- ***Scope:*** Indicates where in the application's code the variable can be used
- ***Lifetime:*** Indicates how long the variable remains in the computer's internal memory
- ***Procedure Scope:*** Procedure-level variable is a variable declared within a procedure. Variable can only be used in procedure in which declared
- ***Module scope:*** Module-level variable is a variable declared in General Declarations section of the form. Variable can be used within all procedures in the form
- ***Block scope:*** Block-level variable is a variable declared inside specific blocks of code

  If...Then...Else or For....Next

  It can be used only inside the block in which declared.

  All variables have a predefined scope that is assigned during initialization. Listed below are a few of the most common scope declarations and their definitions.

- ***Private scope:*** Defines a variables scope as restricted to the current method. A variable defined as having private scope is referred to as a member variable and is commonly prefixed with an "m".
- ***Public scope:*** Allows the parent class, or calling class, access to the data held by a public variable or method.
- ***Friend scope:*** Similar to public scope as far as all code within a project is concerned. The difference between the public scope and friend scope is that variables or methods that are defined with the friend scope cannot be accessed by a parent class outside of the project.

- *Protected scope:* A new scope declaration that allows access to classes that inherit from the variables class.

We have discussed below some of the scope declarations in detail as below.

3.6 DIM

This is how we have been declaring our variables so far. We will now discuss the implications of using this keyword. Here is a sample code:

Code Listing dim.1

```
PRIVATE SUB COMMAND1_CLICK()
1.  DIM var1 AS INTEGER ' var1 initialized to default value of 0
2.  var1 = var1 + 1     ' var1 incremented by 1
3.  MSGBOX  var1        ' pop-up dialog box shows value of var1
END SUB
```

In this code, on line 1, we have declared an *integer* variable named 'var1'. On line 2, we are increasing the value of the variable by 1. ["var1 = var1 + 1"] instructs VB to "add 1 to the current value of var1, then make that the new value of var1").

What is important about this code is that once the line '*end sub*' is reached, the value of 'var1' is lost – the value is 'destroyed' by VB. This is an in-built provision for managing variables and memory efficiently. Once the *end sub* statement of a procedure (a code module of a program) is reached, all *dim*-declared variables in that procedure are destroyed. This way, the memory consumed by them can be freed up. Otherwise, the variables will be unnecessarily eating up memory.

Let us see how Code Listing dim.1 progresses, on a line-by-line basis. When we click on the command button Command1 for the 1st time, the following things happen:

1. Line no. 1 is executed and 'var1' is initialized to 0. (in VB, all numeric data types are initialized to a default value of 0).
2. Line no. 2 increases 'var1' by 1, thus 'var1' becomes '1'.
3. Line no. 3 displays the current value of 'var1' in a message box, i.e., '1'.

When we click on Command1 for the 2nd time, the following things happen:

1. Line no. 1 is executed and 'var1' is initialized to 0.
2. Line no. 2 increases 'var1' by 1, thus 'var1' becomes '1'.
3. Line no. 3 displays the current value of 'var1' in a message box, i.e., '1'.
4. The procedure 'Command1_Click' ends, destroying the variable 'var1'. Since 'var1' is destroyed, its current value of '1' is lost.

When we click on Command1 for the 3rd and even the 300th time, the same things happen as we have just seen. Thus, we can keep clicking on command1 but the value of 'var1' will not increase correspondingly; it will be stuck at 1.

An important implication of this is that the variables declared with *dim* cannot be used in multiple procedures in our program. For example, a variable which we need to use on clicking two different list boxes ('List1' and 'List2') should not be declared with *dim*.

To sum up, variables declared with dim will

- Have only a procedure-level 'scope'.
- Lose their stored value once the *end sub* statement has been executed.

3.7 STATIC

This is used when we want a variable to 'maintain' its value, though the scope is still procedure-level. This means that the variable should 'carry over' its value once a procedure ends, though we should not be able to read/write its value outside the procedure where it has been declared. The *static* keyword can fulfil this dual need. A *static* declared variable will not lose its value on execution of *end sub* statement. To understand this, consider this code:

Code Listing static.1

```
PRIVATE SUB COMMAND1_CLICK()
1.  STATIC var1 AS INTEGER                    'static used instead of dim
2.  var1 = var1 + 100
3.  MSGBOX var1          'pop-up dialog box shows value of var1
END SUB
```

In this code, on line no. 1 we are declaring var1 with the *static* keyword instead of *dim*. The remaining code is the same as Code List dim.1. Now, whenever we click the command button Command1, the value of 'var1' is incremented by 100, on line no. 2. Thus,

- On clicking Command1 the 1st time, '100' is displayed in a message box
- On clicking Command1 the 2nd time, '200' is displayed
- On clicking Command1 the 3rd time, '300' is displayed

and this process will go on and on. Why does this happen?

The reason behind the different results of Code List dim.1 and Code List static.1 is this: In Code Listing static.1, var1 is declared with *static*. So, in Code List static.1, the previous value of var1 is maintained even after *end sub* execution, unlike code List dim.1, where var1 loses its previous value every time *end sub* is reached.

Let us see how Code List static.1 progresses, on a line-by-line basis. When we click on Command1 for the 1st time, the following happens:

1. Line no. 1 is executed and 'var1' is initialized to 0
2. Line no. 2 increases 'var1' by 1, thus 'var1' becomes '1'.
3. Line no. 3 displays the current value of 'var1' in a message box, i.e., '1'.

4. The event procedure 'Command1_Click' ends, preserving the current value of the *static* variable 'var1' created in the event procedure, i.e., '1'.

When we click on Command1 for the 2nd time, the following happens:

1. Line no. 1 is executed and 'var1' continues from previous value of '1'.
2. Line no. 2 increases 'var1' by 1, thus 'var1' becomes '2'.
3. Line no. 3 displays current value of 'var1' in a message box, i.e., '2'.
4. The procedure 'Command1_Click' ends, preserving the current value of the *static* variable 'var1' in the event procedure, i.e., '2'.

When we click on Command1 for the 3rd time, the following happens:

1. Line no. 1 is executed and 'var1' continues from previous value of '2'.
2. Line no. 2 increases 'var1' by 1, thus 'var1' becomes '3'.
3. Line no. 3 displays the current value of 'var1' in a message box, i.e., '3'.
4. The procedure 'Command1_Click' ends, preserving the current value of the *static* variable 'var1' in the event procedure, i.e., '3'.

Thus, as we keep clicking on the command button, the value of 'var1' will keep on increasing correspondingly, because var1 is declared as *static.*

To sum up, variables declared with static will:

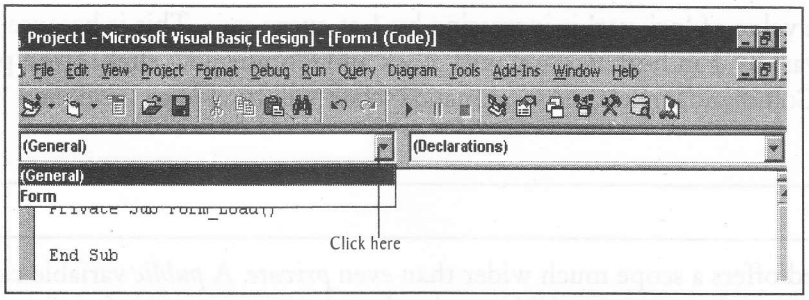
- Have only a procedure-level 'scope'.
- Not lose their stored value once *end sub* is executed.

3.8 PRIVATE

Imagine a situation in which we want to use a common variable for three different procedures. This is where the *private* keyword comes in. A variable declared as *private* will have module- level (or Form-level) scope. This 'module' can refer to a form or to a class. Thus, this kind of a variable will 'maintain' its value till the time the form or the class is in use. Hence, it can be called from any procedure in the form or the class. Also, it will have the same value for all the procedures that are using it. Once the form is closed or the class is destroyed, the variable will lose its stored value.

A *private* variable is declared at a very specific place in our code window. We declare the *private* variable in the *general/declarations* section of the code window. To go to the *general/declarations* section of the code window, simply *follow the following steps:*

1. Open the code window.
2. Position the mouse at the top of the code window, on the arrow at the left drop-down list (as shown in the figure).



3. Click on the drop-down arrow of the combo box.
4. From the drop-down list that appears, choose *general.*

As we can see, we now have a blank line inserted at the top of the code window. This is our '*general/declarations*' section in VB. Once we have done this much, we can type the following lines in the *general/declarations* section of the code window

PRIVATE pri_var AS INTEGER

This line will declare a *private* variable by the name 'pri_var'. This variable, declared in the general/declarations section of the form, will have a Form-level 'scope'. Now consider the following lines

Code Listing private.1

```
PRIVATE SUB COMMAND1_CLICK()
pri_var = pri_var + 1
MSGBOX pri_var
END SUB
```

Code Listing private.2

```
PRIVATE SUB COMMAND2_CLICK()
pri_var = pri_var + 1
MSGBOX pri_var
END SUB
```

In code listing private.1 and code listing private.2, we are using the same variable 'pri_var'. Now follow these steps to understand the 'scope' of variable 'pri_var':

1. Click on button Command1. Note the value of pri_var on the messagebox.
2. Click on the command button Command2. Note the value of pri_var.
3. Click again on Command1. Note the value of pri_var.
4. Click again on Command2. Note the value of pri_var.
5. Click again on Command1. Note the value of pri_var.

What result do we see? The message box should display the following values of 'pri_var' if we properly follow the steps from 1 to 5:

1 ® 2 ® 3 ® 4 ® 5

As we can see, the value of 'pri_var' is increasing by 1 at every step. This is because we have increased the value of 'pri_var' by 1 in both, Command1_click and Command2_click. Since pri_var is of Form-level scope, both Command1_click and Command2_click will be able to 'read' and 'write' the value of pri_var.

3.9 PUBLIC

The *public* keyword offers a scope much wider than even *private*. A *public* variable can be shared in the entire project. Whereas the *private* variable can be shared across an entire form or module, the *public* variable can be shared across the entire application or project. Thus, a variable declared with the *public* keyword will have project-level or application-level 'scope'. It will only lose its value when the project is stopped. If we close the form on which we are currently working, the *private* variables declared on that form will lose their value but the *public* variables in the project will maintain their value.

To see public variables in action, consider the following code:

```
PUBLIC var1 AS INTEGER    'in general/declarations of form1
```

Code Listing public.1

```
PRIVATE SUB COMMAND1_CLICK( )              ' written in form1
var1 = var1 + 1
MSGBOX  var1
form2.SHOW
END SUB
```

Code Listing public.2

```
PRIVATE SUB COMMAND1_CLICK( )          'written in form2
var1 = var1 + 1
MSGBOX  form1.var1
form1.SHOW
END SUB
```

Here, we have written Code Listing public.1 in 'form1' and Code Listing public.2 in 'form2'. Thus, we are using the two code lists on two different forms. However, since the variable has been declared as *public* in form1, we can use the same variable on two or more 'forms'. It will have the same value on 'form1' as well as 'form2'. Any change in the value of 'var1' made by 'form1' can be 'seen' by 'form2'. Similarly, any change in 'var1' made by 'form2' can be 'seen' by form1, since 'var1' has been declared with the keyword *public* in 'form1'.

A variable declared as *public* is qualified by the name of the *form* in which the variable has been declared, e.g., form1.var1

To call a *public* variable, we refer to the form name alongwith the variable name. *For example, we write*

```
MSGBOX form1.var1
```

instead of using the line

```
MSGBOX var1
```

which we have used in the discussion on *private* variables.

3.10 ERROR HANDLING

Error handling is an essential part of programming because it can help make the program error-free. An error-free program can run smoothly and efficiently, and the user does not have to face all sorts of problems such as program crash or system hang.

Programming errors come in three varieties:

1. Syntax errors
2. Run-time errors
3. Logic errors

3.10.1 Syntax Errors

Syntax errors are those errors which occur due to violation of syntax or grammar rules of the language. These errors are noted during compilation of the application. The reasons of the syntax errors can be one or more of the following:

- Punctuation
- Format
- Spelling etc.

Fortunately the smart editor finds most errors for you and even corrects them. The syntax errors that the editor cannot identify are found and reported by the compiler as it attempts to convert the code into intermediate machine language. A compiler-reported syntax error may be referred to as a compile error.

The editor can correct some syntax errors by making assumptions and not even report the error to you. For example, if you type the opening quote of some string for example "This is string" and you forget the closing quote, the editor automatically adds the closing quote for your string when you move to the next line. In the same way, if you forget to place parentheses after a method name, such as sub1( ), the editor relieves you of this job and puts them for you when you move to next line. But don't be too dependent on the editor as the assumptions and interpretations can sometimes prove wrong, so be careful.

The editor identifies the syntax errors as you move to the next line. A blue squiggly line appears under the part of the line that the editor cannot interpret and a message appears in the task list at the bottom of the screen.

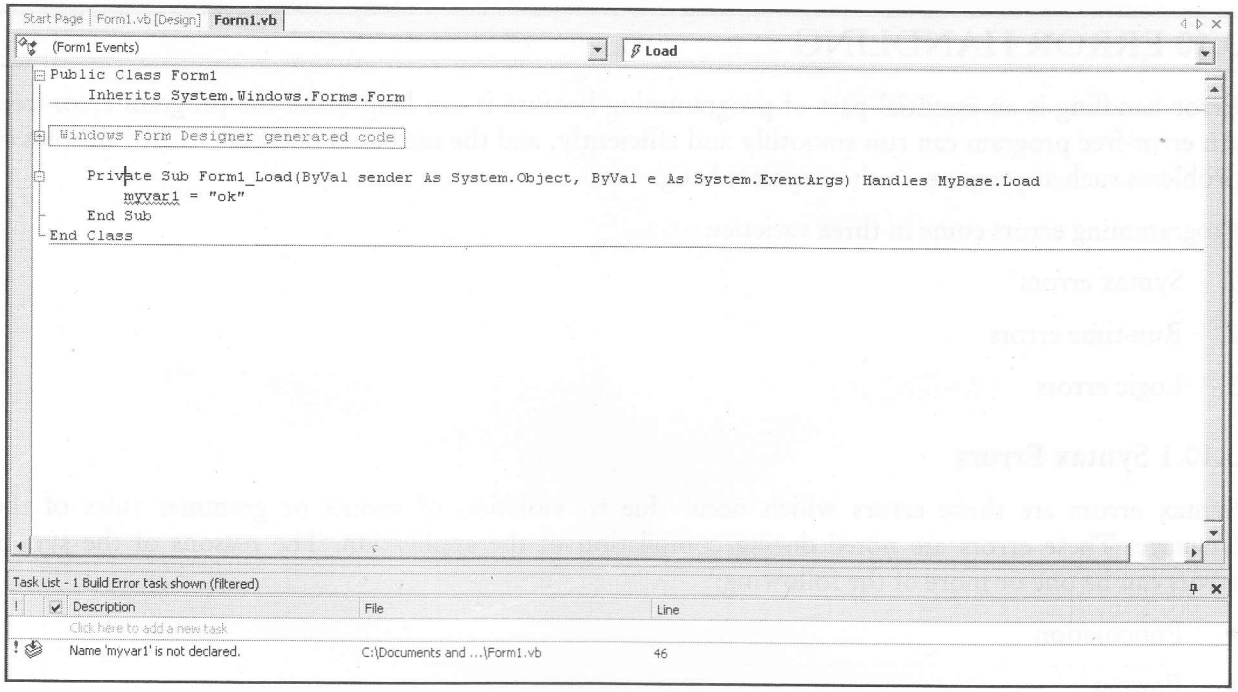


Figure 3.1

To be able to locate the errors in a more proper way you can display the line numbers to locate the line number of the statement that caused the error with

Tools/options/Text editors/Basic/Display/Line Numbers.

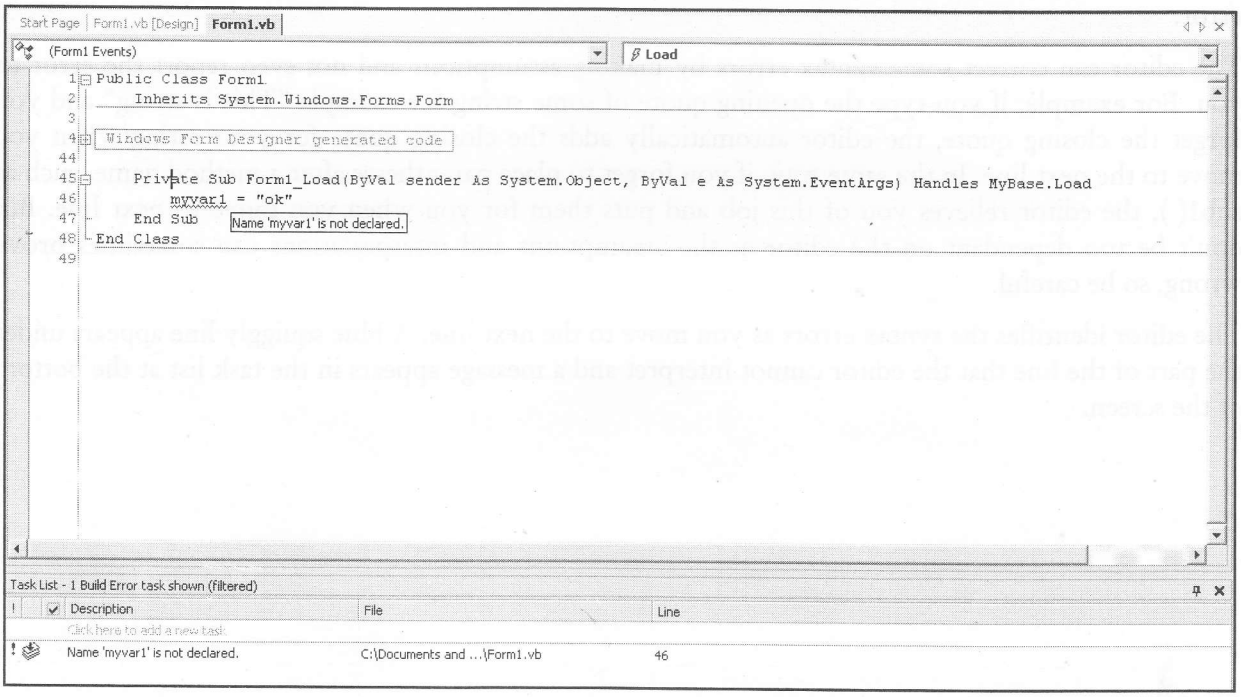


Figure 3.2

The quickest and easiest way to jump to an error line is to point to a message in the Task List and double-click. The line in the error will display in the Editor window with the error highlighted.

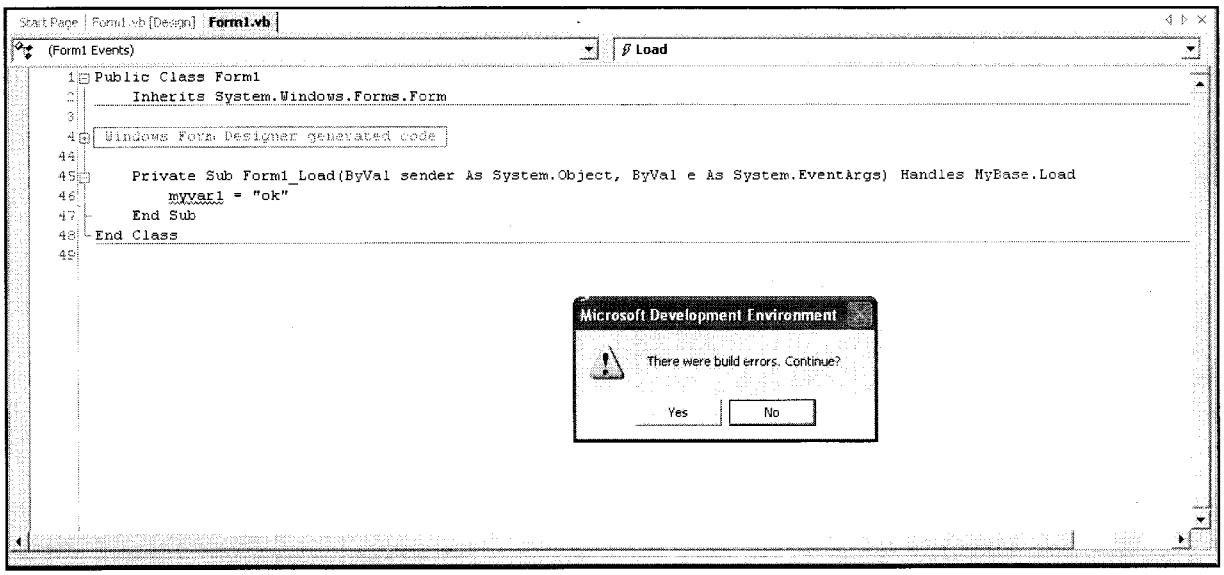


Figure 3.3

3.10.2 Run-time Error

If your project halts during execution, the type of error is run-time error. The run-time error is also called an exception handling which is the most important feature of VB.Net. When run-time error occurs, VB.Net displays a dialog box and highlights the statement that caused the error. Statements that cannot execute correctly cause run-time errors. These statements have passed compilation test so don't have any syntax error, however fail to execute due to so many difficult to avoid reasons. Run-time errors can be caused by attempting to do impossible arithmetic operations such as calculate with non numeric data, divide by zero or find the square root of a negative number.

3.10.3 Logical Errors

Logical errors are the incorrect results the project produces while running. The project might not be executing a loop expected number of times or exit from a particular block unexpectedly or producing wrong calculation results.

3.11 SYSTEM EXCEPTION HANDLING AND DEBUGGING

3.11.1 Debugging

The act of trying to find and fix errors. Computer programmers don't call errors as errors; instead they love to call them by the name "bugs". So, projects don't have errors, they have bugs. Finding and fixing these bugs is called debugging. VB.Net editor is very intelligent and at the same time helpful in helping you find errors, but they are only syntax and logical errors. However it says sorry for helping you in finding logical errors; that's completely your job, though you can take the help of various debugging tools to monitor the flow of the logic in your project.

Source of Bugs

- A mistake in the strategy used to perform some task called logic errors
- A mistake in entering the code, a type
- A mistake caused by user entry or by the data

So, according to whatever we have understood till now, we can say that there are three modes in the life of a program. These three distinct modes are namely;

- Design mode
- Run mode
- Break or debug mode

Design mode is very crucial and very significant phase as the design will make a foundation of the overall project. If the project is not designed and coded properly, can result in production of wrong results, logical errors.

Run mode combines the compilation and execution of the project. So, any syntactical errors oops! Bugs are debugged in this mode where as run-time errors or exceptions to be more specific are handled during writing the code for the project. If there are still any bugs, debugging is performed and bugs are removed.

Earlier debugging an application meant running the code to get an error message if any that would tell you the line number on which your application failed. Then the next step was to print your application code on that wide green and white striped paper with holes on each side and take it home. Then the entire time was spent in trying to trace the code to find the source of error. For the last decade we've had a number of debuggers with several tools for investigating different aspects of the problem to speed up the process of debugging applications and allow the developer to spend more time addressing business problems and less time mired in code searching of a simple syntax error. But not the entire scenario has changed, programming techniques have advanced and the interaction and coordination with a variety of different platforms has increased, there has been a need for more advanced debugging tools. *Visual Studio.Net provides numerous debugging tools, including windows for viewing internal aspects of your application while it runs, advanced tools for managing break-points, and the ability to quickly and easily access code which may reside in a number of different components:*

- Watch window displays values for selected variables.
- Locals window displays all variables in the current procedure.
- Autos window displays all variables in the current and previous statements.
- Quick Watch dialog allows for quick viewing of any variable and works closely with the Watch Window.
- Call Stack window displays the names of functions used to the point of current execution.

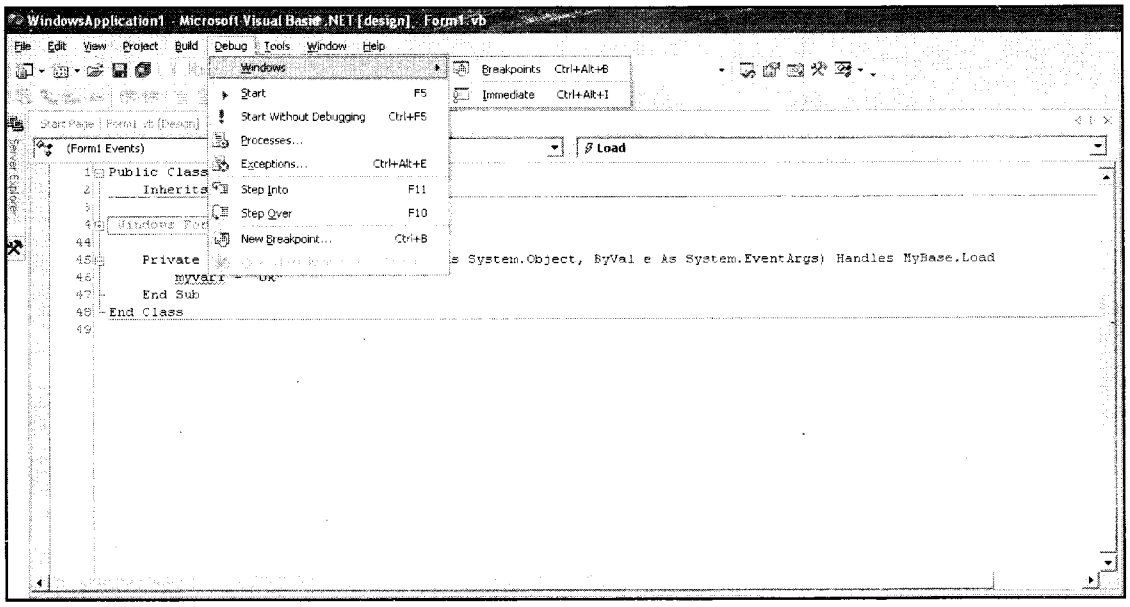


Figure 3.4: The many Debugging Tools available at Runtime

3.11.2 Exception Handling

When in your project you develop user interface and allow user to input numbers and use those numbers in calculations, lots of things go wrong. If the user enters non-numeric date when numeric data is to be input, the project will fail to run and will cause error. Though cautious programmer take precautions beforehand by using conversion functions like CInt, CDec etc., but even these functions fail if user leaves text box blank, or enters non numeric data. Even the data input by user can result in attempt to divide by zero. Each of these situations cause an error to occur or in the terminology of programmers, throws an exception.

Difference between Errors and Exceptions

- Error could be a result not expected.
- Exception - something that happens out of the ordinary and usually unexpectedly in application outside the normal flow of control
- SHE(Structured Exception Handling) defines a particular way to dealing with exceptions.
- Exceptions are shared between all languages supported by .Net platform

There are two main ways to prevent problems with the code. They are as follows,

- Structured exception handling
- Debugging

There are two kinds of exceptions or unexpected occur namely,

- Hardware - caused by h/w. e.g. when program tries to access memory it shouldn't.
- Software - e.g. trying to assign an incompatible value to a variable

Exception handling in managed extensions is much more developed. Exception handling can be carried out by using the Try and Catch block of statements. When an application throws an exception, the exception handler for the same is implemented by the runtime. Managed extensions use the System::Exception class for handling exceptions and this class can be used to trace the function that raised the exception.

3.12 STRUCTURED EXCEPTION HANDLING

A good programming skill is to catch the exception by using VB.Net's structured exception handling. Catching is taking care to handle the exception before it can be detrimental i.e. before it can cause a run-time error and handle the situation, in the program as far as possible. Catching exceptions as they happen is called as error trapping or exception handling. It's a standard feature of Visual Studio.Net.

Structured Exception Handling is a strategy for handling either hardware or software exceptions.

3.12.1 The Try .... Catch ... Finally Statement

Knowing that exceptions will occur is one thing, but knowing what to do when they appear is another issue. VB.Net has adopted a new approach in handling errors, or rather exceptions handling. It is supposed to be more efficient than the old On Error Goto method, where it can handles various types of errors within the Try...Catch...End Try structure.

In structured exception handling, you use the Try .... Catch ... Finally statement to handle exceptions. This can look surprising to you but, Try .... Catch ... Finally statement appears under "Handling Program Flow" as while handling exceptions you are actually controlling the flow of program. Following are the rules of handling exception with Try .... Catch ... Finally:

Code causing exception should be placed in a Try..... End Try block.

Within the Try..... End Try blocking, you write Catch block that responds to exceptions that may occur in the Try block in an attempt to handle exception.

If you want to always execute, regardless of whether exception has occurred or not, write code in finally block.

The structure looks like this:

```
Try
    statements
Catch exception_variable as Exception
    statements to deal with exceptions
finally
    statements
End Try
```

3.12.2 Basic Rules

Basic rules govern the use of a Try block. These are as follows:

- All Try blocks must employ at least one catch or finally clause.
- A Catch clause with no other parameters will catch all unhandled exceptions.

- The finally clause always executes when available except when an Exit Try occurs. As such, the finally clause is a good place to perform component cleanup. If an Exit Try statement is used anywhere in the Try block then it is better to perform cleanup after the End Try block statement.
- Developers familiar with the On Error statement may still perform error handling as they did in Visual Basic only when a Try block does not exist in the procedure.

If an exception occurs in the protected code, it must be caught and dealt with. Let's consider an example of divide by zero situations with and without structured exception handling.

Without structured exception handling

```
module noexception
sub main()
    dim dividend as decimal = 20
    dim divisor as decimal = 0
    dim result as decimal
        result = dividend / divisor
        system.console.readline()
end sub
end module
```

With exception handling

```
module with exception
sub main()
    dim dividend as decimal = 5
    dim divisor as decimal = 0
    dim result as decimal
    try
            result = dividend / divisor
        catch ex as exception
system.console.writeline("division by 0 has occurred")
system.console.writeline("please  check the divisor")
    end try
    system.console.readline()
end sub
end module
```

In this example, the Catch statement will catch the exception when the user enters a non-numeric data and return the error message. If there is no exception, there will not any action from the Catch statement and the program returns the correct answer.

1. Drag a button onto your Windows Form and label it "Try...Catch".
2. Change the buttons name to "btnTryCatch".

3. Drag a label onto your Form and name it lbl1
4. Drag one more label onto your Form and name it lbl_ErrorMsg
5. Apply the following code in the click event of the button "btnTryCatch".

```
Dim firstNum, secondNum, result As Integer
Try
       firstNum = Txt_FirstNumber.Text
       secondNum = Txt_SecondNumber.Text
      result = firstNum / secondNum
       Lbl1.Text = result
Catch ex As Exception
      Lbl1.Text = "Error"
      Lbl_ErrorMsg.Visible = True
      Lbl_ErrorMsg.Text = " One of the entries is not a
number!    Try again!"
End Try
```

3.12.3 The Finally Clause

Let's modify example, to have finally clause. This example employs the same code as the one previously, except it demonstrates that the finally clause always executes:

1. Drag another button onto your Windows Form and label it "Finally".
2. Change the buttons name to "btnFinally".

Apply the following code in the click event of the button.

```
Dim firstNum, secondNum, result As Integer
Try
    firstNum = Txt_FirstNumber.Text
    secondNum = Txt_SecondNumber.Text
    result = firstNum / secondNum
    Lbl1.Text = result
Catch ex As Exception
    Lbl1.Text = "Error"
    Lbl_ErrorMsg.Visible = True
    Lbl_ErrorMsg.Text = " One of the entries is not a number!
    Try again!"
Finally
Dim obj As New System.Text.StringBuilder()
obj = obj.Append("Regardless of whether or not an ")
obj = obj.Append("exception occurs, the Finally clause ")
obj = obj.Append("will execute.")
MessageBox.Show(obj.ToString)
obj = Nothing
End Try
```

The Try statement can be used in one of the three following ways:

1. A Try statement followed by one or more Catch blocks.
2. A Try statement followed by a Finally block.
3. A Try block followed by both Catch and Finally.

Zero or more catch blocks are associated with a try block, and each catch block includes a type filter that determines the types of exceptions it handles.

When an exception occurs in a try block, the system searches the associated catch blocks in the order they appear in application code, until it locates a catch block that handles the exception. A catch block handles an exception of type T if the type filter of the catch block specifies T or any type that T derives from. The system stops searching after it finds the first catch block that handles the exception. For this reason, in application code, a catch block that handles a type must be specified before a catch block that handles its base types, as demonstrated in the example that follows this section. A catch block that handles System.Exception is specified last.

If none of the catch blocks associated with the current try block handle the exception, and the current try block is nested within other try blocks in the current call, the catch blocks associated with the next enclosing try block are searched. If no catch block for the exception is found, the system searches previous nesting levels in the current call. If no catch block for the exception is found in the current call, the exception is passed up the call stack, and the previous stack frame is searched for a catch block that handles the exception. The search of the call stack continues until the exception is handled or until no more frames exist on the call stack. If the top of the call stack is reached without finding a catch block that handles the exception, the default exception handler handles it and the application terminates.

On Error GoTo

If you choose to use unstructured exception handling, use On Error GoTo statement still supported by VB.Net.

The Err object

Err object contains the information about the error if any has occurred and helps you to determine whether to attempt to fix it or let it go and ignore it. Like any other object, Err object has several methods that allow you to raise errors or clear the state of Err object. The Err object, which is inherited from Microsoft Visual Basic 6.0, can only be used to catch errors in a procedure that implements unstructured error handling with the On Error statement.

The Err object contains information about a run-time error that occurred during the execution of the application. As the execution enters a procedure that contains error handling, the properties of the Err object are set to zero (0) or zero-length-string (""). If an error occurs during the execution, the properties are then set to provide unique information about the error that occurred, through its properties. An instance of the Err object can be obtained through a Microsoft.VisualBasic.Information.Err function; however, because the Err object is an intrinsic object with global scope, it is not necessary to create an instance of this object at run time.

In case of a run-time error, the properties of the Err object can be used to handle the error (depending on its type), as well as to display information to the user about the error that occurred.

Example of on err statement

1. Drag a button onto your Windows Form and label it "On Error".
2. Change the buttons name to "btnOnError".
3. Drag a label onto your Form and name it lbl1
4. Drag one more label onto your Form and name it lbl_ErrorMsg.
5. Apply the following code in the click event of the button "btnOnError".

```
On Error GoTo label1 ' Enable error handler
    Dim Result As Integer
    Dim myintValue1 As Integer = 100
    Dim myintValue2 As Integer = 0
    On Error GoTo 0 ' Disables the error handler
   'Moves execution to the line following the line that caused  the error.
   On Error Resume Next
   Result = myintValue1 / myintValue2 ' Division by zero, cause an  overflow
error.
     ' Catch the overflow error caused by dividing by zero.
    If Err.Number = 6 Then
      MessageBox.Show("Error Number: " & Err.Number.ToString)
      Err.Clear() ' Clear Errors
    End If
       label1:  ' Location of error handler
    Select Case Err.Number
      Case 6
        ' Display the error number.
        MessageBox.Show("Divided by zero")
      Case Else
        ' Catch all other type of errors.
        MessageBox.Show(Err.Description)
    End Select
   'Resume execution to the line following the line that caused the error.
    Resume Next
```

Check Your Progress

Differentiate between sub-procedure and function-procedure.

3.13 LET US SUM UP

Procedures allow reuse of code throughout an application and allow programmer teamwork on large and complex applications:

- Function procedures return a value, Sub procedures do not
- Independent Sub procedures and Functions are not associated with any specific object or event
- Use Call statement to invoke an Independent sub procedure
- Number of arguments and data types must match in argumentlist and parameterlist
- ByVal keyword passes contents of variable
- Value of variable being passed cannot be changed in procedure
- ByRef keywords passes the memory address
- Value of variable being pass can be changed in procedure
- Variables in parameterlist have procedure level scope.

A good programmer should be more alert to the parts of program that could trigger errors and should write errors handling code to help the user in managing the errors. Writing errors handling code should be considered a good practice for Visual Basic.Net programmers, so do try to finish a program fast by omitting the errors handling code. However, there should not be too many errors handling code in the program as it create problems for the programmer to maintain and troubleshoot the program later.

The form design should be visually appealing. It should be simple and neat. Although Web pages use bright colors and lots of images to make them attractive, this type of design takes a longer time to load the pages. Therefore, while designing pages, you should keep the following guidelines in mind: (a) The controls that need user input should have the correct tab order and should be grouped and arranged in an order that makes sense while entering data. (b) The controls should be properly aligned.

While naming variables and objects, keep the following guidelines in mind: (a) Use a proper naming notation, such as Hungarian or camel-casing notation, to name variables and objects. Hungarian notation enables you to identify the datatype of the variable from the name of the variable. So, a variable storing the first name of an employee will be declared as FirstName. In camel-casing notation, the variable names take the form firstName, with the second part of the variable, which is a noun, capitalized. (b) Name the variables and objects meaningfully. Meaningful names combined with Hungarian notation make the purpose and type of the variables clear. This results in a self-documented code, which is easy to understand and maintain. (c) Declare the variables and objects in the beginning of a procedure. Declaration in the beginning makes the code execution more efficient, besides making it easy to understand by someone looking at the code text. (d) Always initialize variables to certain default values before using them, to avoid any type conversion issues. (e) Always rely on explicit conversion functions to eliminate confusion.

While implementing the programming logic, you should do a good chunking of the code.

The chunking helps you to maintain the code and speed up debugging. Keep the following guidelines in mind: (a) If you want to implement a programming logic that returns a single result, use a function. (b) If you need multiple arguments to be passed without expecting a return value, use a procedure. (c)

If you want to create a reusable piece of code, use functions or Sub procedures or put the code in a separate class (if the code can be logically grouped).

The program should be easy to read and to understand when you need to refer back to it. Follow these guidelines while coding: (a) Always use "Option Explicit" to catch any undeclared or misspelled variables. Also, the use of "Option Explicit" makes the Web pages run fast. The "Option Explicit" option forces the explicit declaration of variables. (b) Declare one variable per line. This avoids confusion about data types. (c) Use comments wherever possible to document a difficult code section. (d) Use blank lines in the code for clarity. (e) Use proper code block indenting.

The little details do matter, you see. The user may not see the code, and you may feel that following such conventions is a waste. It's not. It helps you, and it sure helps other developers, when they check out your code. In the end, it's merely the difference between what's easy to do, and what's appropriate to do.

3.14 KEYWORDS

Procedure: A procedure is a block of program code that performs a specific task.

ByVal: ByVal keyword passes contents of variable. Value of variable being passed cannot be changed in procedure.

ByRef: ByRef keywords passes the memory address. Value of variable being pass can be changed in procedure.

Optional Keyword: The 'optional' keyword proves to be useful in situations when the user is not required to call the procedure with all its parameters.

Lifetime: Lifetime refers to how long a variable remains in the computer's memory.

Err Object: Err object contains the information about the error if any has occurred and helps you to determine whether to attempt to fix it or let it go and ignore it.

Option Explicit: "Option Explicit" option forces the explicit declaration of variables.

Bug: Bug is the error caused in the program.

3.15 QUESTIONS FOR DISCUSSION

1. Explain the difference between a sub procedure and a function procedure.
2. What is a return value?
3. How can a return value be used?
4. Explain the differences between ByRef and ByVal. When would each be used?
5. What types of errors can be fixed for the program to continue?
6. What does an on error command do?
7. What is an Err object?
8. What does the try...catch...finally statements do?
9. Differentiate between structured and unstructured exception handling.

Check Your Progress: Modal Answer

Function Procedure: It performs actions and returns a value after performing its assigned task.

Sub Procedure: Completes the task but does not return a value.

3.16 SUGGESTED READINGS

Sanjeev Sharma, *Visual Basic 6*, Excel Books

A. Chakraborti et al., *Microsoft.Net Framework*, PHI, 2002

M. Reynolds et al., *.Net Enterprise*, Wrox/SPD, 2002

Vikas Gupta, *.Net Programming*, Dreamtech publication

Mackenzie Sharkey, *Teach yourself Visual Basic.Net in 21 days*, SAMS Techmedia

Bible, Bill Evjen, Jason Beres et al., *Visual Basic.Net Programming*, Wiley India

UNIT II

LESSON

4

WINDOW FORMS

CONTENTS

4.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- State the concept of window forms
- Explain properties and methods of VB form

4.1 INTRODUCTION

Forms are something we see in our day-to-day life very frequently. When we go for a railway reservation, we get a 'form' in which we fill in our name, age, date of journey, train number, train name, station, etc. When we go for admission to a school/college, we are given another 'form' for filling in our details: name, date of birth, gender, qualifying exam marks, father's/mother's name, address, blood group, etc. When we go to the passport office for a passport, we are given a 'form' to be filled out. In the form, we provide details such as our name, age, date of birth, address, educational qualifications, etc. At an e-mail site, we fill in another 'form' providing details such as name, address, user id, password, interests and hobbies, etc., as might be relevant to the site.

Thus, we have been using different forms in our daily lives. These forms are meant for different purposes and each form will be different from the others. However, one thing is common to all of

these forms: they accept some data from us, which is to be stored and processed at a later stage. VB too provides us with these forms, for these very requirements.

As we might have understood, the 'form' is probably one of the most important tools of VB that we will use. The end-user interacts with our application through the form. He/she can perform the following activities using the VB forms:

- Entering data such as the name, age, address, marks, etc.
- Validating the data (ensuring correctness of data).
- Making requests for calculations and other processing activities, etc.
- Ending requests for storage, deletion, or updation of data.
- Sending request for generation of reports.
- We can say that the form is the 'GUI' part of VB. This is what makes the 'front-end' of our VB application.

4.2 WINDOW FORMS

VB is not just a programming language - it is much more than that. It is, in fact, an 'Integrated Development Environment' (IDE for short). Basically, the IDE refers to the integration of three aspects of programming:

- GUI designing (forms creation)
- Coding of the application
- Compiling and run-time environment.

4.2.1 GUI Container: Form Designer

The elements of the tool box interact with the user. However, they cannot exist in plain air - they need a platform on which they can 'rest'. This platform is provided by the *form designer*. We place all of these controls on the *'form'* of VB. Thus, the form designer is like a 'container' for all other controls, similar to the containers we see in our kitchens - for dal, salt, chilli, tea, coffee, etc. The text boxes, command buttons, list boxes and other controls are all drawn and placed on the *form*. Only then can they talk to the end-user.

Positioning the Form: Form Layout Window

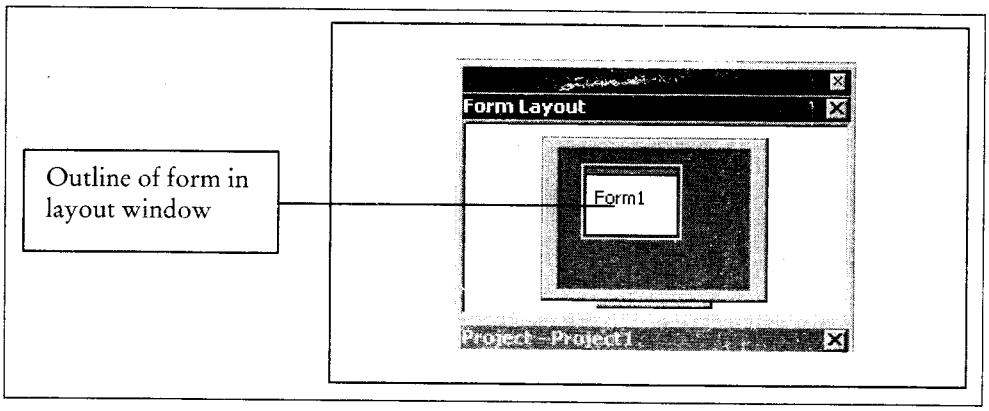
As the name suggests, the *Form Layout Window* helps to layout our *form* on the screen. When we run our project, VB will, by default, place our Form towards the top-left of the screen. But suppose that we want the form at the centre of the screen. What can be done? Or if we want the form at the bottom-left of the screen? For situations such as this, the *Form Layout Window* will be of help. Using this window, we can position our form anywhere on the screen at runtime. The *Form Layout Window* appears on the right of the screen, as shown in the figure of the IDE given before.

To use this window, we perform the following steps:

1. Run the program once
2. Stop the program

3. Bring the mouse over the *Form Layout Window*, pointing it at the icon of the form inside the window. The mouse icon changes to a 4-sided star.
4. Drag the form and drop it at the place where it is to be displayed at run-time.

This figure shows the Form Layout Window, with the outline of the form:



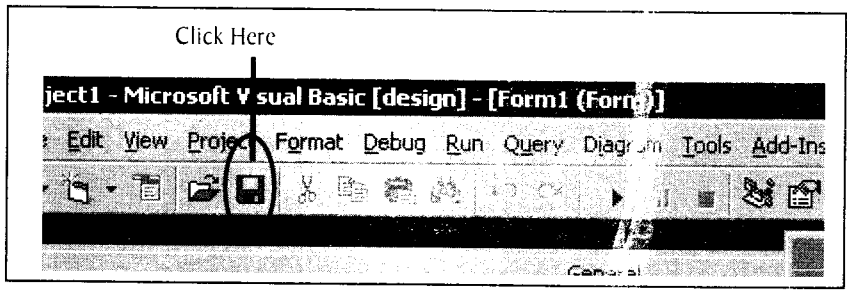
Creating a Form (User Interface)

One of the good things that VB does is that the rather difficult tasks are made fairly easy to perform. As for the already fairly easy tasks.... well, we don't have to perform them at all!! VB works behind-the-scenes for us, performing a lot of the dirty work so that our life is made easier. Creation of a form in VB also falls into the second category of tasks we just discussed, i.e., fairly easy. Thus, when we create a new project in VB we don't have to 'create' a form in VB. As we can see, one form is already created by VB, at the very start. This 'form' will be making the user interface of our application.

Saving the Form and Project

Before we go any further, let us see how to save the form and the project. In order to save the project, we need to perform one of the following tasks:

- Click on the 'save project' button on the toolbar (represented by a floppy disk) as shown in the figure:

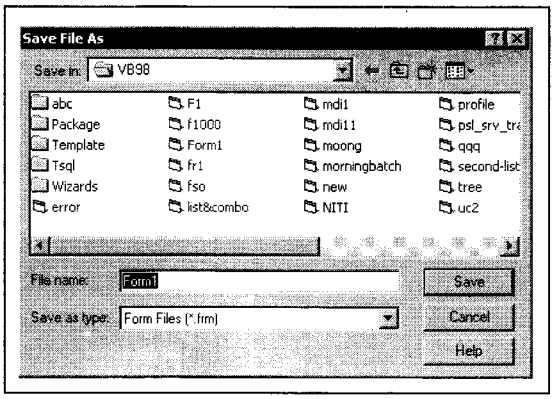


Or

- On the menu bar, click on 'File' →

'Save Project'.

Once we finish the steps mentioned above, we will see the standard Windows 'save file' dialog box. We will type a suitable name for the form and save it. It will be saved with a default extension of '*.frm*'. We should never change this '*.frm*' extension. The 'Save File As' dialog box has been reproduced here:



Once all the Forms, Classes, Reports, etc. have been saved, VB will, at the end, ask us for a name for the project. Type a suitable name for the project and save it with the default extension of '*.vbp*'. We should not change '*.vbp*' extension.

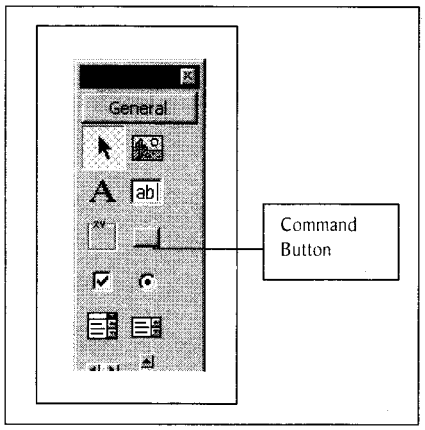
At a later stage, whenever we need to work further on the project, we will open the '*.vbp*' file we have just created and saved. As is quite understandable, a single project will contain multiple Forms, Classes and Reports. Think of the project as a container for all the Forms, Reports, etc. that are part of the project. Later on, whenever the project is loaded, all the Forms, Classes, etc. associated with it will be loaded automatically. Once a project is closed, all the forms, reports, etc. will be closed too.

4.3 PROGRAMMING THE FORM

We have created and saved the form, we have also saved our project by now. Now it is time to begin with our first programming in VB.

To begin with, we will make a simple project. This will include just one *Command Button*. When the project runs, the user will *click* on this button. When the command button is clicked, we plan to display the message "Welcome to Shourya's world of VB programming" on the screen. Since we know that VB is a GUI-based environment, the first thing we need to do is to develop the user interface. In the present sample project, we need just a *Command Button* as the GUI. To get the Command Button on the form, we:

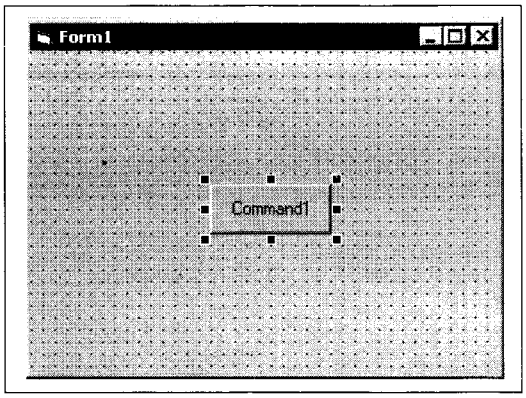
1. Take the mouse to the tool box.
2. Position the mouse on the Command Button. The button has been shown in the next figure:



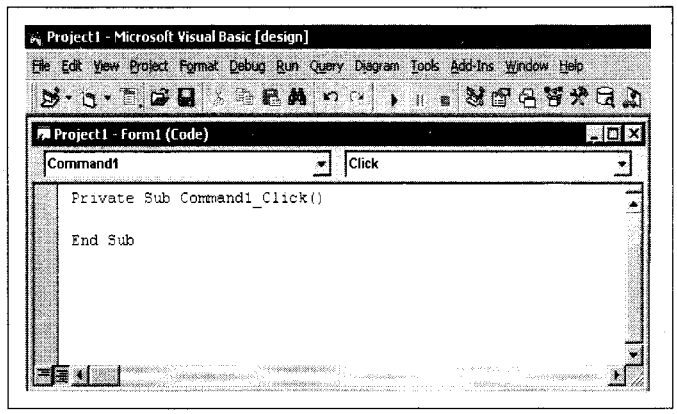
Once we rest the mouse on the Command Button, we will see a small pop-up text in a rectangle mentioning 'Command Button' to let us know that we are on the Command Button. Now we can get the button on the form through either of the two methods:

- ***Method 1:*** Click on the Command Button icon in the toolbox and come to the form. The simple 'arrow' icon of the mouse changes to a 'cross-wire' icon. We now simply press the left mouse button and 'drag and draw' the outline of the Command Button on the form. This outline represents the area over which the command button will be drawn. Once we release the left button, the command button will appear on the form.
- ***Method 2:*** Double-click on the command button in the toolbox. This automatically draws a command button with a default size on the form.

Whatever the method we use for drawing the command button on the form, we end up with a screen, which looks like this:



Now that the button has been set, we will start the programming. At present,we can presently only see the VB Form. Where do we code? To come to the code window, we double-click on the command button. What do we see? The form has disappeared! In its place, we have a new screen, which looks like this:



This screen is the 'code window' in VB. This is the 'window' where – as the name suggests – we will be doing all of our coding. We also need to observe that the title bar has changed from

```
"Project 1 -Microsoft Visual Basic [design] - [Form 1 (Form)]"
```

To

```
"Project 1 -Microsoft Visual Basic [design] - [Form 1 (Code)]"
```

To reflect the fact that we have switched from the 'Form window' to the 'Code window'. Now we need to concentrate on the two lines appearing under the toolbar:

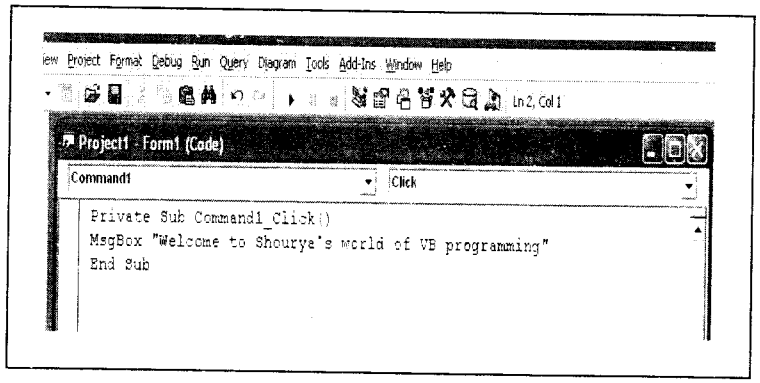
```
Private Sub Command1_Click()
End Sub
```

We will understand these lines in-depth later. What we should know at present is this: the code written between the lines "*Private Sub Command1_Click()*" and "*End Sub*" will be executed when we '*click*' on the command button drawn on the form. That's all we are supposed to know for the present. Just wait for a while and then we will come to more details on this issue.

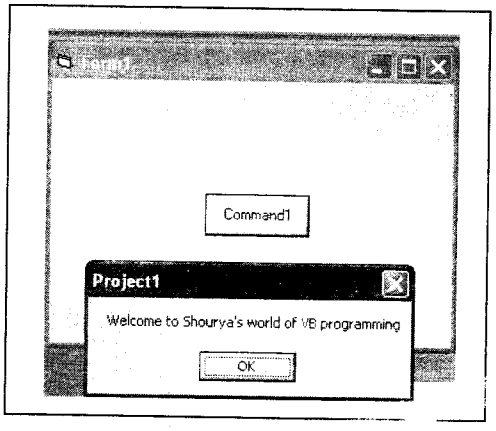
Now we need to position the cursor between the lines "*Private Sub Command1_Click( )*" and "*End Sub*" and type the following code:

```
MsgBox "Welcome to Shourya's world of VB programming"
```

Once the code has been typed, our 'code window' will look like this:



Now that our GUI and the associated code is in place, we can run our first VB project. Once the project starts running, we click on the command button on the form. As we can see, a message box with a simple message 'pops up' on the screen. We will see the output as given here:

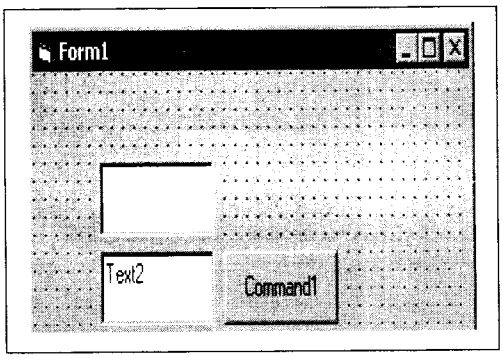


4.4 IMPORTANT PROPERTIES OF VB FORM

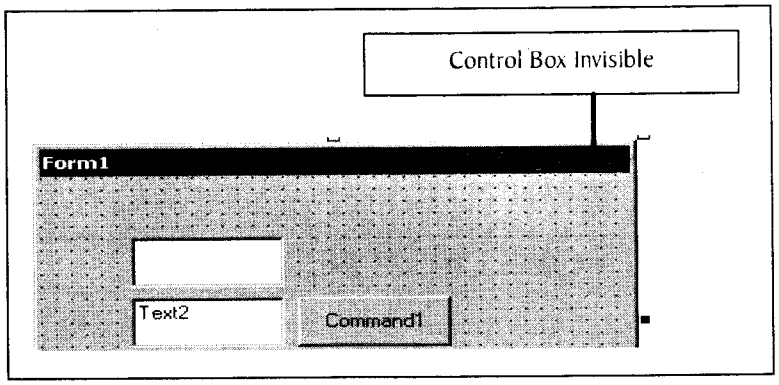
- ***AutoRedraw:*** This is a *boolean* property, i.e., it accepts values either as 'true' or as 'false'. Imagine that we have drawn something on the form through code and then minimized our form, or covered it with another window. Once we restore/maximize our form or uncover it, we see that

the graphics we had drawn through code are gone. This is because a window needs to 'refresh' the screen every time such an event happens. The *autoredraw* property of the form helps to achieve this for us.

- ***ControlBox*:** Boolean property, true by default. It determines whether the three common windows buttons at the top-right of the form are visible or not. These are the minimize, restore/maximize and close buttons, as shown in the next figure:

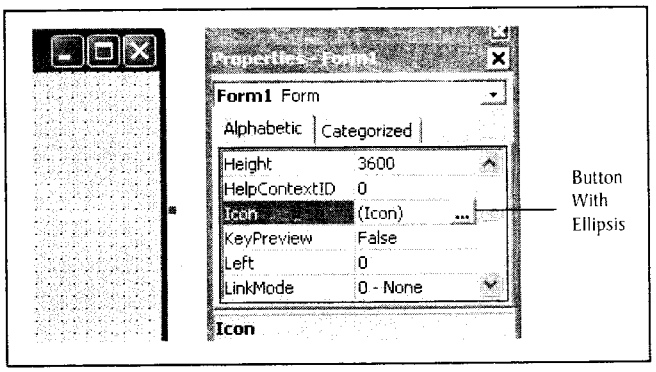


- If the control box property is set to false, it will be removed from the title bar of the form. In the figure that follows, we need to observe that the space reserved for the controlbox is 'blank'.

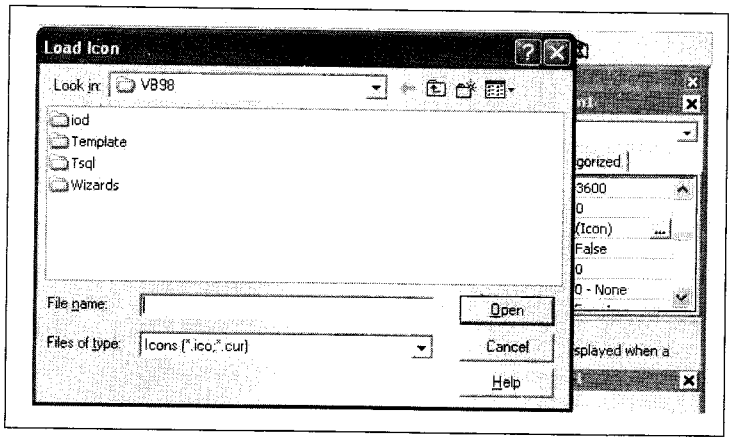


- ***Con:*** Refers to the image that should be displayed on the task bar, when the form is minimized at run-time. By default, when we minimize the VB form at runtime, the usual icon of a VB form is displayed in the taskbar. To change the default image to something else, we make use of this property.

When we select this property, a button with an ellipsis appears at the right of the properties window.



When we click on the button, we are asked for the name of the image file through the Windows dialog box we are familiar with:



- ***MaxButton***: Boolean property, true by default. It sets whether the maximize button should be enabled or not. If set to *false*, the user cannot restore/maximize the form at runtime.
- ***MinButton***: Boolean property, true by default. Sets the state of the minimize button. If set to *false*, the user cannot minimize the form at runtime.
- ***ShowInTaskbar***: Boolean property, true by default. When we run the program, the running form is displayed in the Windows taskbar. When set to *false*, the running form will not appear on the Windows taskbar.
- ***WindowState***: This property determines the starting size of the form at runtime. This property can have one of the three values that follow:

4.5 IMPORTANT METHODS OF VB FORM

- *Hide:* This method hides the current form which is activated and on which the user is working at present. An example of the usage is:

  form1.HIDE = 'TRUE'

- ***PrintForm***: This method sends an image of the form to the printer, bit by bit. Once we invoke the *printform* method, we get the exact output of the form on the printer, as it is exactly appearing on the screen.

Check Your Progress

What is GUI Container?

4.6 LET US SUM UP

The 'form' is probably one of the most important tools of VB. The end-user interacts with our application through the form. The *Form Layout Window* helps to layout our *form* on the screen. When we create a new project in VB we don't have to 'create' a form in VB. As we can see, one form is already created by VB, at the very start. This 'form' will be making the user interface of our application.

4.7 KEYWORDS

User Interface: User interface is creating a form.

MaxButton: It sets whether the maximize button should be enabled or not.

MinButton: It sets the state of the minimize button.

WindowState: This property determines the starting size of the form at runtime.

4.8 QUESTIONS FOR DISCUSSION

1. What is form layout window? Discuss the steps in creating a form.
2. Discuss the properties of VB forms.
3. What are the steps taken in programming the form?

Check Your Progress: Modal Answer

The elements of the tool box interact with the user. However, they cannot exist in plain air - they need a platform on which they can 'rest'. This platform is provided by the *form designer.*

4.9 SUGGESTED READINGS

Sanjeev Sharma, *Visual Basic 6*, Excel Books

A. Chakraborti et al., *Microsoft.Net Framework*, PHI, 2002

M. Reynolds et al., *.Net Enterprise*, Wrox/SPD, 2002

Vikas Gupta, *.Net Programming*, Dreamtech Publication

Mackenzie Sharkey, *Teach yourself Visual Basic.Net in 21 days*, SAMS Techmedia

Bible, Bill Evjen, Jason Beres et al., *Visual Basic.Net Programming*, Wiley India

LESSON

5

VB CONTROLS

CONTENTS

5.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Discuss message box and input box
- Discuss events, textboxes, labels, buttons, panels and group boxes

5.1 INTRODUCTION

This lesson include some vb tools such as message box, events, textboxes, buttons, labels, Panels and group boxes, etc. All these tools are discussed in the following sections respectively.

5.2 MESSAGE BOX

5.2.1 Message Box for Information

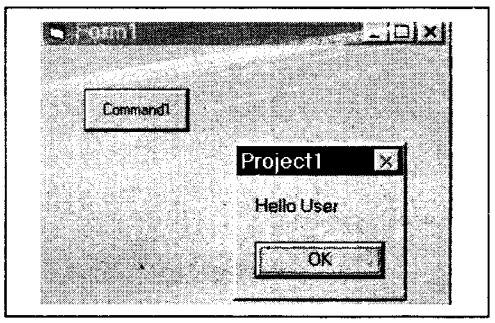
This pop-up box is seen very commonly in Windows. It is used to inform the user of some error, such as when copying a file, reading a floppy disk, insufficient memory, etc. For displaying the message box, we pass some string argument or any other data that we want to show. This data can be a literal, a constant or a variable too. The general syntax of the message box is:

MSGBOX (prompt, [buttons, title]) where,

- 'Prompt' is the message to be displayed.
- 'Buttons' is the button combination to be displayed on the box.
- 'Title' is the title bar text of the message box.

For example, this line will generate a message box with a very simple message 'Hello User':

MSGBOX "Hello User"



A message box is a Modal entity: once it is displayed, further execution of the current project stops. Till the time the box is being displayed, no other code of the program will be able to execute. Of course, other Windows programs will not be affected by the message box - only the current project will come to a halt. Once we click on the 'OK' button, the box will disappear and the program will continue from the line written after the message box line.

We use the message box to display some message alongwith a value, as:

MSGBOX "Hello Mr./Ms." & str_name

5.2.3 Adding Punch to the Message Box

We can include special graphics in our message boxes, making the message more lively and informative. For this, VB offers us the following options:

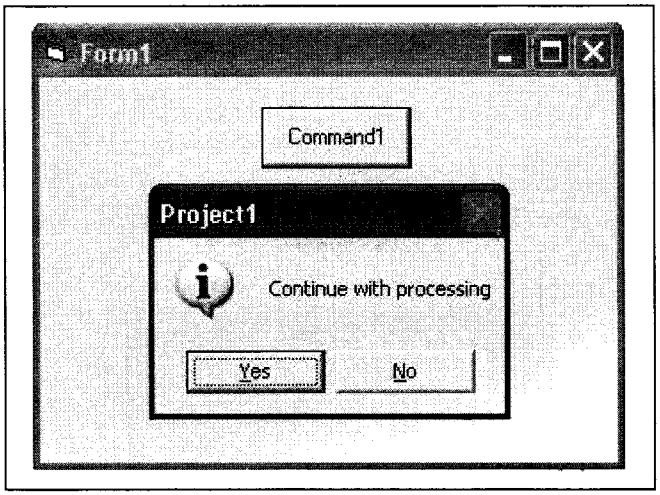
VB Button Name	Effect
Vbcritical	Messagebox with a red cross
Vbinformation	Messagebox with a blue coloured symbol 'i'
Vbexclamation	Messagebox with yellow triangle and an exclamation mark

We will now modify our earlier code listing message.1, to incorporate the vbinformation button, as follows:

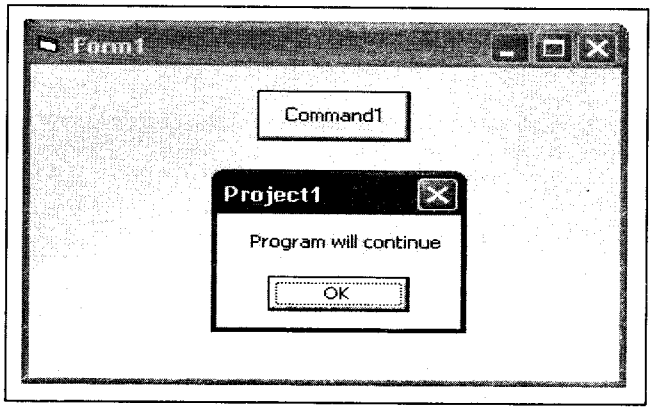
Code Listing message.2

```
PRIVATE SUB command1_CLICK ( )
1. DIM ans AS VB MSG BOX RESULT
2. ans = MSGBOX ("Continue with processing?", VBYESNO + VBINFORMATION)
3. IF ans = VBYES THEN
4. MSGBOX " Program will continue"
5. ELSE
6. MSGBOX "Program will stop"
7. END IF END SUB
```

The only change in the code is on line no. 2, incorporating the vbinformation button. The message box is now displayed in this manner:



Suppose we click on 'Yes', we get this output:



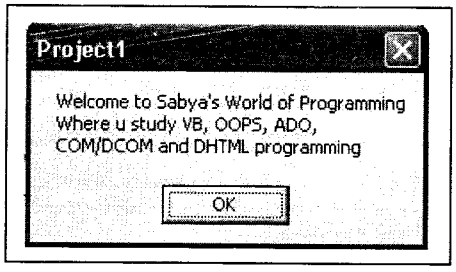
5.2.4 Message Box with a very long Message

If we display a very long message, the message box will become very wide, looking very odd and unattractive. To get around this problem, we build a string with the linebreak character built into it. This is achieved by the vbcrlf statement. (Vb carriage return line feed) Here is a small code for the same:

Code listing message.3

```
PUBLIC SUB command1_CLICK ( )
1.  DIM  str_long_msg  AS  STRING
2.  str_long_msg =  "Welcome to Sabya's World of Programming"   &  VBCRLF
3.  str_long_msg =  str_long_msg & "Where u study VB, OOPS, ADO,"  & VBCRLF
4.  str_long_msg =  str_long_msg & "COM/DCOM and DHTML programming"
5.  MSGBOX  str_long_msg
6.  END  SUB
```

Line 3 builds-up (concatenates) further on the string in line 2. Similarly, line 4 adds (concatenates) further on the string in line 3 through the concatenation operator [&]. The output will come in three lines as shown



5.3 INPUT BOX

The message box is ideal for information-display and for Yes/No choices. For more elaborate entries, we have the dialog box named Input Box. The Input Box is like a message box with a textbox at its bottom. The syntax is:

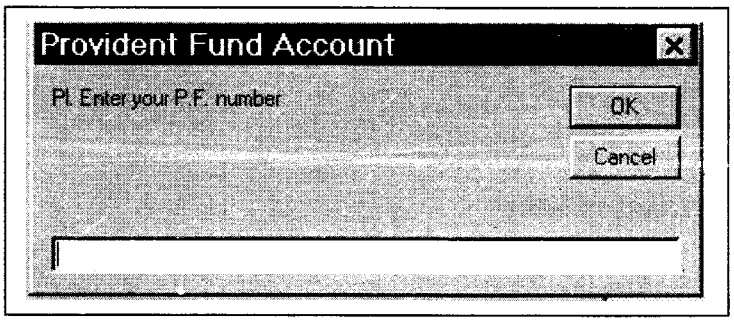
Variable_name = INPUTBOX (prompt, [title])

Here is a sample code to accept data through an Input Box:

Code listing message.4

```
PUBLIC SUB command1_CLICK ( )
1. DIM  str_num1  AS  STRING
2. str_num1  =  INPUTBOX  ("Pl.  Enter  your  P.F.  number  ",  "Provident  Fund
   Account")
3. MSGBOX  " Your P. F. Number is ..  "  &  str_num1
4. END  SUB
```

The output of this code is shown here:



The Input Box gives the 'OK' and 'Cancel' buttons by itself. We enter the value in the textbox at the bottom and click on 'OK" or press the enter key. This value is trapped in a variable for further action. This is an excellent choice for data that is to be entered infrequently. Compared to a TextBox, the Input Box uses less space, and grabs the user's attention immediately as it pops-up on the screen.

5.4 FORM EVENTS

5.4.1 Initialize/Terminate

The initialize is the first event that fires in the lifetime of a form, just as the form is 'beginning to take shape'. Hence, this is not the right place to deal with the controls on the form. Those controls might not have been 'sited' on the Form yet. Thus, their properties, etc., will not be available in this event - if we access these properties in the initialize event, we will get an error. It will be a better idea to wait for the other events that follow. However, this event is the right place for initializing our variables that might be used later in the program.

The terminate is the reverse of the initialize event - the last event in a form's lifetime. This is fired when the form object is being 'destroyed'. This event, thus, is used for cleanup code, to explicitly release resources being taken up by the form. The terminate can be called explicitly through code, as follows:

SET form1 = NOTHING

5.4.2 Load/Unload

The load event follows the initialize event of the form. At this time, the form's. Form file is brought into the RAM. In this event, controls begin to appear on the form, though not all of them. Thus, this is still not the perfect place to deal with some of the controls on our form. The Data Control's recordcount property, for instance, gives an error if referred to in this event.

The unload event fires when we close the form, by clicking on the close button at the top-right of the form. This event is followed by the terminate event. The variables, etc. associated with the form lose their values once the unload event is fired. This event can be initiated programmatically by the following line:

UNLOAD form1

The unload event has an in-built parameter, cancel. This parameter can be used to 'cancel' the unload event, if required. This code will help us check if the user really wants to close the form. If the user says no, the form will be not being unloaded.

Code Listing event.1

```
PRIVATE SUB form_UNLOAD (CANCEL AS INTEGER)
1.  DIM  ans  AS  VBMSGBOXRESULT
2.  ans  =  MSGBOX (" Sure to close the form? ", VBYESNO, " Please confirm ")
3.  IF  ans  =  VBNO  THEN  CANCEL = TRUE
END  SUB
```

Here is how the code works:

1. Line no. 1 declares a VBMSGBOXRESULT type of a variable.
2. The line no. 2 'confirms' from the user whether he really wants to unload the form, through a message box with the yes/no buttons.
3. If the user clicks on the 'No' button, the ans variable will have the value VbNo. If so, the cancel parameter of the form_unload event is set to 'true'. This forces the event form_unload to be 'cancelled'.

5.4.3 Queryunload

This event is fired just before the unload event. This event carries an extra parameter compared to the unload event. This is the unloadmode parameter. Using this parameter, we can come to know how the unload event was fired - through code, through the close button, etc. This way we can have precise control on how the form should be unloaded. If required, based on the triggering event, we can 'cancel' the unload operation.

The various options of the unloadmode available to us are:

INTEGER VALUE	VB CONSTANT	WHAT IT MEANS
0	VbFormControlMenu	The 'close' button was pressed.
1	VbFormCode	The form is being unloaded through code.
2	VbAppWindows	The Windows session is coming to an end.
3	VbAppTaskManager	The task manager is closing the application
4	VbFormMdiForm	An MDI Form is causing unload of the Form.

This code checks for the source of the unload event. If the user is closing the form by pressing the 'close' button, the form will not be unloaded.

Code Listing event.2

```
PRIVATE SUB form_QUERYUNLOAD (CANCEL AS INTEGER, UNLOADMODE AS INTEGER)
IF UNLOADMODE = VBFORMCONTROLMENU THEN CANCEL = TRUE
END SUB
```

5.4.4 Activate/Deactivate

Load and Unload are useful events. However, there is a small problem - these events fire only once in a form's lifetime. Thus, they cannot be used if we frequently shift from one form to another and we want to keep track of variables. If our application requires that data on form1 be seen on form2 whenever we switch from form1 to form2, the load event will not give the desired results at all times; it will give the correct result only the 1st time - the only time the load event fires. For situations such as this, we need to use the activate/deactivate event combination.

The activate event fires whenever we shift from one form to another form in the same application. Suppose we shift from form1 to form2, then:

1. The activate event of form2 is fired.
2. The deactivate event of form1 is fired.

When we shift back to form1 from form2 then:

1. The activate event of form1 is fired.
2. The deactivate event of form2 is fired.

As an example, consider the following code. Assume we have written this code in the activate event of form2. Here, we are assigning the value of a public variable pub1 of form1 to the textbox text1 of form2.

Code Listing event.3

```
PRIVATE  SUB  FORM_ACTIVATE ( )  '  in form2
text1.TEXT  =  form1.pub1 ' public variable declared in form1
END  SUB
```

Now, whenever we shift to form2, the activate event will assign the value of pub1 to the textbox text1 of form2. Thus, we can use the value of one variable in more than one VB Forms, without worrying about any details of transferring the values manually.

5.4.5 Gotfocus/Lostfocus

The gotfocus event fires when the blinking cursor (caret) 'enters' it. This is the stage when the control will receive the keyboard input and the mouse click event. The gotfocus fires whenever the user uses either the tab key to 'enter' a control or clicks the mouse on a particular control.

It is important to note that the gotfocus event of the form might never be fired, because the form can only receive the focus if:

- There are no controls on the form.
- There are no controls on the form which can receive the focus.

As should be very obvious, both the conditions mentioned above are practically impossible. Thus, it is not really possible for a form's gotfocus event to be fired.

5.5 TEXT BOX

We use the textbox to enter data as well as display data. Thus, the textbox can accept numbers, alphabets, dates, etc from the user. These values can then be stored in the database, if required. Alternatively, we can also use the textbox to show the results of some processing requested by the user, such as displaying data fetched from the database, or showing the result of some mathematical operation.

The most important properties of the textbox are as follows:

Alignment: This property determines the alignment of the contents (text) inside the textbox. Three possible values are:

Value	Meaning
0	Left aligned (default)
1	Right aligned
2	Centre aligned

BorderStyle: Determines whether a textbox will have a 'border' around it or not. The possible values are:

- 0 - None (No border).
- 1 - Fixed Single (Border is present, Default Value).

Hide Selection: When we select some text using either the mouse or the keyboard, the 'selected item' becomes highlighted. Once we click anywhere else, the 'selected item' is lost and the highlight around the selected item disappears. If we want the selection (highlight) to persist even if we click anywhere else on the form - or till the time we make a new selection - we can set *hide selection* property to false. This property is Boolean and default value is 'true'. That is, the 'selection' is lost the moment we click on any area after making our selection. If we want the selection to persist, we can set the *hide selection* property to 'false'.

Max Length: Limits the number of characters a user can type. The default value is 0, allowing approximately 32 KB of data. This property only controls the quantity, not the quality of the data. Thus, *maxlength* cannot be used for accepting only numbers or only alphabets.

Multiline: 'False' by default. As we enter our data, text longer than the width of the textbox scrolls from right to left, i.e., horizontally. Once *multiline* is set to 'true', the excess data will scroll from top to bottom, i.e., vertically. Once multiline is set to true, we can use the 'Enter' key to insert line breaks in our data.

- ***Password Char***: Any character in the textbox can be clearly seen by anyone. If we want to, we can 'encrypt' our typing into a specific character. This way, no one can see what we are typing. Using this property, we can implement a password-protection in our application. For example, we can type an '*' (asterisk) symbol for the *Password Char*. Once we do this, any text entered by the user will be displayed as '*'. Please note that this property only changes the way the character is 'displayed'. The actual character still remains the same, which we can read from code.
- ***Scroll Bars***: We commonly see scrollbars in Windows Explorer, Internet Explorer, PowerPoint, etc. The same scrollbars can also be attached to our textbox. The default is set to 'no scrollbars'. In this situation, we can use either the arrow keys or the 'home' and 'end' keys for moving around in a big textbox. Once we set the property to use scrollbars, the scrollbars will become 'active' (or enabled) as soon as we type some extra text. We can choose from the following values:

Value	Meaning
0	No Scroll Bar (default)
1	Horizontal Scroll Bar
2	Vertical Scroll Bar
3	Both Horizontal and Vertical Scroll Bars

Text: The most important property for textbox. The textbox is supposed to accept and display data. The *text* property helps us perform that task: it is used to take in data and to give out results. The *text* property can accept any kind of data - numbers, strings, dates, etc. *Text* property is the default property of the textbox. This means that we can refer to it through code even without explicitly (directly) mentioning it. Thus the lines

```
text1.TEXT = "Some value"
```

and

```
text1 = "Some value"
```

will give us the same result, and no error will be generated either.

Rich Text Box

The text box provides features for entering and displaying text. It provides options for changing colours, fonts, etc. However, these changes are always global, i.e., they apply to all the contents of the text box. If we want to apply these changes to a specific portion of text, say, for font size, colour, paragraph indentation, etc., we are unable to do this.

These limitations of the text box are overcome by the rich text box. As the name suggests, the rich text box is a text box with a 'rich' set of features. We can set the colour options for a few words, change the font type and size for a select few words, set the indentation for a particular paragraph, save the contents as a file, load a file from the disk into the rich text box, etc. In fact, these features are all provided by MS-Word and other common word-processing software.

Now we'll draw the control on the form and make a small application using it. To work with the richtextbox, we use the following properties:

Use this property	To achieve this
SelBold	Making selected text bold
SelItalic	Making selected text italic
SelUnderline	Making selected text underlined
SelStrikethrough	Giving a strikethrough effect of selected text
SelText	Getting the selected text and working on it.
SelBullet	Generate a bulleted list
SelRtf	Getting the RTF code of the selected text.
SelStart	Setting the position of 1st character of selected text.
SelLength	Getting the length of the selected text.

Here are some examples of using these properties:

Save File

Saves the contents of the RichTextBox to a disk file. This file can later on be retrieved by us, using the standard Windows methods.

The syntax is as follows:

```
controlname.SAVEFILE [FILE NAME, FILE TYPE]
```

where,

- Filename is the name we want to give to the file (the full path needs to be given).
- File type is the format in which to save the file. *It can have two values:*

VB Constant	Value	What it means
RtfRTF (Default)	0	Saves file in RTF format
RtfText	1	Saves file in Text format

Load File

Loads a disk file into the RichTextBox control at runtime.

The syntax is as follows:

```
controlname.LOADFILE [FILE NAME, FILE TYPE]
```

To make a selection of the file to be loaded at design-time stage itself, we set the filename property to an appropriate value.

5.6 LABEL

This control is for read-only data. Consider the following details:

The number of records in a particular file,

- The result of an income tax calculation,
- The number of days the students/employees have been present.

In the normal case, all of these values come from some internal calculations. Thus, they are all read-only data. The user should not be allowed to directly change these values using the keyboard. For this reason, they need to be shown in a label control.

The latch label can also serve as a prompt, helping the user understand what is to be entered in which textbox or listbox

Link Label

Link Label displays a hyperlink. Multiple hyperlinks can also be used and each hyperlink can perform a different task within the application. Properties of the Link Label control are the Active Link Color, Link Color and Link Visited which are used to set the link color.

5.7 OPTION BUTTON/RADIO BUTTON

We use this control very commonly in Windows. It is a small hollow circle, which helps us to make a unique choice from a host of options. Consider the screen we get when we begin to shutdown our system: rounded options for 'restart', 'shutdown' and 'restart in MS-DOS mode'. We can make only one selection from the choices available to us. These are the option buttons (seen as white circles). We can click on one of these circles, which give us a 'dot mark' ('●') if it is not already there. We can choose only one of the options we like, at one time. Thus, for making unique selections from a given set of options, we use an option button.

The important properties of the option button are the same as the checkbox. Hence, we won't be discussing them here. You can always refer to the checkbox section and find out the details. Some properties, which have a different 'set' of values compared to the checkbox are mentioned here:

Value: The meaning of this property is the same as that of a checkbox. In an option button, this property is boolean. Thus, the possible values are:

- 0 - False (Default)
- 1 - True

If the *value* is set to 1 and the *style* is 'graphical', the option button will appear as a 'pressed' command button, slightly depressed (similar to what we see in MS-Word, when we press the 'bold' or the 'italics' button on the toolbar).

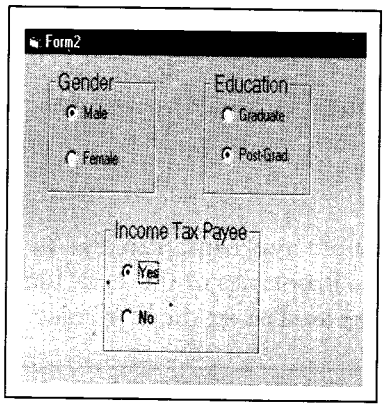
Grouping Option Buttons

With a mutually exclusive selection, we can face a problem: what if we need more than one 'group' of option buttons on our form? For example, consider a form with options of:

- gender : male/female

- education : graduate/post-graduate
- income tax payee : yes/no

Quite clearly, we will need three groups of option buttons, and all three groups should be mutually exclusive. For such a situation, we will draw our option buttons on another control such as a 'Frame' or a 'Picture Box' which has been put directly on the form. We will not put the Option Buttons directly on the form. For this reason, the Frame and the Picture Box are also known as a *'container control'*. Now, we can easily treat the three options as three distinct groups. The following figure shows the three groups, in three frames. Note that one button from each group has been selected:



5.8 CHECK BOX

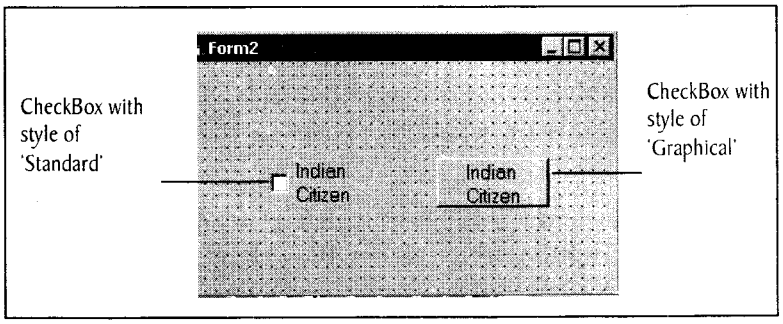
Used very commonly in Windows, it is a small hollow square box which helps to make multiple choices from a host of options. We see these boxes when we begin formatting a disk, asking us if we want to display the volume label, summary information, etc. We can click on these boxes, which gives us a 'check mark' ('√') if it is not already there. If the check mark is already present, clicking once again will make the check mark disappear. We can choose any number of options we like, at the same time. Thus, for making multiple choices from a given set of options, we use a check box.

The important properties of a check box are:

- ***DisabledPicture:*** Tells VB which image has to be shown on the check box when it is 'disabled', i.e., when the 'enabled' property is set to 'false'. Any kind of an image file can be used (.bmp, .gif, .ico, etc).
- ***DownPicture:*** This property tells VB which image has to be shown on the check box when it is in the 'down position', i.e., when the check box has been 'checked'.
- ***Picture:*** Sets the image to be shown on the check box when it is in the normal state.
- ***Style:*** This property determines whether the check box will appear as a normal check box ('standard') or like a command button ('graphical'). The possible values for this property are:
  - 0 - Standard (Default)
  - 1 - Graphical

The next picture shows two check boxes with the caption 'Indian Citizen'. The *style* property of check box on the left is the default 'standard'. The *style* property of check box on the right is set to

'graphical'. As can be seen from the picture, the check box on the right looks like a simple command button.

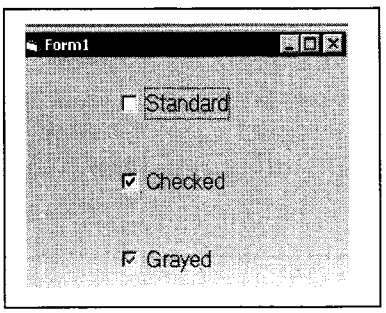


***Value*:** By default, a check box on the form is in the 'unchecked state'. That is, the check box is not checked by default: we have to click on it to make it checked. In case we want a particular Check Box to start with a 'checked state', we set this property to an appropriate value as follows:

Value	Meaning
0	Unchecked (Default)
1	Checked
2	Grayed

A check box with a 'Checked' *value* will start off with a check mark on it by default, even when the user has not clicked on it.

A Check Box with a 'Grayed' *value* will appear as a 'Checked' checkbox. However, it will also have a 'Grayed-out' effect, similar to the effect we see in Windows when a particular item or choice is 'not available', i.e., not enabled. The results of the three settings are seen here, at run-time:



The checkbox doesn't have any unique important method. The general ones have already been covered under the common methods in VB.

5.9 PANELS AND GROUP BOXES

Panels are the controls that contain other controls within it. It acts as a container for group of controls, for example, a set of radio buttons, checkboxes, etc. By using panels, we can give one single name for group of controls.

Groupboxes are similar to the panel control and is used to Group controls. Groupboxes display a frame around them and it allows to display captions to them which is not done with the Panel control. The Groupbox class is based on the Control class. Panel is container for group of controls like groupbox.

Both panel and groupbox acts as a container to other controls. They help a lot in some applications where we want a group of controls or where we want to make the objects disabled or enabled according to the specified task.

The main difference between panel and group box is that group box has a place where u can place a text of your own ie it contains caption which is not possible in panel. A panel is just like a frame where we can provide a collective name. Also panel has a scrollbar where as group box does not contain scrollbar.

For adding controls to the panel, drag a Panel (Panel1) to the form from the toolbox. For placing some controls, say, checkboxes on this Panel drag three checkboxes from the toolbox and place them on the Panel. All the checkboxes in the Panel are together as a group but they can function independently.

Check Your Progress

1. What is the effect of vbinformation button on the message box?
2. What is Load/Unload event?

5.10 LET US SUM UP

The VB.NET message box function is very similar to that of old version, but the way you use it is slightly different.

Most Windows applications request for user input. Message box information is used to inform the user of some error, such as when copying a file, etc. The Input Box is like a message box with a textbox at its bottom. The initialize is the first event that fires in the lifetime of a form. The terminate is the reverse of the initialize event - the last event in a form's lifetime the load event follows the initialize event of the form. At this time, the form's. From file is brought into the RAM. The unload event fires when we close the form. We use the textbox to enter data as well as display data. Radio button is a small hollow circle, which helps us to make a unique choice from a host of options. The main difference between panel and group box is that group box contains caption which is not possible in panel.

5.11 KEYWORDS

Initialize: The initialize is the first event that fires in the lifetime of a form.

Terminate: The terminate is the reverse of the initialize event.

Gotfocus: The gotfocus event fires when the blinking cursor (caret) 'enters' it.

5.12 QUESTIONS FOR DISCUSSION

1. What is a Message box? In what ways is the syntax of the message box is different from the one in earlier versions?

2. Differentiate between panel and group boxes.
3. How are grouping options used in radio button?
4. What is rich text box? How the file is loaded in it?

Check Your Progress: Modal Answers

1. vbinformation button shows a message box with a blue coloured symbol.
2. In load event, controls begin to appear on the form, though not all of them. Thus, this is still not the perfect place to deal with some of the controls on our form. The unload event fires when we close the form, by clicking on the close button at the top-right of the form.

5.13 SUGGESTED READINGS

Sanjeev Sharma, *Visual Basic 6*, Excel Books

A. Chakraborti et al., *Microsoft.Net Framework*, PHI, 2002

M. Reynolds et al., *.Net Enterprise*, Wrox/SPD, 2002

Vikas Gupta, *.Net Programming*, Dreamtech Publication

Mackenzie Sharkey, *Teach yourself Visual Basic.Net in 21 days*, SAMS Techmedia

Bible, Bill Evjen, Jason Beres et al., *Visual Basic.Net Programming*, Wiley India

UNIT III

LESSON

6

WINDOWS COMMON CONTROLS: MENUS, DIALOG BOXES

CONTENTS

6.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Discuss various tools that appears in the tool box such as list box, combo box, image list and picture box

- Discuss how menus are used
- Discuss dialog boxes

6.1 INTRODUCTION

The toolbox helps us to build the GUI or front-end of our application. The controls that normally form part of the toolbox when VB starts are known as *intrinsic controls* in VB. We will study some *intrinsic controls such as list box, combo box,* picture box which are discussed in this lesson.

When we work with windows, menus are the basic tools used in any of the windows application. One of the prime features of a language, especially based on GUI concept, is ease of use and free flow of information between the user and the system. This is ensured by Dialog Boxes.

6.2 LISTBOX

We see lists very commonly in Windows. Of course, we use them in our daily life as well - a shopping list (!), a list of important days, a list of pending bills to be paid - telephone, electricity, etc. In a computer program, a list is a very convenient way to enter data. The list becomes even more useful when we have a fixed range of values to enter.

In Windows, we choose a value from a list by clicking the mouse on a particular item. The value we choose at run-time (the selected item) gets a highlighter bar around it, indicating that it has been 'selected'. This value can then be put into a label or a textbox. Whenever needed, we can make multiple selections from a listbox. This can be done in the typical Windows style, using either the 'Shift' key or the 'Ctrl' key. However, we cannot directly make any run-time entries - we either choose from the available options, or we don't choose at all.

A listbox offers the following benefits:

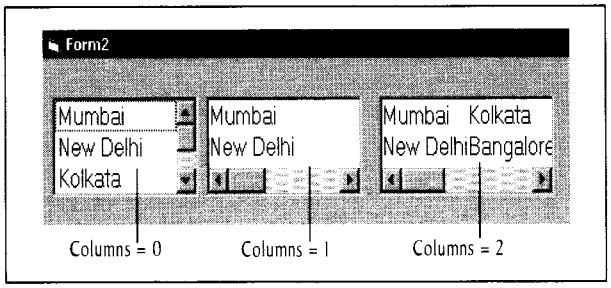
- Automatic validation of data, if the entries have been properly made in it.
- Less risk of typing errors.
- Saving of time, especially if the data is very bulky.

The following picture shows a listbox with an item 'Kolkata' selected. The blue band around Kolkata shows that it is the selected item in the list, which is a very common occurrence in Windows.

6.2.1 Important Properties of ListBox

Columns: Normally, the contents of the listbox scroll vertically. This property can help change the scrolling to horizontal. At the default setting of '0', the listbox items scroll vertically. Any value

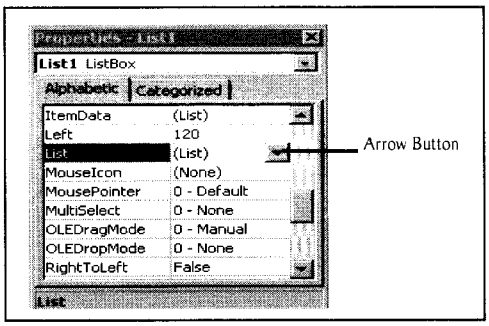
greater than this, makes the items scroll horizontally. If the value of *columns* is 1, only one column will be visible at run time. The remaining columns can be seen through the use of the horizontal scroll bar. If value of *columns* is 2, two columns can be seen at the same time. The other columns can be seen through the use of the scroll bars. Here is a picture that shows three listboxes at run time. They all have the same data, but the setting of *columns* property is different for each.



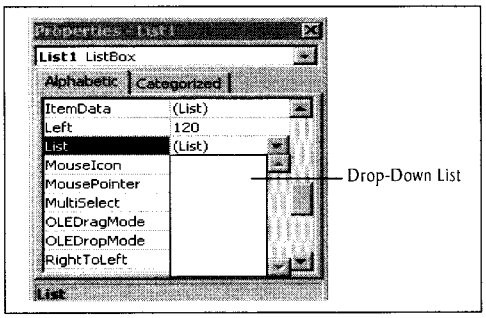
In this figure, the listbox on the left has the *columns* at the default of 0. So, it shows the data in a single column, using a vertical scrollbar. The listbox in the centre has the *columns* set at the value of 1. So, it is showing data horizontally, one column at a time, using a horizontal scroll bar. The listbox on the right has the *columns* set at 2. So, it shows the data horizontally, two columns at a time, using a horizontal scrollbar.

List: Probably the most important property. This property helps us to type in all the values into the listbox. At run time, the user will select from this list of options. To enter values into the list through the list property, we need to take the following steps:

1. Click on the listbox to select it.
2. Click on the '*list*' property in the properties window. A button with a downward-pointing arrow appears on the extreme right.



3. Click on the arrow. We see a 'drop-down list' appearing.



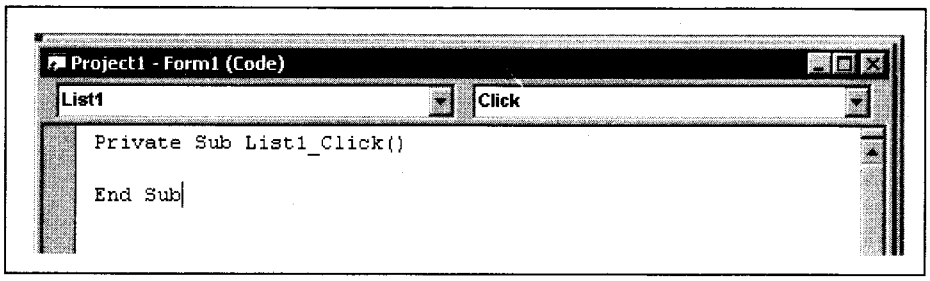
4. Type the first entry into the drop-down list.
5. Press 'Ctrl' + 'Enter' key. The cursor will come to the next line.
6. Repeat steps 4 and 5 to enter the list items one by one.
7. Once all data has been entered, press only 'Enter' key. The drop-down list disappears and data entered by us appears in the listbox. Our listbox is now ready to be used.

We will now make a small program. Suppose we enter this data in a listbox:

New Delhi, Mumbai, Kolkata, Chennai, Bangalore, Hyderabad.

When our program runs, we will 'click' on an item in the listbox. The item selected by us should appear in a textbox. Can you guess which *event* will we use for our program? The *click* event of the listbox. (remember *event-driven programming*?) That is, when the user *clicks* on the listbox, the selected item will be displayed in a textbox.

We will, first of all, make the form for our project. Start VB, open a new Standard EXE project and design a form with a Listbox, Textbox and appropriate labels. Then switch to the code window by double-clicking on the listbox. Concentrate on the two lines appearing at the top of the code window:



Once again, we will be writing our code between the lines

```
Private Sub List1_Click( )
   And
End Sub
```

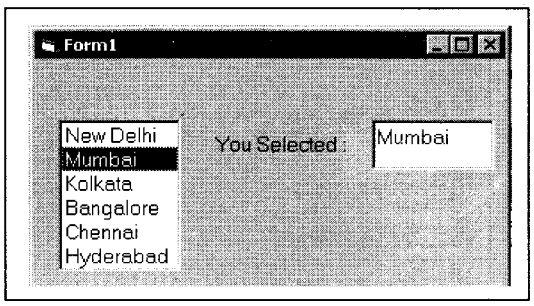
This code will run whenever we click on the listbox named 'list1' at run-time. The code is as follows:

Code Listing listbox.1

```
PRIVATE SUB list1_CLICK()
    text1.TEXT  =  list1.TEXT
END SUB
```

Now run the program, either by pressing the 'Start' button on the toolbar or the 'F5' key or by selecting 'Run' 'Start' from the menu bar.

When the program starts running, click on 'Mumbai' in the listbox. The output of code listing listbox.1 will be as follows:



It is to be noted that we have selected Mumbai from the listbox and the textbox is also showing Mumbai. This is because when we clicked on the listbox,

1. The *click* event for the listbox was fired.
2. The code written for the click event of the listbox was executed.
3. This caused selected item [list1.text] to be displayed in textbox [text1.text]

Now stop the program, either by Clicking on the 'stop' button on the toolbar or by clicking on the 'Close' button in the control menu of the Form. Again run the program, this time selecting New Delhi from the listbox. This time, we will see New Delhi displayed in the textbox.

ListIndex: This property gives the 'position' of the selected item in the listbox. Thus, if we click on the 1st item in the listbox, we get a *ListIndex* of 0, not 1. If we click on the 2nd item in the listbox, we get a *ListIndex* of 1, and so on.

This way, the maximum value of the *ListIndex* in a listbox with 'n' number of elements will be (n-1). For a listbox with 20 elements, thus, the maximum value of listindex will be (20-1) = 19. Also, remember that if no item has been selected, the value of listindex will be '-1'. The *ListIndex* property is only available at run-time. It cannot be seen in the properties window.

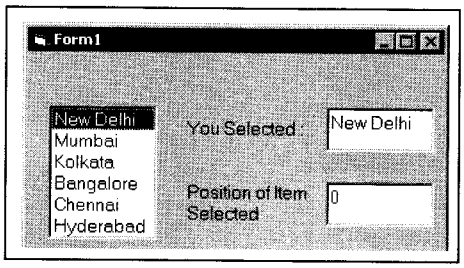
Now we will make another program. Similar to the code listing listbox.1, when we click on a listbox, we will get the selected item in a textbox. We will also get the *listindex* of the selected item in another textbox. For this, we will modify our previous form first of all. Open the form used in our previous exercise and modify it to get a form with another set of textbox and label.

We will now modify our code listing listbox.1. Switch to the code window and get the following code in:

Code Listing listbox.2

```
PRIVATE SUB list1_CLICK()
text1.TEXT  =  list1.TEXT
text2.TEXT  =  list1.LISTINDEX
END SUB
```

Now, whenever 'list1' is clicked, the code for the *click* event of 'list1' will be fired. We will see an output similar to this figure:



It is to be noted that we have selected New Delhi from the listbox and the textbox 'Text1' is also showing New Delhi. The textbox 'Text2' is showing the position of New Delhi (*listindex*) in the listbox. This is because when we clicked on the listbox 'list1', the following things happened:

1. The *click* event for the listbox was fired.
2. The code written for the click event of the listbox was executed. [Code Listing listbox.2]
3. This caused the selected item [list1.text] to be displayed in the textbox 'text1'. This value is presently 'New Delhi'.
4. The next line caused the 'position' of the selected item [list1.listindex] to be displayed in the textbox 'text2'. This value is presently '0'.

While the program is still running, click on another city name. We will see the selected text [list1.text] and also position of selected text [list1.listindex] change.

ListCount: This property tells us how many items are present in the listbox. The following line, for instance, will give us the total number of items present in the listbox 'list1'. The total count will be displayed in the textbox 'text1'.

```
text1.TEXT = list1.LISTCOUNT
```

ItemData: We can associate a number with every item that is typed in the listbox. This property is represented by *itemdata*. Thus, we can enhance the utility of the listbox, by including an additional numeric data for each item in the list. For example, we can have a list of the names of students in a class, and the *itemdata* can be used to represent their percent marks in the previous exam. As another example, consider a listbox with the names of districts of a state. The *itemdata*, in this case, can be used to show the population of each district.

The *itemdata* for a selected item can be read by passing the *listindex* of the selected item to the *itemdata* property. The general syntax is as follows:

```
list1.ITEMDATA  (INTEGER)  AS  LONG
```

Where list1 represents the name of the listbox and integer refers to the listindex of the listbox. Once the listindex is passed to *ItemData*, the value of itemdata can be displayed or can be used for some further processing.

For example, we can represent the order of the cities in terms of their population through the *ItemData* property. Thus, Mumbai can be given the number 1, Kolkata can be given the number 2, and so on. To do so, switch to the design window and follow the same sequence of steps as we have done for entering the *list* property of the ListBox. When the drop-down list for *itemdata* appears, we will see that VB has already assigned a default *itemdata* value of '0' for all the items in the ListBox.

Suppose, we have given following values as the *ItemData* and the *List*:

3 New Delhi

1 Mumbai

2 Kolkata

4 Chennai

5 Bangalore

6 Hyderabad

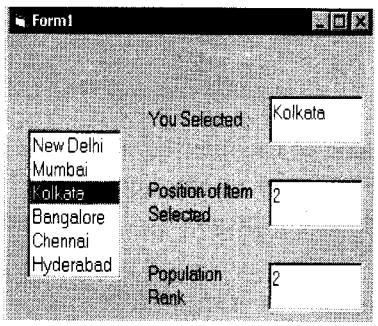
We will now modify our code listing listbox.2 so that when we click on an item, the selected text is shown in a textbox 'text1', the *listindex* is shown in another textbox 'text2' and the *itemdata* for the selected text is shown in another textbox 'text3'. Modify the form by adding another textbox and label to it. Our form should now appear like this:

This is the complete code, modified to accommodate the *itemdata*:

Code Listing listbox.3

```
PRIVATE SUB list1_CLICK( )
text1.TEXT = list1.TEXT
text2.TEXT = list1.LISTINDEX
text3.TEXT = list1.ITEMDATA (list1.LISTINDEX)   'new line of code
END SUB
```

Notice on the last line, we are passing *listindex* of 'list1' to *itemdata* of 'list1'. By doing so, the textbox 'text3' will display the *itemdata* of the selected text of the listbox. If you have typed the same data as given by me for the cities and the itemdata, the result will be as shown here:



As the figure shows, we have clicked on Kolkata. Thus, going by the code listing listbox.3,

- text1 is showing 'Kolkata' (list1.*text*).
- text2 is showing '2' (list1.*listindex* for Kolkata).
- text3 is showing '2' (list1.*itemdata* for Kolkata).

If we change our selected city, the other values will change accordingly.

MultiSelect: This property sets whether we can select multiple items from a listbox or not. The default value is for single selection. To allow the user to select more than one item at the same time, we have to set this property to a different value. The possible values for this property are:

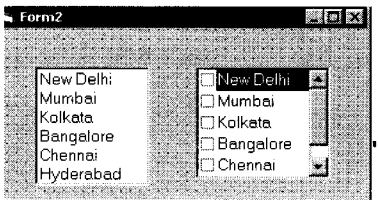
Value	Meaning	How to Select
0	None (default)	Not Applicable
1	Simple	Mouse Only
2	Extended	Mouse + Shift / Mouse + Ctrl

Sorted: Boolean property, 'false' by default. Thus, the list items will not be arranged in a proper order by default. If we want the listbox contents to be arranged in a proper order alphabetically, we set this property to true. In fact, the contents of the listbox are displayed in the order in which they have been entered in the *list* property. Once the *sorted* property is set to true, the items will be re-arranged and displayed in an ascending order, i.e., from smaller to higher values (based on the 'ASCII' value).

Style: Determines whether the listbox has checkboxes in front of all the elements or not. The possible values are

- 0 – Standard (Default)
- 1 – Checkbox

Checked List Boxes: By default, the listbox does not show any check boxes. Once the property is set to '1-Checkbox', checkboxes are displayed in front of all the elements of the listbox. The following figure shows a form with two listboxes. The listbox on the left has been kept at the default style property of '0-Standard'. The listbox on the right has been set at the style property of '1-Checkbox'.



List: The *text* property tells us the value of the selected item. The *list* is a property which is more of an array. By using the *list* property, we can refer to any item of the listbox, whether selected or not. This is the difference between the *text* and the *list* property, though both of them refer to the value of items in the listbox. The general syntax is:

```
list1.LIST  [ INTEGER ]  AS  STRING
```

Thus, the *list* property accepts an integer as the input and gives a string as the output. This integer will be the *listindex* and the output will be the text of the item with the specified *listindex.*

For example, if we use the listbox we have been using so far, the following line will give the output as 'New Delhi'.

list1.LIST (0) 'output is New Delhi, because listindex of New Delhi is 0

Similarly, if we give listindex as 2, we will get the output 'Kolkata'.

list1.LIST (2) 'output is Kolkata, because listindex of Kolkata is 2

NewIndex: This property gives the index of the item most recently added to the listbox. Consider the following line, which displays the listindex of the newest item added to the listbox, by using the *newindex* property:

MSGBOX " index of the new item is" & list1.NEWINDEX

The following line gives the *text* of the newly-added item to the listbox:

MSGBOX " text of new item is" & list1.LIST (list1.NEWINDEX)

6.2.2 Important Methods of ListBox

AddItem: Helps add an item to the listbox at run-time. We know that we cannot type directly into a listbox while the program is running. The set of values given at design-time is persisted with at run-time. This method offers an indirect way of adding an item, through code. The general syntax is:

```
listboxname.ADDITEM  ( item as STRING )
```

For example, this line will add the entry in 'text1' to the listbox 'list1'

```
list1.ADDITEM (text1.TEXT)
```

Clear: This property, as the name suggests, helps 'clear' the contents of the listbox, i.e., remove all the elements. The general syntax is:

```
listboxname.CLEAR
```

The *clear* method needs no parameters. The following line will clear off the contents of the listbox 'list1'

```
list1.CLEAR
```

RemoveItem: This property 'removes' items from a listbox, one-by-one. Thus, it is different from *clear*, which removes all the items at once. To mention the item to be removed, we give its *listindex*. The general syntax for the *RemoveItem* method is:

```
listboxname.REMOVEITEM  ( INDEX  AS  INTEGER )
```

The parameter passed to *removeitem*, i.e., *index,* refers to any integer value, the *listindex* value of the item to be removed.

This line will remove the item selected in the listbox 'list1'.

```
list1.REMOVEITEM (list1.LISTINDEX)
```

This code will remove the 1st item from the listbox:

```
list1.REMOVEITEM (0)  'listindex of 0 means the 1st item
```

The *index* we pass to *removeitem* must be present in the listbox, otherwise we will get an error. Suppose we have a listbox with 10 elements. If we pass an index value of 12 to the *removeitem* method, we will get an error, because this listindex is not present in the listbox. Thus, we must first check whether the index of the item to be removed exists or not. If it exists, we can remove it. If it does not exist, we can inform the user of the fact and 'skip' the code that removes the item from the list.

This code will do precisely that - it will accept a number from the user and check if that listindex exists in the listbox. If the index exists, the item will be removed. If it doesn't, the user will be given an appropriate message. We will be using the click event of a command button 'button1' for the program.

Code Listing listbox.4

```
    PRIVATE SUB button1_CLICK( )
1.  IF text1.TEXT  =  ""  OR  VAL ( text1.TEXT ) < 0 THEN EXIT SUB
    ' if no item entered or if improper item number, quit
2.  IF  VAL ( text1.TEXT )  <  list1.LISTCOUNT THEN
    ' if entered index is less than total items in the listbox
3.  list1.REMOVEITEM   VAL ( text1.TEXT )
4.  ELSE
5.  MSGBOX " Please give a value less than  "  &  list1.LISTCOUNT
6.  END IF
    END SUB
```

Here, the code runs as follows:

1. Line no. 1 checks if user has not entered any value [text1.TEXT = ""] or has entered a negative number [text1.TEXT < 0]. If any of the conditions is true, the program jumps out of the code routine [Exit Sub]. After all, there is no point in continuing with wrong data.
2. Line no. 2 checks if the value of text1 is less than the listcount. After all, the maximum listindex of a listbox will be one less than the listcount. Thus, if the user enters a value equal to or more than the listcount, an error will be raised (because that listindex will not exist in the listbox). If the value of text1 is less than the listcount, the program goes to line 3. If not, the program goes to line 5.
3. Line no. 3 removes the listitem whose listindex we entered in 'text1'.
4. Line no. 4: This line is the else part of the if-then-else block.
5. Line no. 5: This line displays a messagebox and informs the user of the proper value of listindex to be entered in the textbox 'text1'.

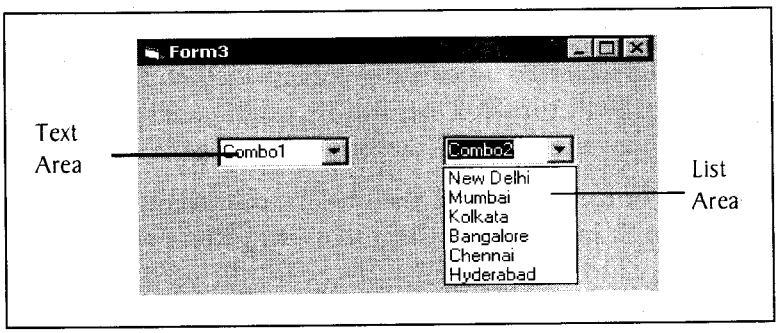
6.3 COMBOBOX

This control is a 'combo' (combination) of a listbox and textbox:

- Similar to the listbox, it helps us make a choice from a readymade list.
- Similar to a textbox, it helps us type in a new value.

However, this list is drop-down and not permanently open, unlike a listbox. Again, unlike a listbox, we cannot make multiple selections in a *combobox*. Generally, a Combo takes up less screen area compared to the listbox.

We make a selection by clicking on an item in the drop-down list. This is shown in the figure that follows. On the left, we see the combobox in the 'normal' state. On the right side, we see the combobox in the drop-down list state, showing its *listarea*. The selected item is shown in the *textarea* at the top. This item appearing in the textarea is the *text* property of the combobox.



If required, we can type a new item in the textarea. This item will not be added to the list of the combo, though. However, we can read it as the *text* of the combobox. This way, the user can temporarily get a new value as the *text*.

To see the ComboBox in action, design a form with a ComboBox and a command button. Add some items to the *list* property of the Combo. Now switch to the code window and type the following code. I assume that you have not changed the default *name* property of the command button and the ComboBox. Once the code has been typed, it should look like this:

Code Listing combo.1

```
PRIVATE SUB command1_CLICK( )
MSGBOX    "The text property is" & combo1.TEXT
END SUB
```

Now run the program and do the following:

- Click on the right of the combobox (on the arrow button). A drop-down list will appear showing the values entered in the *list* property.
- Click on any item in the drop-down list. The list will disappear. The name of the city clicked by you will appear in the textarea at the top.
- Now click on the command button. A messagebox will pop-up and show the *text* property of the combobox 'combo1'.

Now type a new value into the textarea of the Combo and click on the command button. See the result for yourself - though the new item has not been added to the *list* of the Combo, it appears as the *text* property. Now select another item from the Combo. The recently-typed text will be lost forever.

The important properties of the ComboBox are as follows. Properties similar to listbox are not discussed here. Only the unique properties have been mentioned.

Style: This reflects the way we can use our Combo. The possible options for *style* are:

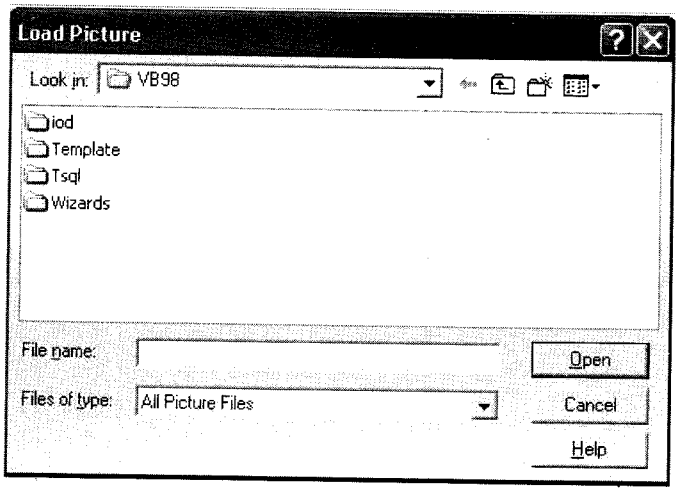
Value	VB Constant	Effect
0	VbComboDropDown (Default)	Normal Combo
1	VbComboSimple	Drop-Down feature is lost
2	VbComboDropDownList	We cannot type in TextArea

6.4 PICTUREBOX

Picture Box is one of the important controls for graphics. Picture Box control is used to display pictures. Similar to the frame, the Picture Box can also act as container control. The Picture Box has a Boolean property, autosize. If set to true, the PictureBox will change in size to fit the image perfectly.

Thus, in the Picture Box the image will stay in the original dimension. The PictureBox doesn't have the stretch property. To assign an image to this control, we set picture property. For this,

1. We select the control.
2. Select the Picture property.
3. Click on the ellipsis at the right, the "Load Picture" (Windows File Open) dialog box pops up.



4. Once we select the picture and click on "Open", the image is displayed in the control. To set the picture at runtime, we use the LOADPICTURE method, passing the filename as a parameter:

```
PictureBox1.PICTURE = LOADPICTURE(complete path of the image file)
```

Calling on load picture without any argument will remove the current picture from the control.

Here is a picture showing 1 Picture Box As we can observe, the picture in it is much bigger than the other controls. Other controls show the picture in original dimension.

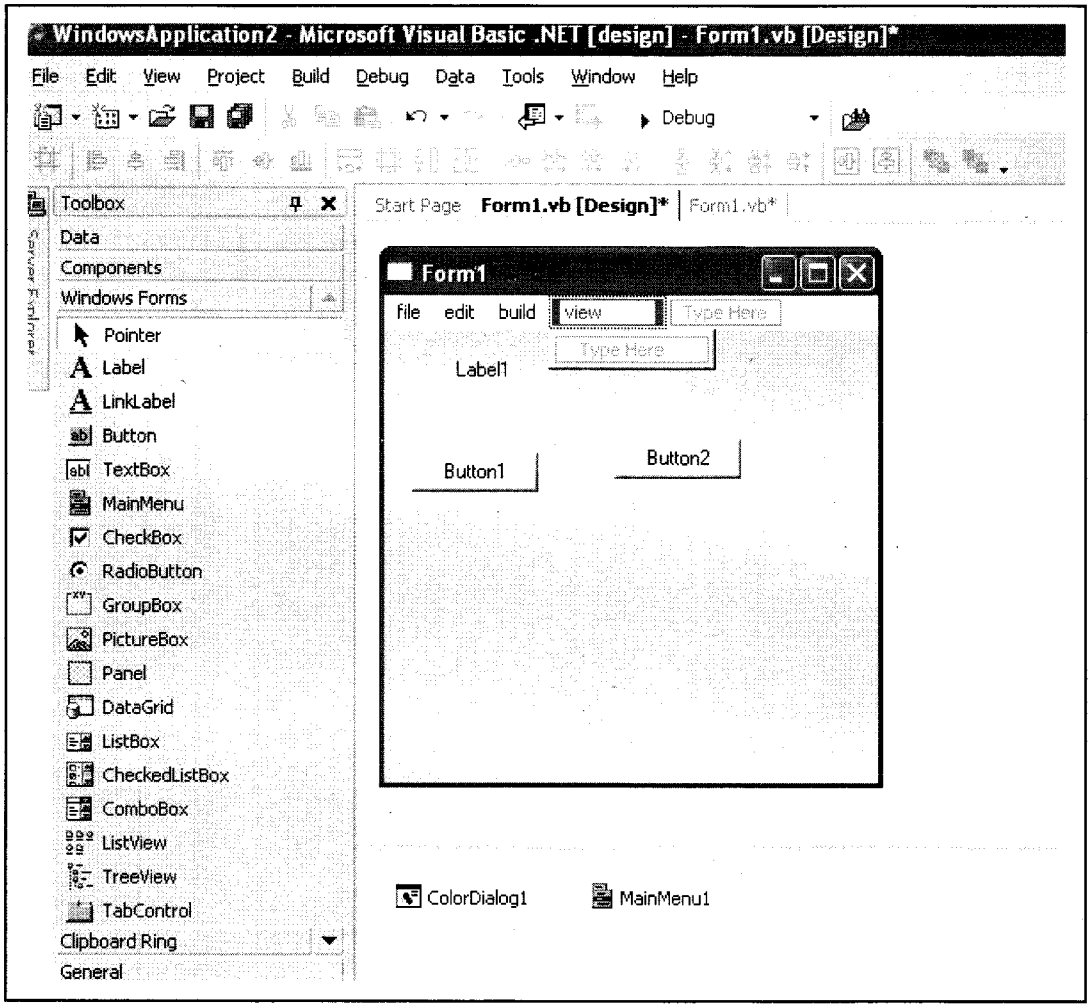
6.5 MENUS

Everyone is familiar with Menus. When we work with windows, menus are the basic tools used in any of the windows application. Examples of Menu items are File, Edit, Format etc. Menus in applications are those that allow us to make a selection out of the choices, when we want to perform some action with the application, for example, to format the text, open a new file, print and so on. In VB.Net. MainMenu is the container for the Menu structure of the form. Menus are made of MenuItem objects that represent individual parts of a menu (like File->New, Open, Save, Save As etc.).

The two main classes involved in menu handling are, MainMenu and MenuItem.

6.5.1 MainMenu

Windows users are familiar with Menu objects. The MainMenu control is the container for the menu structure of the form. Menus are made up of MenuItem objects that represent the individual parts of a menu. You can add submenus to menus that will pop up when the user clicks an arrow in the menu item, display check marks, create menu separators, assign shortcut keys to menu items, even draw the appearance of menu items yourself. The MainMenu class let's us assign objects to a form's menu class.



6.5.2 MenuItems

MenuItem is the class which supports the items in a menu system. Menus like File, Edit, Format etc. and the items in those Menus are supported by this MenuItem class. It's this MenuItem's click event that makes these Menus work. For a MenuItem to be displayed, we need to add it to a MainMenu object.

MenuItems in a MDI application work in a special way. When an MDI child window appears, its menu is merged with the MDI parent window. You can also specify how this menu is to be added to the MDI parent window with the MergeOrder and MergeType properties.

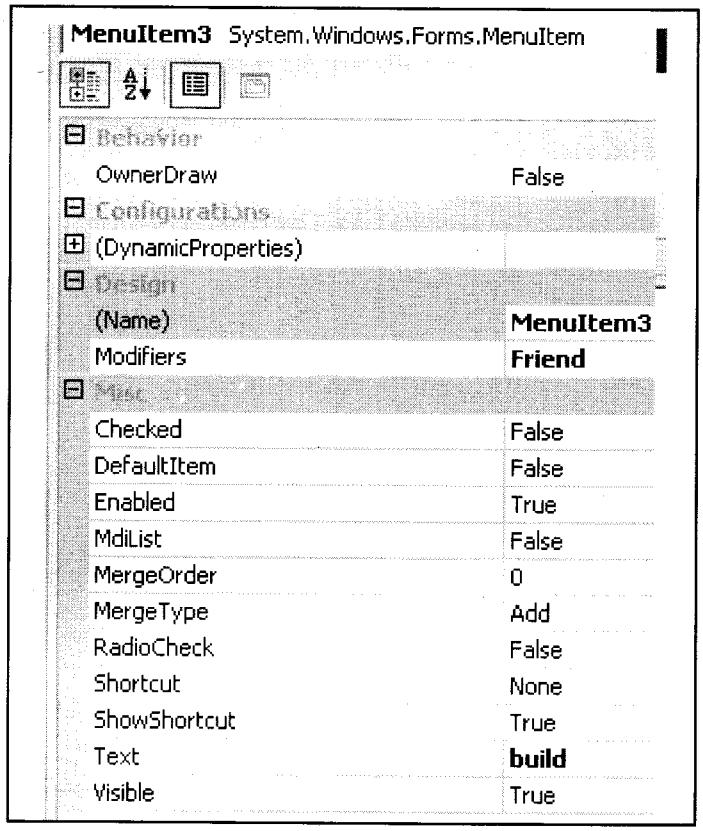
Event of the MenuItem

The default event of the MenuItem is the Click event which looks like this in code:

```
130
131     Private Sub MenuItem1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Menu
132
133     End Sub
134
135     Private Sub MenuItem3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Menu
136
137     End Sub
138 End Class
```

Properties of MenuItem

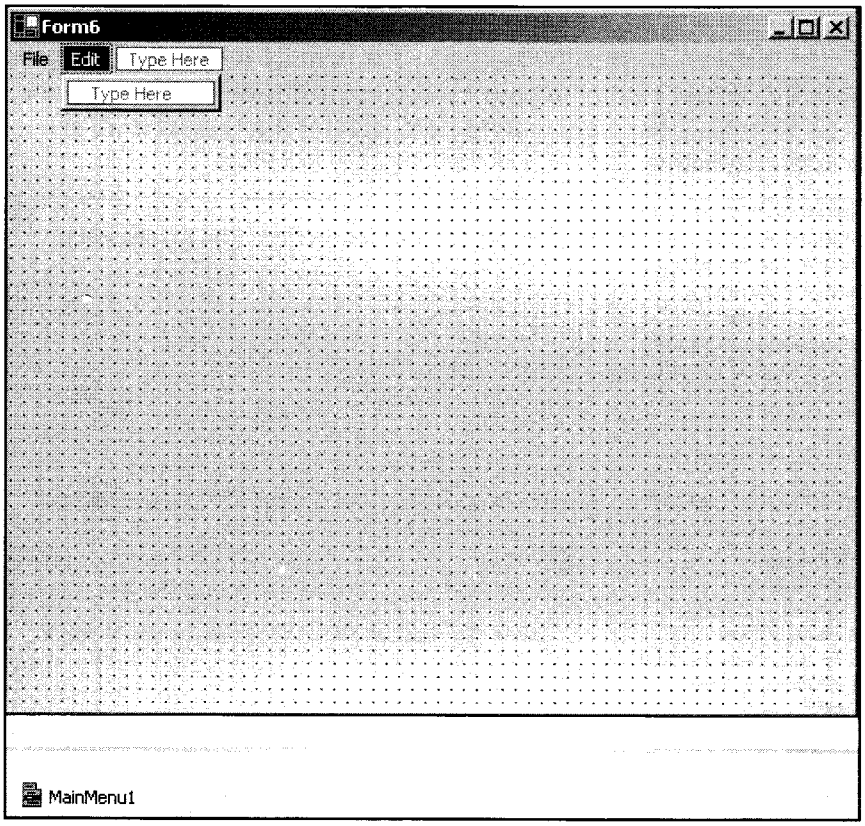
Few of the important properties of the MenuItem class are listed below:



- *Checked:* Default value is set to False. Changing it to True makes a checkmark appear towards the left of the Menu.
- *DefaultItem:* Default value is set to False. Changing it to True makes this menu item default menu item.
- *RadioCheck:* Changing it to true makes a menu item display a radio button instead of a checkmark.
- *Shortcut:* Enables to set a short cut key from a list of available shortcuts for the menu item.

6.5.3 Creating Menu

Creating Menus is simple. Drag a MainMenu component from the toolbar onto the form. When you add a MaiuMenu component to the form it appears in the component tray below the form. Windows form designer will add the MenuItem's for this by default, you need not add this. Once when you finish adding a MainMenu component to the form you will notice a "TypeHere" box towards the top-left corner of the form. To create a menu all you have to do is click on the "TypeHere" text which opens up a small textbox allowing you to enter text for the menu. You can view that in the image below. You can use the arrow keys on the keyboard to create a submenu or add other items to that menu or click on the first menu item and use the left/right arrow keys on the keyboard to create a new menu item. That's all it takes to add a menu to the form.



Working with an example

Let's work with an example to understand Menus. Drag a MainMenu and a TextBox onto the form. In the "Type Here" part, type File and under file type "New" and "Exit". Our intention here is to display "Welcome to Menus" in the TextBox when "New" is clicked and close the form when "Exit" is clicked. The Menu which we will create should look like this File->New, Exit (New and Exit below File). The code for that looks like this:

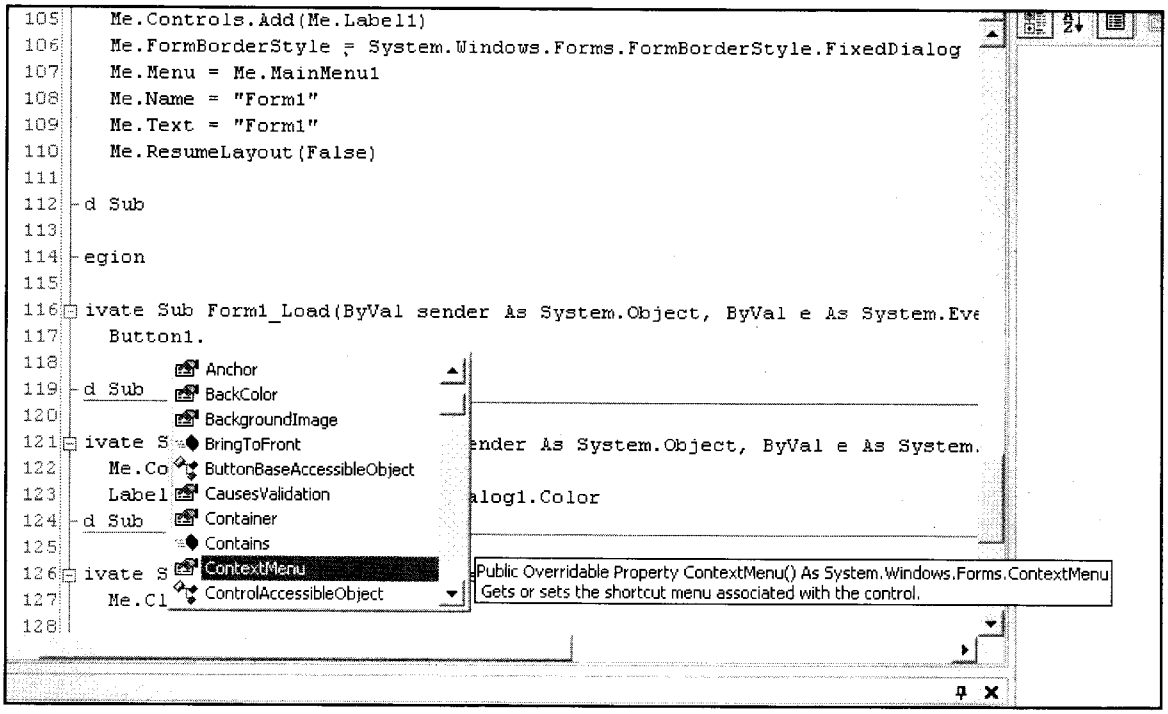
Public Class Form3 Inherits System.Windows.Forms.Form

```
#Region "Windows Form Designer generated code"
Private Sub MenuItem2_Click(ByVal sender As System.Object, ByVal e_
As System.EventArgs)_ Handles MenuItem2.Click
```

```
TextBox1.Text = "Welcome to Menus"
End Sub
Private Sub MenuItem3_Click (ByVal sender As System.Object, ByVal e As
System.EventArgs)_ Handles MenuItem3.Click
Me.Close()
'Me refers to the current object (form)
End Sub
End Class
```

6.5.4 Context Menu

Another popular type of menu is the context menu. The context menus are invoked by right clicking on another control. We can use context menus to display menus that will be specific for that control. You can achieve this by setting the ContextMenu property of the control with the name of the menu created separately as shown below



6.6 DIALOG BOXES

Most Windows applications request for user input. Dialog boxes are one means of requesting users for specific kinds of inputs. Therefore, VB.Net allows its designers to create a number of different types of dialog boxes. Standard Dialog boxes are included in classes that fall within the purview of the CommonDialog.

1. FileDialog
2. ColorDialog

3. FontDialog
4. PageSetupDialot
5. PrintDialog

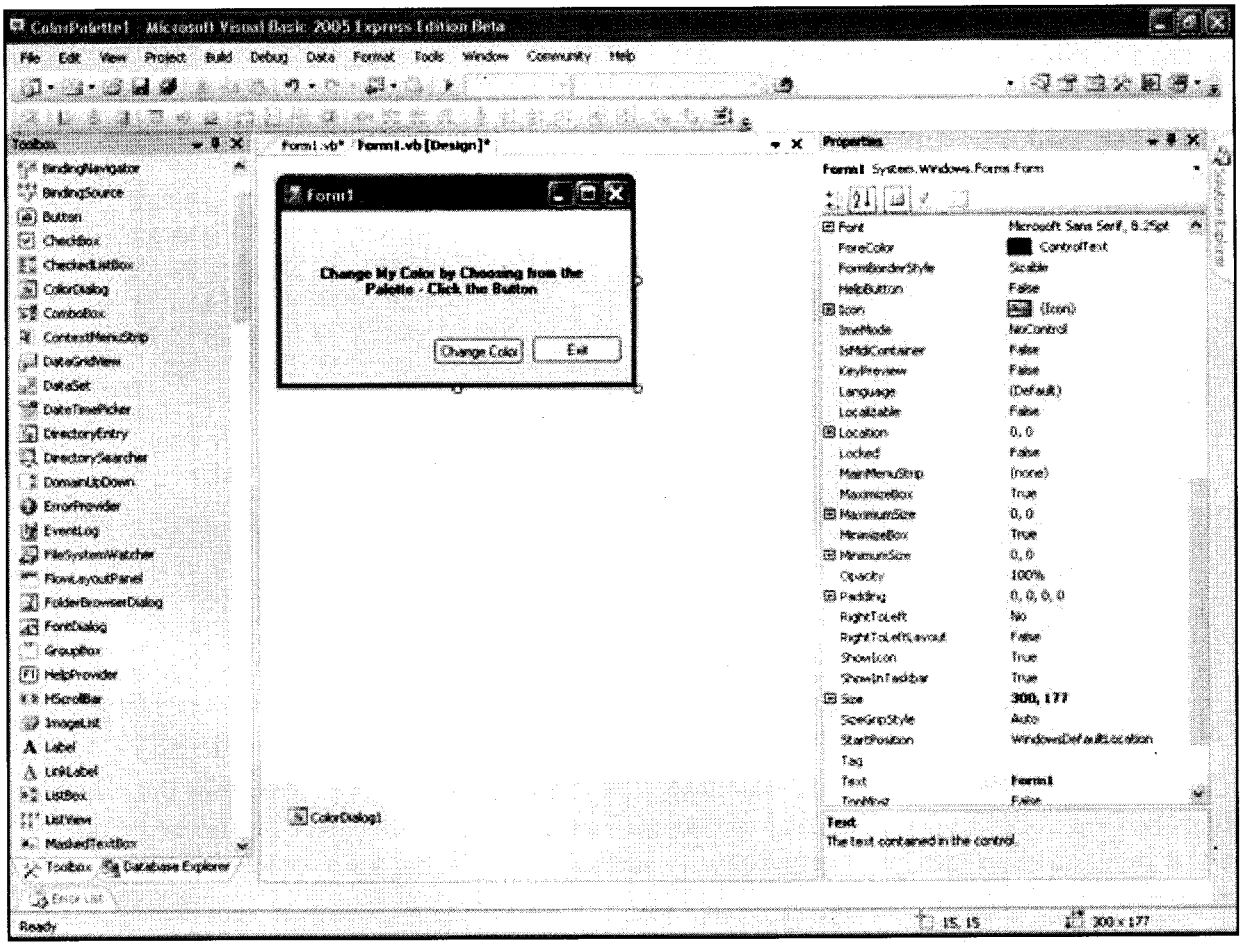
Let us now briefly study the features of the CommonDialog boxes

6.6.1 The ColorDialog Class

This dialog box shows the color palette for allowing user to select a color and add that color to the palette. The properties of the Class are given below:

Property or Method	Description
ShowDialog	Displays the dialog box
Color	Determines the color selected by the user
AllowFullOpen	Specifies if the user can add custom colors to the box
SolidColorOnly	Determines if the user can use dithered colors

Create a new Visual Basic Windows Applications project in the Visual Studio IDE. To the form Form1 add a ColorDialogBox. Add two Buttons and name them as given below:

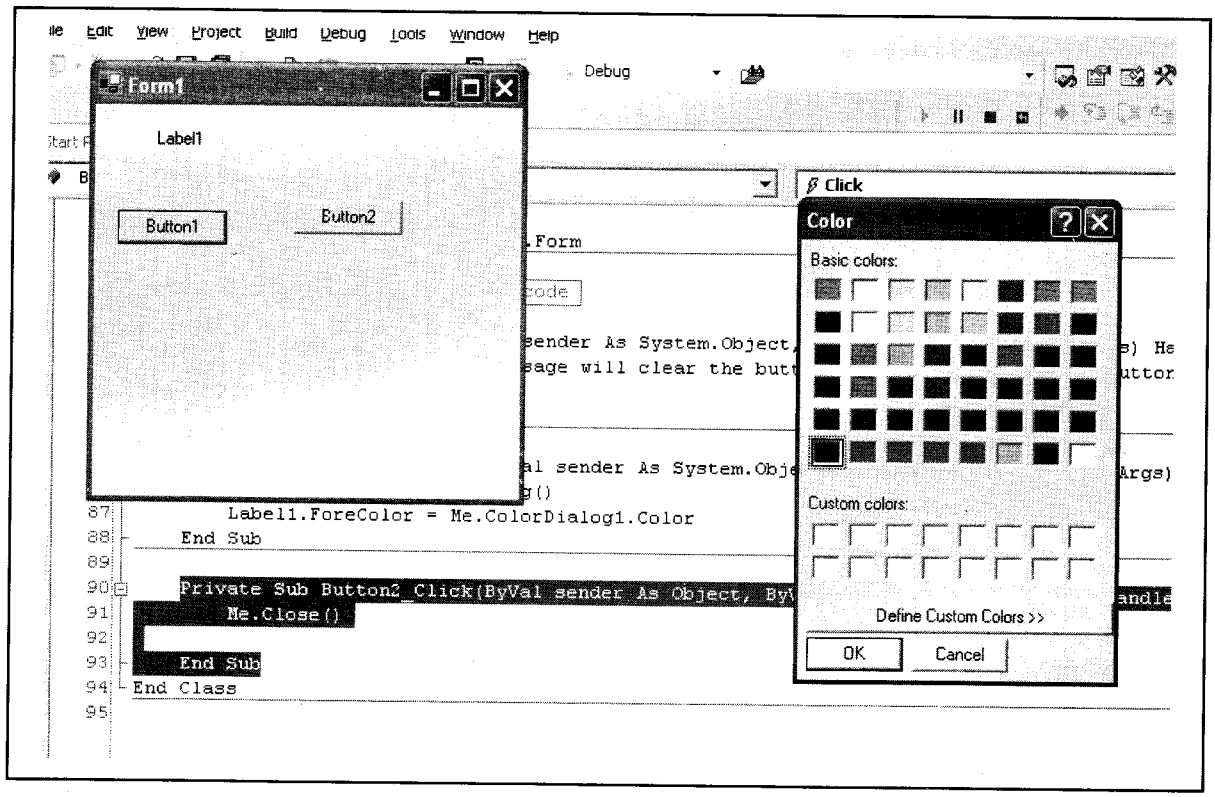


Now in the code behind form add the following codes:

```
Public Class Form1

Private Sub Button1_Click (ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click

    Me.ColorDialog1.ShowDialog()

    Label1.ForeColor = Me.ColorDialog1.Color

  End Sub

Private Sub Button2_Click(ByVal sender As Object, ByVal e As System.EventArgs)
Handles Button2.Click

    Me.Close()

  End Sub

End Class
```

Press F5 to execute the program

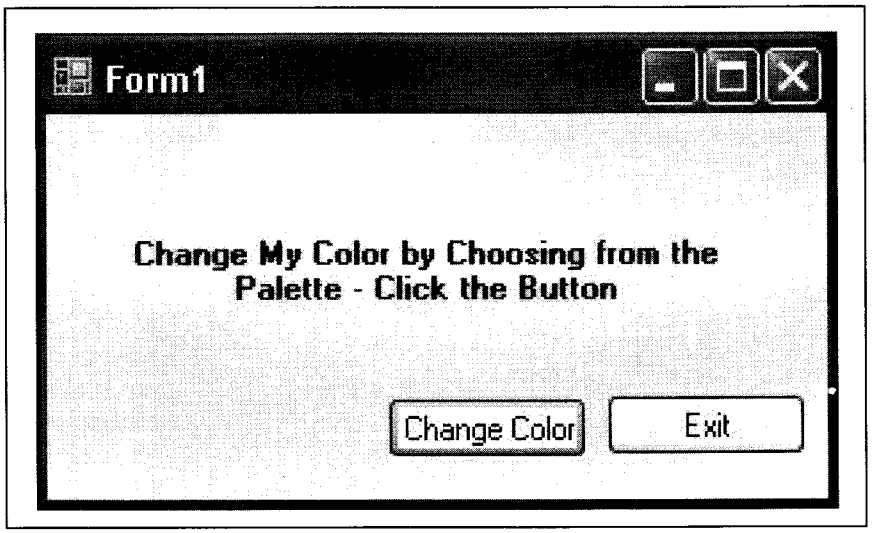
The program will run like the one shown below:



The three screenshots above illustrate how different common dialog boxes are displayed.

"Change Color" button opens the ColorDialog box. You can choose any color from the palette and click Ok. (The codes for event handling of Ok and Cancel button in the ColorDialogBox are not given in this program. You may however write it as shown earlier, if you want to see the impact of your program.)

You can write some message in label's text property. The next screenshot will show the changed fore color of the label1:



6.6.2 OpenFileDialog Class

This class provides users with the file selection capability. The properties and methods of this dialog *boxes are given below:*

Property or Method	Description
ShowDialog	Displays the dialog
MultiSelect	Sets/unsets the selection of multiple files
ShowReadOnly	Sets/unsets the read-only check box checked
Filter	Sets the type of files that will appear in the dialog box
FilterIndex	Sets the index of the filter selected in the dialog box

6.6.3 SaveFileDialog Class

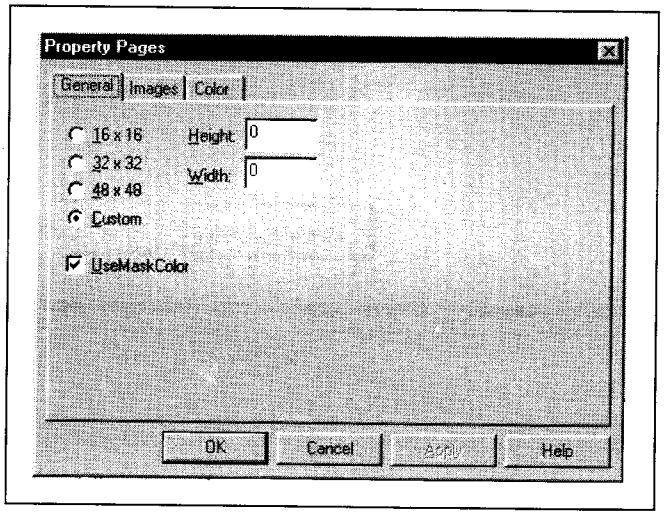
The SaveFileDialog class offers you the standard window that we see while saving the file. The methods and properties of this dialog box are given below:

Property or Method	Description
ShowDialog	Displays the message
CheckFileExists	Checks for the existence of file specified
FileName	Determines the file name selected by the user
Filter	Condition for files to be shown in the dialog box
FilterIndex	Determine the index of the filter selected in the dialog box

6.7 IMAGELIST CONTROL

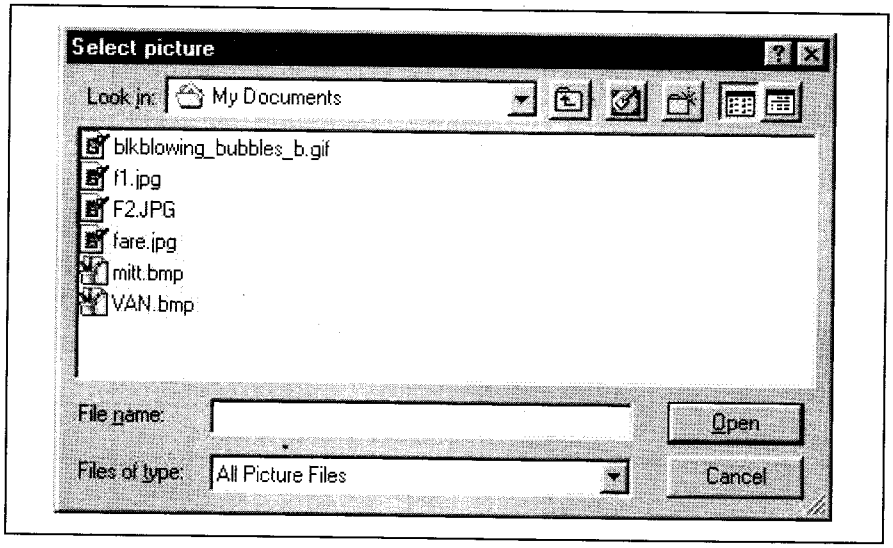
This control is supposed to 'serve' other controls. It is not visible at run-time. It holds a collection of images for other controls to use. Thus, the ImageList is a storehouse of images for other controls.

To set the properties of Imagelist, we right-click on it. When the drop-down menu appears, we select 'properties'. A new window appears now, displaying the Property Pages for the ImageList, as shown here:

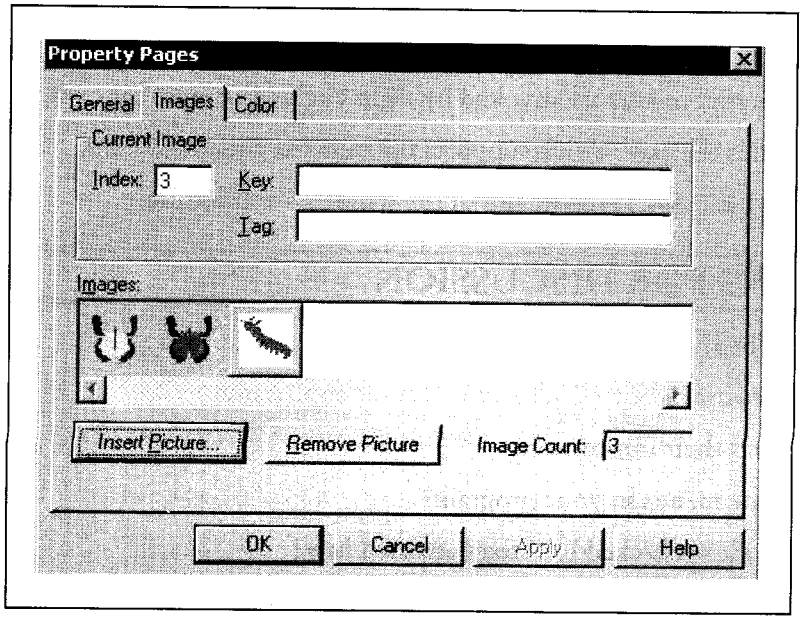


To set the images in the imagelist control, take the following steps:

1. Click on the tab with the caption 'Images'. A new tab opens up.
2. In the new tab, click on the 'insert picture' button. A 'Select picture' dialog box opens up to select the image (see figure).



3. Select the image from the dialog box and click on 'OK'. The image is inserted into the ImageList (see figure):



To add additional images, repeat the steps 1 and 2 mentioned above. As we add more and more images, the 'index' caption at the top-left gets set automatically. The index is 1 for the 1st image, 2 for the 2nd image and 3 for 3rd image, and so on. This index property is unique for each and every image. This index will be used later, to link an image from the imagelist to another control.

Check Your Progress

1. What does Multiselect property of the listbox specify?
2. Define the term 'Context Menu'.

6.8 LET US SUM UP

In Windows, we choose a value from a list by clicking the mouse on a particular item. We can make multiple selections from a listbox. This can be done in the typical Windows style, using either the 'Shift' key or the 'Ctrl' key. , unlike a listbox, we cannot make multiple selections in a *combobox.*

Everyone is familiar with Menus. When we work with windows, menus are the basic tools used in any of the windows application. Examples of Menu items are File, Edit, Format etc. Menus in applications are those that allow us to make a selection out of the choices, when we want to perform some action with the application, for example, to format the text, open a new file, print and so on. In VB.Net, MainMenu is the container for the Menu structure of the form. Menus are made of MenuItem objects that represent individual parts of a menu (like File->New, Open, Save, Save As etc.).

Creating Menus is simple. Drag a MainMenu component from the toolbar onto the form. When you add a MaiuMenu component to the form it appears in the component tray below the form.

Dialog boxes are one means of requesting users for specific kinds of inputs. VB.Net allows its designers to create a number of different types of dialog boxes such as file dialog, color dialog, etc.

Image list holds a collection of images for other controls to use.

6.9 KEYWORDS

Context Menu: The context menus are invoked by right clicking on another control.

MenuItems: MenuItem is the class which supports the items in a menu system.

Menus: Menus are the basic tools used in any of the windows application.

6.10 QUESTIONS FOR DISCUSSION

1. What is a Dialog box?
2. Describe the process of adding Dialog box to your application.
3. Explain Menus and their importance.
4. How do you create menus in your program?
5. What is list box? How checked list boxes are used in it?
6. What is picture box?

Check Your Progress: Modal Answers

1. Multiselect property sets whether we can select multiple items from a listbox or not. The default value is for single selection. To allow the user to select more than one item at the same time, we have to set this property to a different value.
2. The context menus are invoked by right clicking on another control. We can use context menus to display menus that will be specific for that control.

6.11 SUGGESTED READINGS

Sanjeev Sharma, *Visual Basic 6*, Excel Books

A. Chakraborti et al., *Microsoft.Net Framework*, PHI, 2002

M. Reynolds et al., *.Net Enterprise*, Wrox/SPD, 2002

Vikas Gupta, *.Net Programming*, Dreamtech Publication

Mackenzie Sharkey, *Teach yourself Visual Basic.Net in 21 days*, SAMS Techmedia

Bible, Bill Evjen, Jason Beres et al., *Visual Basic.Net Programming*, Wiley India

LESSON

7

WINDOWS COMMON CONTROLS: TREEVIEW, STATUS BAR, PROGRESS BAR, TAB CONTROL

CONTENTS

7.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Discuss treeview control
- Discuss status bar, progress bar, and tab control

7.1 INTRODUCTION

The treeview is a non-intrinsic controls, member of the Microsoft Windows Common Controls-2 6.0 (SP4) collection. We add them to the toolbox. It is probably the most complex of all of the controls. Most of its tasks are performed through programming. For this reason, we have to deal with a lot of their properties, methods and events as compared to other Windows Common controls.

Status bar is also one of the non-intrinsic controls. It informs us of our current page, the total number of pages, the line number, etc.

Progress bar and Tab control are also discussed in this lesson.

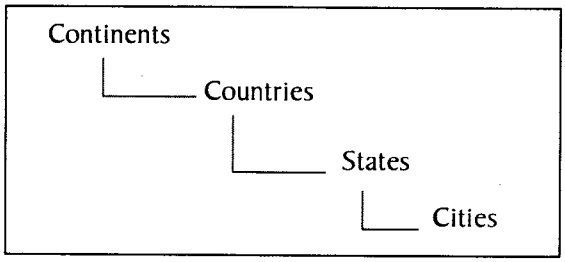
7.2 TREEVIEW CONTROL

When we open Windows Explorer, we see two panes - one on the left, the other on the right. The left pane lists the drives, folders and sub-folders in a 'hierarchy' in this manner:

- Once we open the 'c' drive, we can see all the folders in the drive.
- Once we open a particular folder, we can see its sub-folders listed.
- Once we open a sub-folder, we can see all the files in the sub-folder.

This is the hierarchy we are referring to. This kind of a view is provided by the treeview control. It helps view hierarchical data as a 'tree', with roots, nodes and sub-nodes (or branches). Thus, where we have hierarchical data, we can represent the hierarchy 'diagrammatically', by using the treeview control.

Consider a Program on Geography: all the continents, countries, their states, capitals, and so on. It would be an ideal case for the treeview control - we show the continents at the top level in the hierarchy, the countries at the next level, the states at the next, and the cities at the next level in the hierarchy. Diagrammatically, we represent the hierarchy as:



Isn't this the kind of a view we get in the left pane of Windows Explorer? The treeview is essentially a collection of nodes. In the diagram given above, 'continents' represents the root node, and all other nodes are child nodes of some node or the other. For instance, 'countries' is a child node of 'continents' and the parent node of 'states'. Similarly, 'states' is a child node of 'countries' and a parent of 'cities'. This means that a node can be the parent to some node and the child for another node. The treeview offers this facility without too much of an effort. We can manipulate these nodes through code. Let us see how to work with the treeview control.

7.2.1 Adding a Node to the Treeview

The treeview is a collection of nodes. Thus, the first thing to do when using treeview control is to add nodes to it. The first step in this direction is to 'declare' a variable of the type of node. This is done in the general/declarations section:

PRIVATE nod AS NODE (in general/declarations)

Doing so declares a variable 'nod' of the type node. It is a variable, which represents an 'object' so it is called as an object variable, to distinguish it from other traditional variables. Later on in the program, through this variable we will be able to refer to any node in the treeview, whenever required. To add a node to the treeview, we use the add method of the nodes collection of the treeview. The general syntax for creating a node is:

SET nod = tv.NODES.ADD(RELATIVE,RELATION,KEY,TEXT,IMAGE)

where,

- Nod is the name of the node (object variable).
- tv is the name assigned to the treeview.
- Relative is the key of the node to which the current node is related.
- Relation is the relationship of the current node with the node specified under 'relative'. It can have the following possible values:
  0 tvwfirst
  1. tvwlast
  2. tvwnext
  3. tvwprevious
  4. tvwchild
- Key is a unique name given to the node to distinguish it from other nodes. This parameter is mandatory.
- Text is the text we want to be displayed on the current node. This parameter is mandatory.
- Image is the index of the image (from the imagelist control) to be shown alongwith the 'text'.
- Selected Image is the index of the image (from the imagelist control) to be shown alongwith the 'text', when the node is 'selected' by the user.

Based on this syntax, we can write the following code to add the root node to our treeview control (we are using the form_load event):

```
SET nod = tv1.NODES.ADD (,,"root_geog","EARTH")
```

In the line given above, we are 'creating' an object variable by the name 'nod' (it has already been 'declared' in the general/declaration section). Let us analyze the values we have given for the different parameters:

- *Relative:* This parameter has not been passed by us in the present case. This is because this node ('nod') is going to be the top-level node (root node) of the treeview. This missing parameter is represented by the presence of a comma without any value in the position of the parameter.
- *Relation:* This parameter too has not been passed by us in the present case. This is because this node ('nod') is going to be the top-level node (root node) of the treeview. Thus, it will be illogical to specify the relation of the node with itself. This missing parameter is represented by the presence of a comma without any value in the position of the parameter.
- *Key:* This value has been specified as 'root_geog'. This value, always unique, will be used to identify the node later on in the program.
- *Text:* This value has been specified as 'earth'. This value will be used to display the text to the user.
- *Image:* This parameter has also been omitted.
- *Selected Image:* This parameter has also been omitted.

Once we have done this, we will add more nodes to the treeview. These nodes will contain the names of continents. Thus, these nodes will all be the child nodes of the root_geog node. We will add the names of continents as follows:

```
tv.NODES.ADD "root_geog",TVWCHILD,"cont1","Asia"
```

The line given above adds a node to the treeview. The details of the statement are as follows:

- *Relative:* This node will be created 'relative to' the node with the key of "root_geog".
- *Relationship:* We specify the value of tvwchild. Thus, the current node will be a child node of node specified under relative, i.e., "root_geog".
- *Key:* This unique value is specified as "cont1".
- *Text:* This value, meant for display, has been set to "Asia".
- *Image and Selected Image:* These two optional values have not been set yet.

Now that we have added one child node, we'll add one more child node:

```
tv.NODES.ADD "root_geog",TVWCHILD,"cont2","Africa"
```

The line given above adds a node to the treeview. The details of the line are:

- *Relative:* The node will be created 'relative to' node with key of "root_geog".
- *Relationship:* We specify the value of tvwchild. Thus, the current node will be a child node of node specified under relative, i.e., "root_geog".
- *Key:* This unique value is specified as "cont2".
- *Text:* This value, meant for display, has been set to "Africa".
- *Image and Selected Image:* These two optional values have not been set yet.

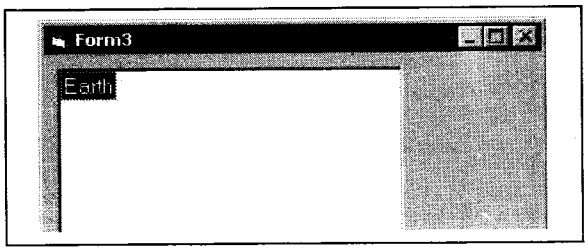
We will now consolidate all the statements given by us so far, as shown in the following code segment:

Code Listing treeview.1

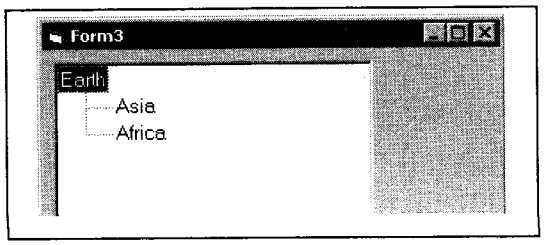
```
PRIVATE nod AS NODE 'in general/declaration
PRIVATE SUB FORM_LOAD( )
```

```
SET nod =  tv.NODES.ADD (,,"root_geog","Earth") 'root node
tv.NODES.ADD "root_geog",TVWCHILD,"cont1","Asia"
tv.NODES.ADD "root_geog",TVWCHILD,"cont2","Africa"
END SUB
```

Now run the project and see the output. We will see the text of the root node ('Earth'), as shown in the figure:



This appears to be alright, but wait !! We have also given two child nodes with the text 'Asia' and 'Africa'. Where are they? After all, we also have not received any error either from the program. To find the child nodes, just try and double-click on 'Earth'. And what do we see? The two child nodes now expand (opens up), and 'Asia' and 'Africa' can be seen by us, as the figure shows:



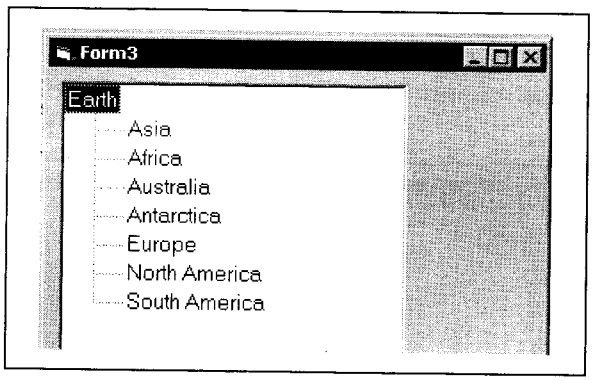
Note that this view represents a hierarchy, since the root node is at the left and the child nodes are indented towards the right.

Now add some more lines to code listing treeview.1, to get a code like this:

Code Listing treeview.2

```
PRIVATE nod AS NODE 'in general/declaration
PRIVATE SUB FORM_LOAD()
SET nod = tv.NODES.ADD (,,"root_geog","Earth") 'root node
tv.NODES.ADD "root_geog",TVWCHILD,"cont1","Asia"
tv.NODES.ADD "root_geog",TVWCHILD,"cont2","Africa"
tv.NODES.ADD "root_geog",TVWCHILD,"cont3","Australia"
tv.NODES.ADD "root_geog",TVWCHILD,"cont4","Antarctica"
tv.NODES.ADD "root_geog",TVWCHILD,"cont5","Europe"
tv.NODES.ADD "root_geog",TVWCHILD,"cont6","North America"
tv.NODES.ADD "root_geog",TVWCHILD,"cont7","South America"
END SUB
```

Now run the program and observe the output. It is the same story. Once we double-click on Earth, the treeview opens up and shows all the child nodes. Thus, names of all the continents become visible, as shown here:



7.2.2 Keeping Nodes Expanded at the Very Start

We have to always double-click on the root of the treeview, to expand it and to see all the child nodes. If we want to ensure that all the child elements are expanded right from the time they are added, we use the expanded property of the node object. This is a boolean property, whose value is 'false' by default. We set it to 'true', so that the nodes are expanded automatically. The code is:

```
nod.EXPANDED = TRUE
```

Adding this line to our existing code, we get:

Code Listing treeview.3

```
PRIVATE nod AS NODE 'in general/declaration
PRIVATE SUB FORM_LOAD( )
SET nod = tv.NODES.ADD (,,"root_geog","Earth")
tv.NODES.ADD "root_geog",TVWCHILD,"cont1","Asia"
tv.NODES.ADD "root_geog",TVWCHILD,"cont2","Africa"
tv.NODES.ADD "root_geog",TVWCHILD,"cont3","Australia"
tv.NODES.ADD "root_geog",TVWCHILD,"cont4","Antarctica"
tv.NODES.ADD "root_geog",TVWCHILD,"cont5","Europe"
tv.NODES.ADD "root_geog",TVWCHILD,"cont4","North America"
tv.NODES.ADD "root_geog",TVWCHILD,"cont5","South America"
nod.EXPANDED = TRUE 'new line added
END SUB
```

When we run the program now, the new line (last line) ensures that all the nodes are in the expanded state, even if we have not double-clicked on the root node. Thus, our output will be exactly as the last image seen by us.

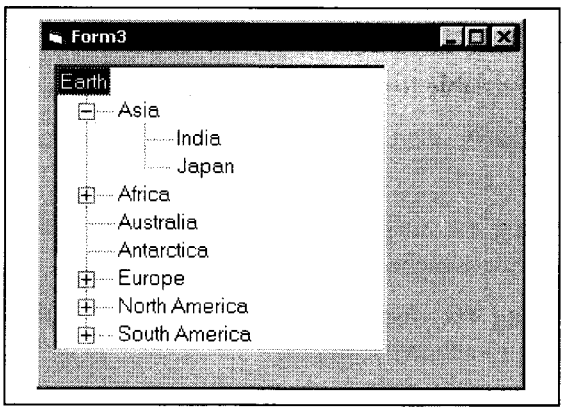
Now that the continents have been added to the treeview, we need to add the countries to the continents. This means that we need to add more nodes. As is obvious, these nodes will be the child

nodes of their respective continents. To add more nodes to the treeview to represent the countries, we modify code listing treeview.3 to get the following code segment:

Code Listing treeview.4

```
PRIVATE nod AS NODE 'in general/declaration
PRIVATE SUB FORM_LOAD()
SET nod =  tv.NODES.ADD (,,"root_geog","Earth")
tv.NODES.ADD "root_geog",TVWCHILD,"cont1","Asia"
tv.NODES.ADD "cont1",TVWCHILD,"cont1_1","India"
tv.NODES.ADD "cont1",TVWCHILD,"cont1_2","Japan"
tv.NODES.ADD "root_geog",TVWCHILD,"cont2","Africa"
tv.NODES.ADD "cont2",TVWCHILD,"cont2_1","South Africa"
tv.NODES.ADD "cont2",TVWCHILD,"cont2_2","Egypt"
tv.NODES.ADD "root_geog",TVWCHILD,"cont3","Australia"
tv.NODES.ADD "root_geog",TVWCHILD,"cont4","Antarctica"
tv.NODES.ADD "root_geog",TVWCHILD,"cont5","Europe"
tv.NODES.ADD "cont5",TVWCHILD,"cont5_1","Germany"
tv.NODES.ADD "cont5",TVWCHILD,"cont5_2","France"
tv.NODES.ADD "root_geog",TVWCHILD,"cont6","North America"
tv.NODES.ADD "cont6",TVWCHILD,"cont6_1","U.S.A."
tv.NODES.ADD "cont6",TVWCHILD,"cont6_2","Canada"
tv.NODES.ADD "root_geog",TVWCHILD,"cont7","South America"
tv.NODES.ADD "cont7",TVWCHILD,"cont7_1","Brazil"
tv.NODES.ADD "cont7",TVWCHILD,"cont7_2","Argentina"
nod.EXPANDED = TRUE
END SUB
```

The output of this code is shown in the figure that follows. Note the 'Plus' symbol in front of the continents. The 1st continent, Asia, has been expanded:



7.2.3 Counting the Nodes

To get the total count of the nodes, we use the count method of the nodes collection. The output can be assigned to an integer type of a variable, or can be displayed in a messagebox.

To get the count of the nodes in the treeview, add a command button and a label to the form. Code for the click event of the command button as follows:

Code Listing treeview.5

```
PRIVATE SUB command1_CLICK()
1.  DIM int_count AS INTEGER
2.  int_count = tv.NODES.COUNT
3.  label1.CAPTION = "Total NODES = " & int_count
END SUB
```

When we run the program and click on the command button 'command1', line no. 2 counts the total number of nodes in the treeview and assigns the result to the variable int_count. This value is then displayed in the label 'label1'.

7.2.4 Iterating through the Entire Treeview

For going through all the nodes of a treeview, we can make use of the for...each....next loop. This loop, we have already seen, helps us to loop through a collection effortlessly. While we are looping, we can print the text and the key of the current node in a listbox. The code for this will be as follows:

Code Listing treeview.6

```
PRIVATE SUB command1_CLICK()
   1. DIM txt_msg AS STRING
   2. FOR EACH nod IN tv.NODES
   3. txt_msg = "Current node's text is " & nod.TEXT
   4. txt_msg = txt_msg & " and key is " & nod.KEY
   5. list1.ADDITEM txt_msg
   6. NEXT
END SUB
```

Here, the code runs as follows:

1. Line no. 1 declares a string variable for storing the text and the key to be displayed.
2. Line no. 2 begins the for...each...next loop. The loop will continue till the time more nodes are available.
3. Line no. 3 prepares the string 'txt_msg' for displaying the text of the current node. It uses the text property of the node object to achieve this.
4. Line no. 4 prepares the string 'txt_msg' for displaying the key of the current node. It uses the key property of the node object to achieve this.
5. Line no. 5 adds this string to a listbox named "list1", which we have added to the form.

6. Line no. 6 continues with the iteration of the for...each loop, shifting the program control to line no. 2. Once no more nodes are available, the loop will terminate.

Ensuring Nodes are visible during Iteration

In the code segment we have just seen, we face a peculiar situation. While iterating through our treeview, if the current node is present deep down in the hierarchy, it might not be visible to the user. During iteration, we might want to show that particular node to the user. To do so, we use the ensurevisible method of the node object. This method, true to its name, makes the node visible (if it was invisible so far). We can modify the previous code segment, adding just one line as shown here:

Code Listing treeview.7

```
PRIVATE SUB command1_CLICK()
1.  DIM txt_msg AS STRING
2.  FOR EACH nod IN tv.NODES
3.  txt_msg = "Current node's text is " & nod.TEXT
4.  txt_msg = txt_msg & " and key is " & nod.KEY
5.  list1.ADDITEM txt_msg
6.  nod.ENSUREVISIBLE 'make current node visible, if invisible
7.  NEXT
END SUB
```

In the code given here, line no. 6 is the only addition to the previous code. When this line is executed at every iteration, the treeview makes sure that the current node ('nod') is made visible.

7.2.5 Determining Path of a Node

In Windows, we refer to 'path' of a file or a folder. For instance, the VB.EXE file might have the path c:\program files\microsoft visual studio\vb98 on our computer. This way, we can access it through the path whenever required. The treeview also provides this feature, through the path property, as follows:

nodename.PATH (KEY)

For example, through our code listing treeview.3, the path of 'Europe' can be found through:

nod.PATH ("cont5")

Once we run the code given above, we get the output as:

Earth\Europe

Similarly, the path of 'Africa' can be found through:

nod.PATH ("cont2")

Once we run the code given above, we get the output as:

Earth\Africa

7.2.6 Counting Number of Child Nodes of a Particular Node

When we need to count the number of child nodes for a particular node, we use the children property, as follows:

```
tv.NODES (KEY).CHILDREN
```

Using this syntax, we get the number of children of 'Asia' node as follows:

```
MSGBOX tv.NODES ("cont1").CHILDREN
```

When iterating through the treeview, we can display the number of children of the current node as shown here:

```
MSGBOX nod.CHILDREN
```

7.2.7 Adding values to Nodes Dynamically

So far, we have given pre-determined values to our treeview. We can also specify the text of our nodes at run-time, through an input given by the user. To do so, we add another button to our existing project, and code for the click event like this:

Code Listing treeview.8

```
PRIVATE SUB command1_CLICK()
   1. DIM txt_node AS STRING
   2. DIM txt_key AS STRING
   3. DIM int_count AS INTEGER
   4. int_count = tv.NODES.COUNT 'find total count of nodes
   5. int_count = int_count + 1  'increment total count by 1
   6. txt_key = "key" & int_count    'prepare key for new node
   7. txt_node = INPUTBOX ("WHAT TEXT TO ADD")
   8. tv.NODES.ADD ,,txt_key,txt_node    'new node added
END SUB
```

Here, the code runs as follows:

1. Line nos. 1 - 3 declare variables.
2. Line no. 4 counts the total no. of nodes in the treeview and assigns the result to the variable int_count.
3. Line no. 5 adds 1 to the total node count, i.e., int_count.
4. Line no. 6 prepares the key for the new node. Since the user might type an existing key, we prepare the unique key through code. For this, we concatenate the word 'key' and the variable int_count and assign the output to the variable txt_key. This combination, as we can easily understand, will always give us a unique value.
5. Line no. 7 asks the user for the text of the new node.
6. Line no. 8 adds the new node to the treeview. Note that no relative has been specified, making this node a top-level node. The key is passed as txt_key and the text is passed as txt_node.

7.3 STATUS BAR CONTROL

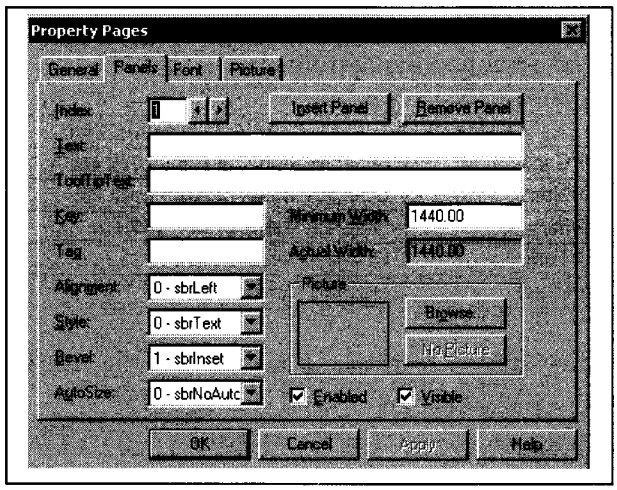
When working in MS-Word, we see the status bar at the bottom. It informs us of our current page, the total number of pages, the line number, etc. VB can provide the same functionality through the statusbar control. Using this control, we can give a lot of useful information, such as displaying the current record number in a set of records, the current date/time, etc. Once we get the statusbar on the form, we add panels to use it.

A statusbar is a collection of 'panels'.

7.3.1 Adding Panels to the Statusbar

To add panels:

1. Right-click on statusbar.
2. Select properties from the pop-up menu that appears. A property pages dialog box opens up.
3. Select the panels tab from this. As we can see from the figure, the 1st panel has already been added, with an index of 1.



4. Assign a suitable text for it.
5. To add more panels, click on 'insert panel' button. A new panel is added to the statusbar, with index value of 2.
6. Give a suitable text for panel.
7. Repeat the process as required.

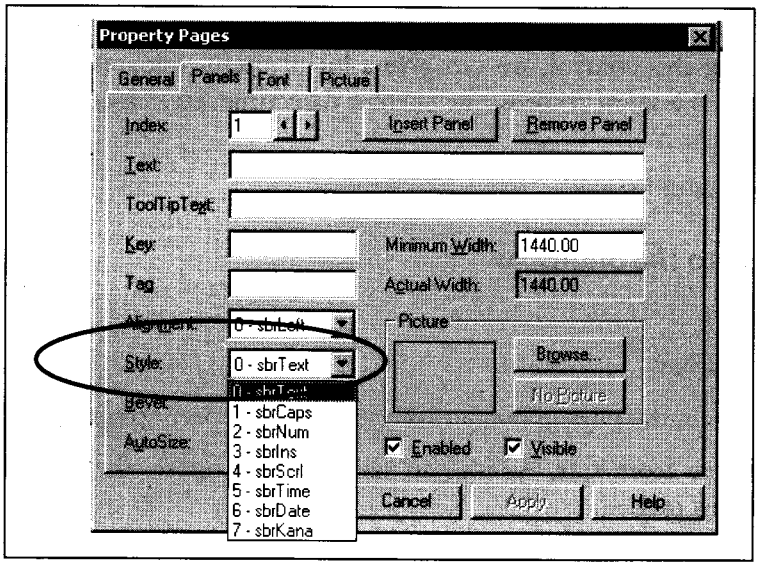
Style Property

This property is for useful features set as displaying present date or time, Caps Lock or Num Lock status, etc.

To display these features in the status bar, we take the following steps:

1. Open property pages of the statusbar.
2. Select 'panels' tab.

3. Select index of the panel where we want to show the date or the time.
4. Click on style drop-down button. Observe drop-down list that appears:



From here, we choose one of the following options:

Style Value	Effect on Status Bar
SbrCaps	Displays status of Caps Lock key
SbrNum	Displays status of Num Lock key
SbrIns	Displays status of Ins Lock key
SbrTime	Displays current Time
SbrDate	Displays current Date

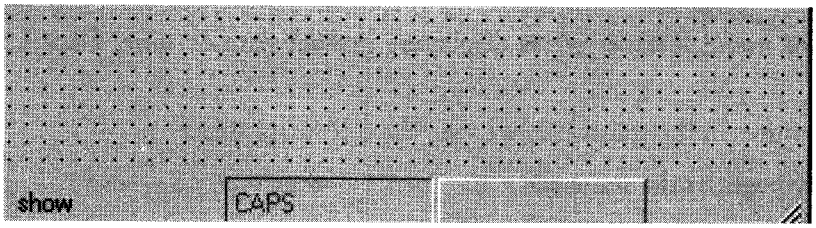
If Caps Lock is off, the word 'caps' will appear 'gray' (dull). If the caps lock is on, 'caps' will be black. Same is the case for Num Lock and Ins key.

Setting Appearance of the Panels

The panels, by default, appear to be 'inset' on the statusbar. We can set the appearance to three possible values, using the bevel property on the panels tab:

- 0 - sbrNoBevel
- 1 - sbrInset (the default)
- 2 - sbrRaised

Here is a picture of the statusbar, showing all the three possible bevel values:



Adding Panels at Run-time

So far, we have added the panels at design-time and worked and seen the results. If we want to add panels at run-time, we can use the add method of the panels collection of the statusbar, as follows:

Statusbar.PANELS.ADD [index,key,text,style,picture] AS PANEL

Code Listing status.1

```
PRIVATE SUB command1_CLICK()
   1. DIM int_count AS INTEGER
   2. DIM int_new_count AS INTEGER
   3. DIM str_panel_text AS STRING
   4. int_count = statusbar1.PANELS.COUNT
   5. int_new_count = int_count + 1
   6. str_panel_text = INPUTBOX("Enter text for the panel ..")
   7. statusbar1.PANELS.ADD int_new_count,,str_panel_text
END SUB
```

Here, the code runs as follows:

1. Line nos. 1 to 3 declare variables for holding various kinds of data.
2. Line no. 4 uses 'count' method of statusbar's panels collection. This method helps count the number of panels. This value is stored in the variable int_count.
3. Line no. 5 increases int_count by 1 and stores this value in the variable int_new_count.
4. Line no. 6 asks for the text to be used for the panel, through the inputbox.
5. Line no. 7 uses 'add' method of the statusbar's panels collection, which adds a new panel to the statusbar. We pass the value of int_new_count as the index and str_panel_text as the text argument for the add method.

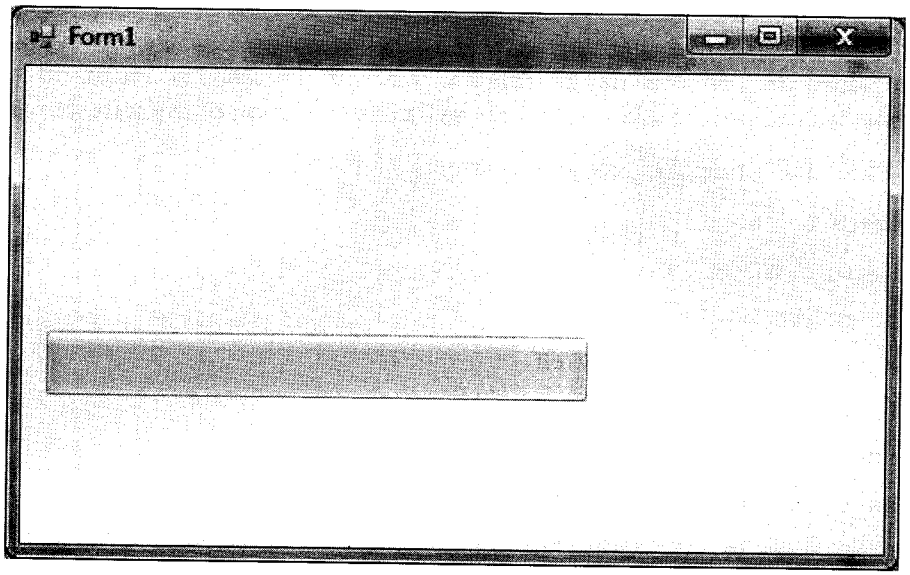
7.4 PROGRESS BAR

A Progress Bar control shows the progress of an operation that takes time and a user has to wait for the operation to be finished.

7.4.1 Creating a Progress Bar

Here, we will show the creation of a Progress Bar control using a Forms designer at design-time

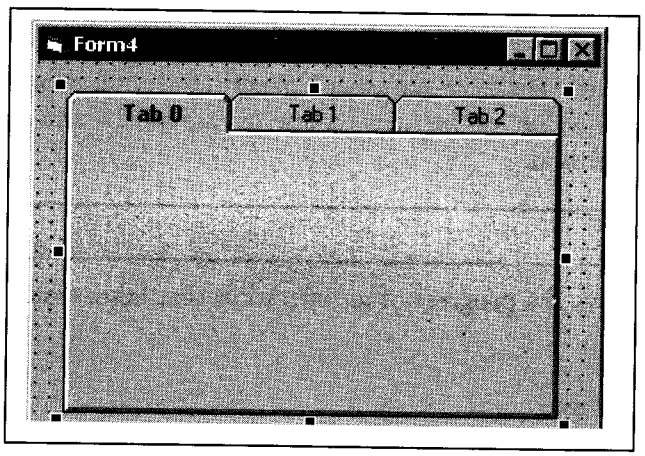
To create a Progress Bar control at design-time, drag and drop a ProgressBar control from Toolbox to a Form in Visual Studio. The ProgressBar1 is added to the Form after you drag and drop a ProgressBar on a Form and it looks like Figure below.



Source: http://www.vbdotnetheaven.com/UploadFile/mahesh/2708/Default.aspx

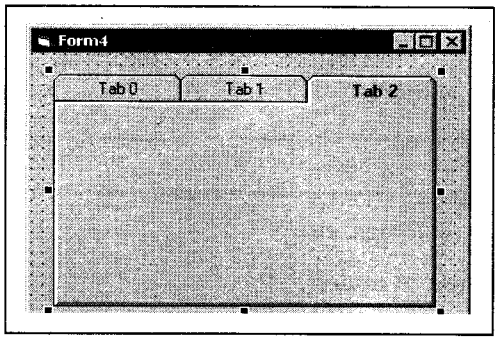
7.5 TAB CONTROL

Another commonly used Windows control is the Tabbed Dialog Control. We see this control when configuring ODBC data sources, when setting the properties of the VB ToolBar, StatusBar, etc. We can add the same tabbed dialog control to our form. To do so, we go to the menu bar and click on 'Project' 'Components'. From the dialog box that appears, select 'Microsoft Tabbed Dialog Control 6.0'. When we draw the control on the form, it will appear this way:



In situations where we have a very complex form, such that all the controls will not be able to fit in the form properly, we can use the tabbed dialog control. The logically related controls can be grouped into tabs.

To add controls to a specific tab, just click on the tab buttons at the top of the control. The tab on which we click comes 'on top' of the other tabs. We can now draw controls on the tab, just as we do on a form. Suppose we click on the tab with the caption 'tab2'. The following figure shows the tab 'tab2' covering the other tabs:



7.5.1 Setting Caption of the Tabs

To set the caption of the different tabs, click on a particular tab and then set the caption in the properties window. For example, to set the caption of the 1st tab, click on it and come to the properties window. Here, type a new caption for the tab in the caption property. Now click on the 2nd tab and come to the properties window. Once again, type a new caption for the currently selected tab in the caption property.

7.5.2 Setting the Number of Tabs

The tabs property of the control sets/returns the number of tabs on the tabbed dialog control. The default value is 3, i.e., the control has three tabs by default.

7.5.3 Determining which Tab was clicked

The tab property of the control returns/sets the tab number of the active tab. For example, the following code returns the number of the tab clicked by the user. We are using the double-click event of the control:

Code Listing tabbed.1

```
PRIVATE SUB sstab1_DBLCLICK( )
MSGBOX sstab.tab  'assuming sstab is name given to Tabbed Dialog Control
END SUB
```

Try the code and see the result in the message box. Double-click on different tabs and see the message box changing tab names at every double-click.

Check Your Progress

1. What is progress bar?
2. Discuss the settings performed for caption of the tab.

7.6 LET US SUM UP

Treeview helps view hierarchical data as a 'tree', with roots, nodes and sub-nodes (or branches).The treeview is a collection of nodes. To get the total count of the nodes, we use the count method of the nodes collection. The output can be assigned to an integer type of a variable, or can be displayed in a messagebox. Using status bar control, we can give a lot of useful information, such as displaying the

current record number in a set of records, the current date/time, etc. A ProgressBar control shows the progress of an operation. We see the tab control when configuring ODBC data sources, when setting the properties of the VB ToolBar, StatusBar, etc.

7.7 KEYWORDS

StatusBar: It informs us of our current page, etc.

Treeview: The treeview is a collection of nodes.

ProgressBar: It control shows the progress of an operation.

7.8 QUESTIONS FOR DISCUSSION

1. What is treeview? Discuss how to add node to the treeview.
2. Discuss how iteration is performed through the entire treeview.
3. Discuss the steps used for adding panels to the status bar.

Check Your Progress: Modal Answers

1. A ProgressBar control shows the progress of an operation that takes time and a user has to wait for the operation to be finished.
2. To set the caption of the different tabs, click on a particular tab and then set the caption in the properties window.

7.9 SUGGESTED READINGS

Sanjeev Sharma, *Visual Basic 6*, Excel Books, New Delhi

A. Chakraborti et al., *Microsoft.Net Framework*, PHI, 2002

M. Reynolds et al., *.Net Enterprise*, Wrox/SPD, 2002

Vikas Gupta, *.Net Programming*, Dreamtech Publication

Mackenzie Sharkey, *Teach yourself Visual Basic.Net in 21 days*, SAMS Techmedia

Bible, Bill Evjen, Jason Beres et al., *Visual Basic.Net Programming*, Wiley India

UNIT IV

LESSON

8

OBJECT ORIENTED SYSTEMS

CONTENTS

8.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Understand the OOPS concepts like abstraction, inheritance, polymorphism, etc.
- Understand classes and objects
- Discuss creating classes using class module and class builder utility

8.1 INTRODUCTION

In OOPS, a basic thinking is to segregate, i.e., isolate the various units of a system. We have the benefit of flexibility in the system - we can change one sub-system with either little or no changes at all in the associated sub-systems. We will focus on the core concepts of classes and objects. These are the central characters in the entire OOPS story. We will discuss OOPS concepts like abstraction, encapsulation, inheritance, and polymorphism. Also we will discuss the concept of creating classes using class module and class builder utility.

8.2 OOPS V/S OTHER TECHNIQUES

The world of computing is constantly changing. The gentleman who first said "Change is the only thing permanent in this world" probably didn't know how true his words would be, especially in computing. Everyday - ok, every week - we see or read something new in the field of information technology. The floppy disc of yesteryears changed from 5.25" to 3.5" to 0" (almost vanished). The CDs changed into DVDs, VCDs, and what not! The walkman gave way to CD/DVD players, and now we see the I-pod. We also see the pen drives, followed by the USB mp3 players- the list is endless!

The world of programming, too, is continually changing. Hand-written code gave way to interpreters, which in turn gave way to compilers. In programming languages, the change has been as follows:

- Machine language
- Assembly language
- High-Level languages - BASIC, Pascal, Fortran
- Procedural languages - C
- 4th Generation Languages - SQL
- Object-Oriented programming languages - C++, Java
- Event-Driven programming languages - VB

Though every step forward in programming has been historical, the arrival of OOPS is - probably - the biggest step forward. When programming in BASIC, Pascal, etc. the applications were monolithic - the entire program was just a single body of code. The code was executed linearly, from top to bottom. As the application requirements increased, the code too increased proportionately. As a result, all of these activities got tougher and tougher:

- Coding
- Testing/debugging
- Error-handling etc.

The core concept in procedural languages is that they are code-centric. That is to say, in these languages, the 'code controls the access to data'. In OOPS, the code-centric approach is replaced by a data-centric approach. Thus, in OOPS the 'data controls the access to code'. This means that we can initiate different segments of code in an application, based on the type of data involved - date, string, number, etc.

In OOPS, a basic thinking is to segregate, i.e., isolate the various units of a system. This way, we have the benefit of flexibility in the system – we can change one sub-system with either little or no changes at all in the associated sub-systems. On the other hand, in a system with very tightly integrated sub-systems, it will be a tough task to make changes; any change in one part will have an effect on the associated parts. Thus, in the procedural languages, we need to manage the change at multiple places, not just one place.

Taking an easy example of OOPS, think of the typical application at kirana shops - the common software that accepts item codes and quantity, and then generates and prints bills. In OOPS, the creation of the bills can be separated from their printing. With this approach, if we need to change some processing in the bill-generation system, we can do so without affecting the printing system. We simply need to re-define or re-create the interfaces between the bill-generation and bill-printing systems. The attempt should be to create interfaces that are fairly generic. This way, the internal processing of printing need not be made known to the bill-generation system. As a result, if we change the methodology of generating the bills, the printing technology need not necessarily be changed. Similarly, if the printing methodology undergoes a change, the billing system need not be changed. We can continue to use the invoice-generation and printing systems as they have been used so far, as far as interface between them does not change.

This way, OOPS offers a better option for software development. Under OOPS, the applications are sub-divided into classes, and different classes can be combined in different.

8.3 CLASSES AND OBJECTS

Now we focus on the core concepts of classes and objects. These are the central characters in the entire OOPS story. At its most basic level, a class is simply a container for code and data. It defines the structure and behaviour of the contained data. Thus, a class might specify:

- a particular variable is of the Integer type (structure) and,
- it should be multiplied by 3 and then squared (behaviour) and,
- it should be multiplied by 6.75 to get the final result (behaviour).

A class is supposed to just provide a template or a building block. It will not do anything more than this. To take a simple analogy,

- The map of a housing complex drawn by an architect is a class.
- The original engineering drawing of a car/scooter is a class.
- The drawing of a nuclear reactor is a class.

Thus, as we can now understand, a class only provides the specifications of what is to be done and how it is to be done.

From the viewpoint of OOP, the data of a class becomes a property of the class. On the other hand, the code of the class becomes a method of the class. The property describes the features of the class, whereas the method helps to work on the data in the class. In VB 6.0, a class also exposes events, which determine when a class will react to an external stimuli or a message. The properties, methods, and events of a class are collectively referred to as data members of the class.

To understand the concept of data members, think of a TextBox. The textbox class exposes the following data members:

- Property - name/width/height/backcolor
- Method - move/refresh
- Event - keyup/keydown/keypress/change, etc.

Once a class has been created, we need to do something in order to use it. To use the class, we create an object of it. This means that while the class only 'defines' things, the object actually 'does' things. If we think of the example of a map drawn by an architect,

- the map is the class.
- the colony made on the basis of the map is the object.

Using a similar logic,

- the engineering drawing of a car is the class.
- the actual car built on the basis of this drawing is the object.

By now, it should be clear that the class is the starting point for all development under OOPS. A class is like a template or a building block for an object. An object, on the other hand, is an instance of the class. The object, thus, puts the structure and behaviour of the class to use. As we can understand by now, a class will not be able to expose any of its data members. However, the object of the class will expose these data members, allowing us to manipulate them through a code-based approach.

Having understood so much of OOPS, let us now consider the toolbox of VB. In the toolbox, we see the icon of the textbox. This icon represents the TextBox class. However, when we draw a Textbox on the VB form, it is an object of the TextBox class. The TextBox icon – in the toolbox – does not expose any methods, properties or events. Opposed to this, the TextBox object drawn on the Form exposes all of these members for us. Thus, we have the backcolor, forecolor, height and width property exposed; the move and refresh method exposed; as also the click and change events exposed by the textbox object drawn on the VB form.

If I told you that we have been using OOP in VB so far, even without knowing a single line about OOPS, would you believe it? Believe me, it is true! Whatever programming we have done so far in VB is based on OOP, though we haven't realized it. We have been using OOP in VB, though we have not studied it at all. But how? As we have been working in VB, we are already using the syntax of OOP. For example, the code text1.HEIGHT = 700 is a clear case of OOP-based syntax. The code text1.MOVE 1500, 2000 is also confirming to the syntax of OOP, i.e., object.method or object.property.

This is exactly how we access the members of a class in OOPS- the object.method or object.property syntax is used always in OOPS. This is known as the 'dot notation' in OOPS terminology.

8.4 OOPS CONCEPTS

8.4.1 Abstraction

Abstraction basically means how we, as human beings, deal with complexity. It means the same thing in OOPS, too. Instead of looking at the finer details of an application, we look at a big system as being composed of many small sub-systems. This way, our application becomes more easily understandable and more easily manageable. To take a simple example, suppose we own a Mercedes Maybach car. Now, we could describe our vehicle in two ways:

- as a 'Mercedes Benz Maybach' - quite simple really, or
- as a '3500-c.c. coolant-based petrol-driven vehicle with 6 synchromesh gears and reverse gear, autogear, GPS system, refrigerator, bar, ...'.

Needless to say, the first description is really very simple. It leaves out the finer details and presents a high-level view, which in OOPS terminology can be termed as an 'abstracted view'.

8.4.2 Encapsulation

Consider the example of a vehicle that we drive. When we press the brake, the vehicle stops. When we press the horn, it blows. Similarly, we press the kick, which turns the vehicle 'on'. What is the common link between all of these actions? It is that we are able to use our vehicle without even knowing how these things work. This is the concept of encapsulation – we can use an object without worrying how it works internally. Thus, though we might not be automobile engineers, we are all experts at driving a vehicle. Thus, even if we don't know the brake works or how the horn works, we can still use the break, clutch, gear, etc. This is the beauty of encapsulation.

We need to consider a class as a container for code and data. The data and the code is kept 'concealed', i.e., 'hidden' from the outer world. Then, through careful planning, we expose some well-defined and unique interfaces, through which others can use our class. Of course, since the interfaces are not disclosing the inner details, our user can only work in a fixed, well-defined manner. This way we get two benefits:

- The user will never know the inner working of the class. Thus, all the inner complexities are hidden from the user.
- The data and the code become highly secure, since, no one can accidentally or deliberately misuse our class.

Another common word for Encapsulation is 'data-hiding'. Encapsulation refers to the fact that the 'interface' and the 'implementation' are totally separated from each other. The keywords responsible for data hiding, which determine the visibility and the scope of our class members, are known as access specifiers or access modifiers. The following are access specifiers available in VB:

- Public - can be read/written from outside the class
- Private - can be read/written only from within the class
- Friend - can be read/written only within the project, in which the class is being used.

8.4.3 Inheritance

In OOPS, the word inheritance is exactly what its English meaning is. We inherit the property of our parents - cash/land or even skin colour and/or health. As mammals, we inherit the traits of talking, reproduction, respiration, etc. from our biological class, i.e, mammals. In the same manner, a class can 'inherit' another class. The big advantage of this approach is that we get a certain amount of functionality readymade. With this facility, we have to concentrate only on the extra features to be given in our 'inheriting' class, and we are ready.

As human beings, we all have similar characteristics - hair, erect posture, eyes, ears, hands, legs, fingers, etc. However, as we know, the finger-print of each human being is distinct. Even the iris of each human being is distinct. Thus, though the eyes, ears and fingers are common features, there is still an element of uniqueness in each person's organs. This is the real advantage of inheritance - some of the functionality is readymade; the remaining can be customized to define every application or every object uniquely. Technically, this is known as Code reusability. Code reusability is a key advantage of OOPS, offered through inheritance.

'Code once, use often' - this is the motto of the supporters of OOPS. This benefit of OOPS is provided by inheritance. Let us say we have a class named 'maths' with the following features for calculations:

- Sum
- Product
- Average of numbers

Now suppose we need a class with the three features mentioned above, plus the median and mode of the numbers involved. Now, we have two choices:

- either we create a class from scratch, or
- we use our existing 'maths' class.

In the first case, we will use a lot of time, coding for everything afresh. In the second case, however, we can:

- inherit our existing 'maths' class, and
- add one extra method each, for calculating the median and mode.

As is quite natural, the second option will save a lot of coding time – sum, product, and average coding is readymade, coming from the inherited class. Thus, we will be over with the complete coding in minimum time possible.

8.4.4 Polymorphism

Going by the exact meaning, this term means 'many forms'. The basic concept behind polymorphism is 'One interface for multiple methods'. To understand this, consider the case of a car with automatic gears. The lever of the gears is the same, whether driving by manual shifting of gears or by the automatic gear. Whichever method we choose, the gear lever remains the same. This means, by using the same lever in different ways, we can get different results from it.

Now also think of the gear-change lever on a scooter with gears (Kinetic Honda/TVS Scooty users please excuse!). When we simply press the lever, we are using the clutch. However, when we press the lever and turn it upwards or downwards, we are using the gears. Thus, we are getting different results from the same interface, by using it in a different manner. Again, the same interface is giving different results, depending on how we are using it.

Along similar lines, think of the [+] operator in VB. When we pass a string on either side of it, such as:

Z = "abc" + "def"

we will get the output in the variable 'Z' as

abcdef

If, on the other hand, we pass the code as

Z = abc + def

where 'abc' and 'def' are numeric variables, we will get the sum of 'abc' and 'def' in the variable 'Z'. This way, depending on the datatype on either side of the [+] operator, we get different results:

- If the data is a String, it will be concatenated.
- If the data is a numeric type, it will be added.

This is another example of polymorphism.

In Java, the function to get absolute value of any numeric type is 'abs( )'. In 'C', different datatypes have different functions for their absolute values:

- 'abs( )' for integer.
- 'labs( )' for long integer.
- 'fabs( )' for float, and so on.

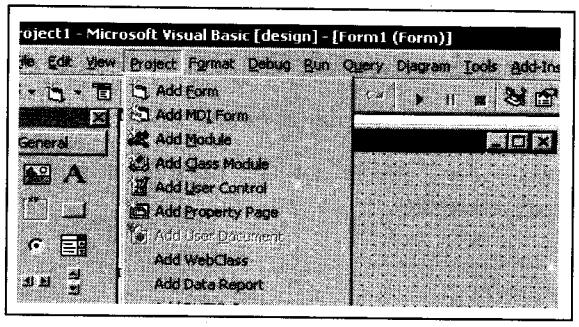
8.5 CREATING CLASSES USING CLASS MODULE

Having got a basic understanding of OOPS, we now start with coding for OOPS. The first step, as expected, will be the creation of a class in VB. For creating classes in VB, we use the class module. All of the OOP related task is carried out in the class module.

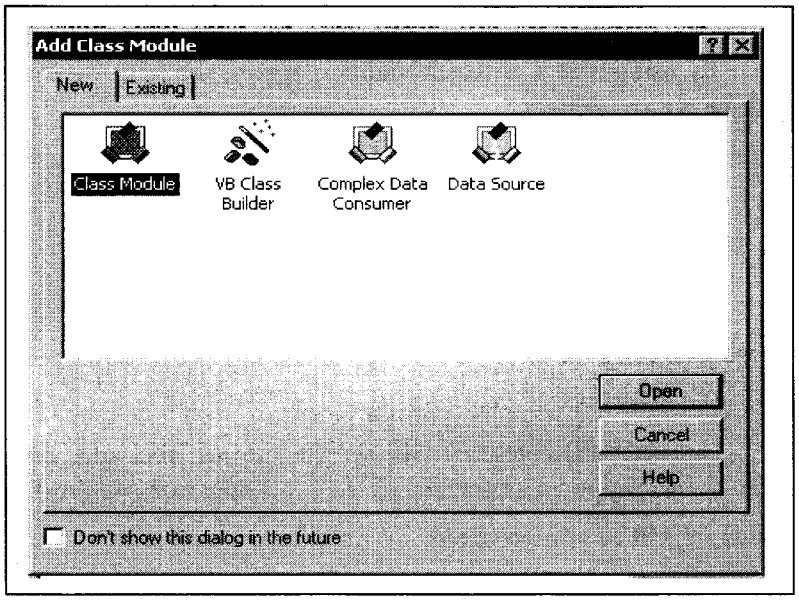
8.5.1 Adding Class Module to Project

To add a class module to a project, we perform the following steps:

1. Click on 'Project' 'Add Class Module' on the menu bar. This is shown in the figure:



Once we click as mentioned above, we get the following dialog box:



This is self-explanatory- either choose new or use an existing class.

2. Select the default option of 'Class Module' and click on 'Open'. We will now see the window for the class module opening up.

As we can see, the class module is just a code window, nothing else. An icon for the class module appears in the Project Explorer.

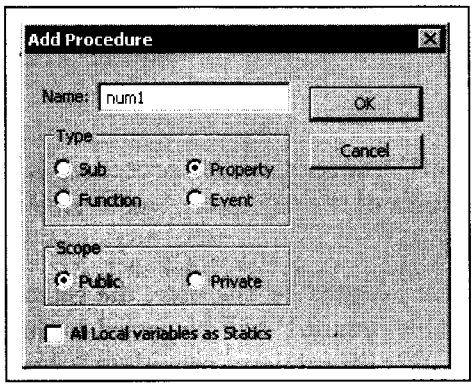
Now we start working on our class. Let's create a very simple class- to add two numbers. As we can understand, we will need three properties to work on:

- One property for number1
- One property for number2
- One property for storing the total

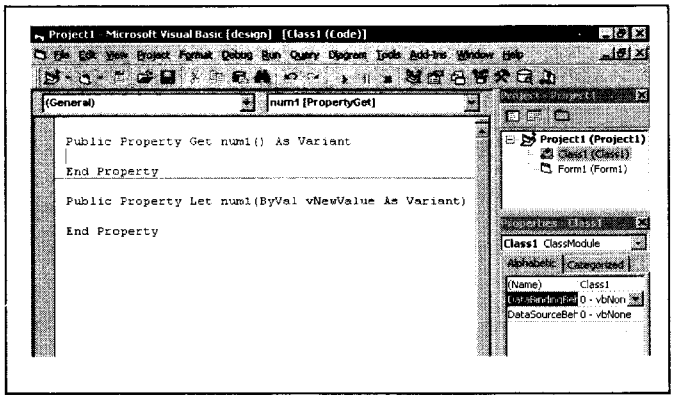
8.5.2 Adding Custom Properties

The first thing to do now is to assign some properties to the class. The newly-created class module is the starting point for OOPS. Now we will add two properties to our class, one for each number to add. To add a property to the class, we take the following steps:

1. Switch to code window and click on 'Tools' menu. A sub-menu appears.
2. Click on 'Add Procedure' from the sub-menu. An 'Add Procedure' dialog box appears as shown:
3. In the 'Name' textbox at the top, type the name of the property 'num1'.
4. Click on property option as the type. Keep the default scope of public. Here is the picture representing our entries and selections:



5. When we now click on 'OK', we get the code window as follows:



The Public Property Let num1 procedure will be called when we give a value to the 'num1' property. The Public Property Get num1 procedure will be called when VB reads that value from the memory. In other words,

- The let procedure will help us write a property value.
- The get procedure will help us read the property value.

This two-step process will help us in getting better control over the validation process for the property value.

Now go to the general/declarations section and type the following line:

```
PRIVATE mnum1 AS INTEGER
```

This line will create a class-level variable of Integer type. Since it is private, it can only be accessed from inside the class. This is a neat case of encapsulation, seen in the discussion on OOPS. This variable will help us to take in a value and then assign it to the property. Type the following code for the let and get property procedures:

Code Listing oop.1

RIVATE mnum1 AS INTEGER ' in general/declarations

```
PUBLIC PROPERTY GET num1 ( ) AS VARIANT
num1 = mnum1
END PROPERTY
```

```
PUBLIC PROPERTY LET num1 ( BYVAL vnewvalue AS VARIANT )
mnum1 = vnewvalue
END PROPERTY
```

When user writes a value into 'num1', the let property procedure is called alongwith the value set by the user. This value will be in the form of the parameter 'vnewvalue'. The value of 'vnewvalue' will be assigned to the private variable 'mnum1'.

When the computer reads the value assigned by the user, the get property procedure will be called. Here, the value of the private variable 'mnum1' will be passed to the property 'num1'. (Based on our knowledge of VB, you will observe that get property procedure behaves like a function).

Once property 'num1' is in place, we need to get the second property up and running. We repeat the steps mentioned above for adding a new property. Let us call our second property as 'num2'. We now code for num2:

Code Listing oop.2

```
PRIVATE mnum2 AS INTEGER ' in general/declarations
PUBLIC PROPERTY GET num2 ( ) AS VARIANT
num2 = mnum2
END PROPERTY
PUBLIC PROPERTY LET num2 ( BYVAL vnewvalue AS VARIANT )
mnum2 = vnewvalue
END PROPERTY
```

When user assigns a value to 'num2', let property procedure will be called alongwith the value set by the user. This value will be in the form of the parameter 'vnewvalue'. The value of 'vnewvalue' will be assigned to the private variable 'mnum2'.

When the computer reads the value assigned by the user, the get property procedure will be called. Here, the value of the private variable 'mnum2' will be passed to the property 'num2'.

8.5.3 Inheritance Peculiarities in VB

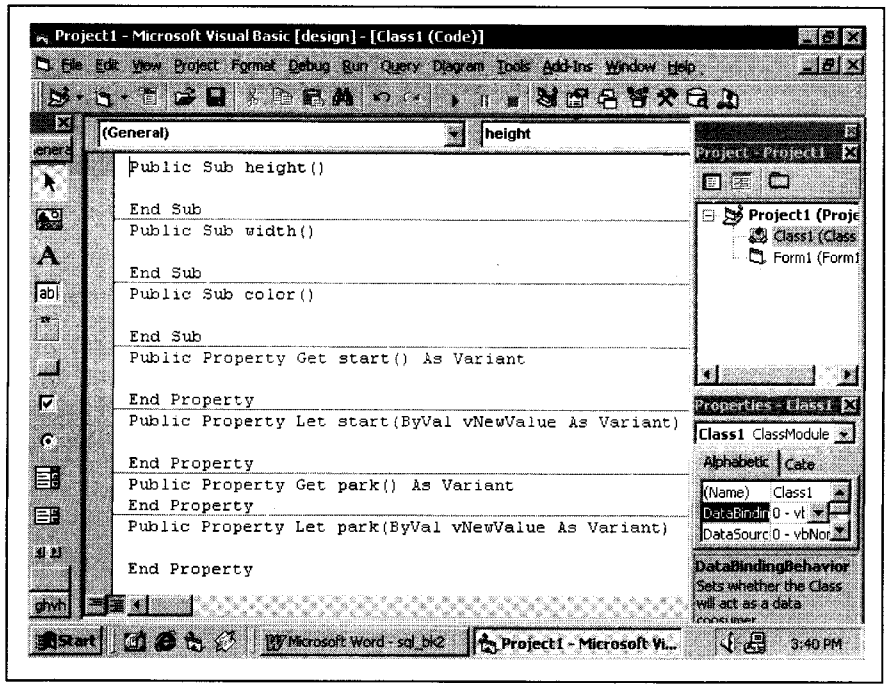
A major drawback with VB is that it does not support inheritance. In fact, inheritance, as we have discussed in OOPS, does not apply to VB in the real sense of the word. Thus, some experts say that VB is not a true object-oriented language, and they are right. After all, inheritance is one of the cornerstones of OOP, and VB's absence of support for inheritance is a major topic of debate between VB-lovers and VB-haters.

However, VB offers inheritance support from another perspective - though VB doesn't support implementation inheritance, it supports interface inheritance. This means that a method declared in a VB Class cannot be automatically inherited by another VB Class. Suppose we have a Class 'cls_aa' with these properties and methods:

- Start
- Park
- Colour

- Height
- Width

The coding for a class 'cls_aa' has been shown in this figure. Observe that we have created stubs for the methods and properties. However, we have not provided any code for the class.



8.5.4 Interface Class

The type of class we have just created is known as an interface class in VB; it only provides interfaces for other classes to use, without providing any implementation or coding of these interfaces. Thus, the interface class is like an abstract class- a class which provides behaviour, but no implementation.

Now, we create a new Class 'cls_bb' which will inherit the interfaces of 'cls_aa'. This way, 'cls_bb' will have to provide the code for all the properties and methods of 'cls_aa'. This is interface inheritance. The Class 'cls_bb' will inherit all the interfaces of 'cls_aa'. However, it will have to give its own coding of these interfaces.

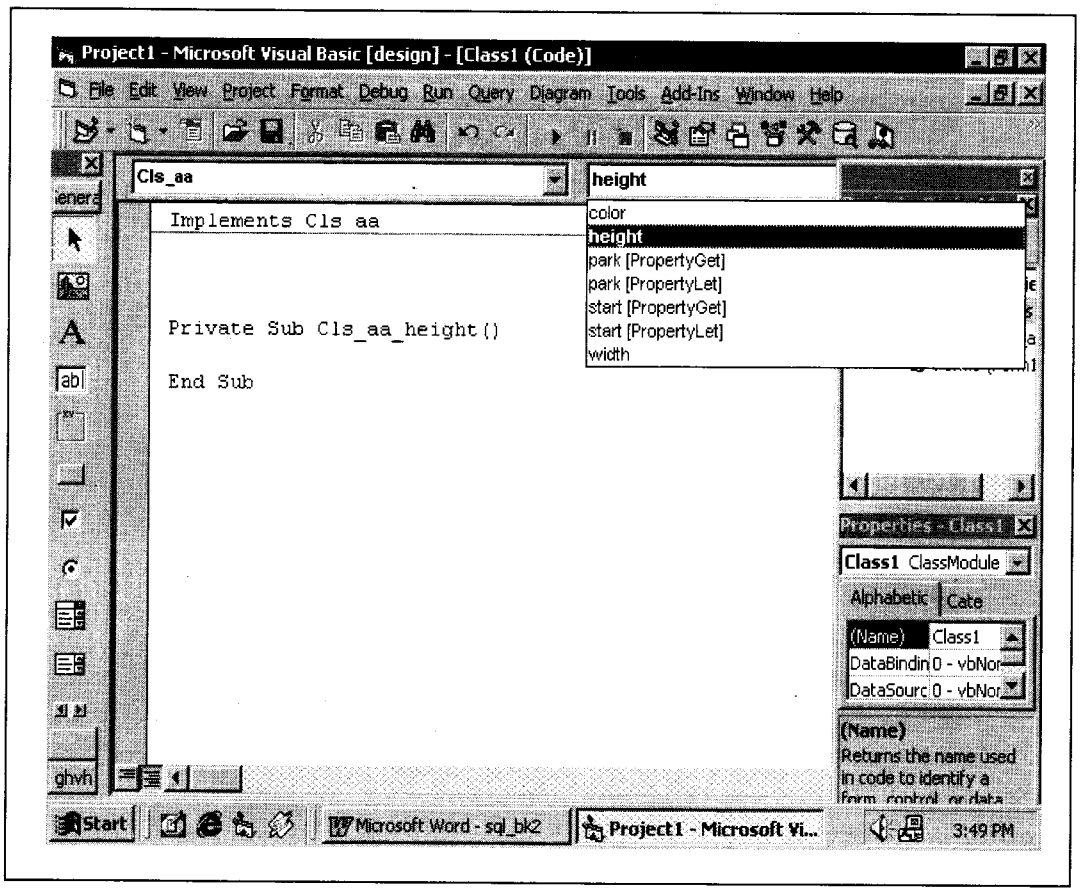
To allow 'cls_bb' to inherit the interface of 'cls_aa', we type this in the general/declarations of 'cls_bb':

IMPLEMENTS cls_aa

Once we type the line given above, the 'object' drop-down list in code window gets 'cls_aa' and the 'procedure' drop-down list gets all public interfaces of 'cls_aa'. The following figure shows interfaces of 'cls_aa' in 'cls_bb', as seen in the 'procedure' drop-down list at the top-right of the code window:

Now, in 'cls_bb' we have to code for all the interfaces of 'cls_aa'. If we run the program without coding for all the properties and methods that 'cls_bb' has inherited from 'cls_aa', the program will not compile.

Interface inheritance is the maximum that VB supports for inheritance. Implementation inheritance is supported by other OOP languages such as Java and SmallTalk. This drawback of VB has been removed in VB.NET.

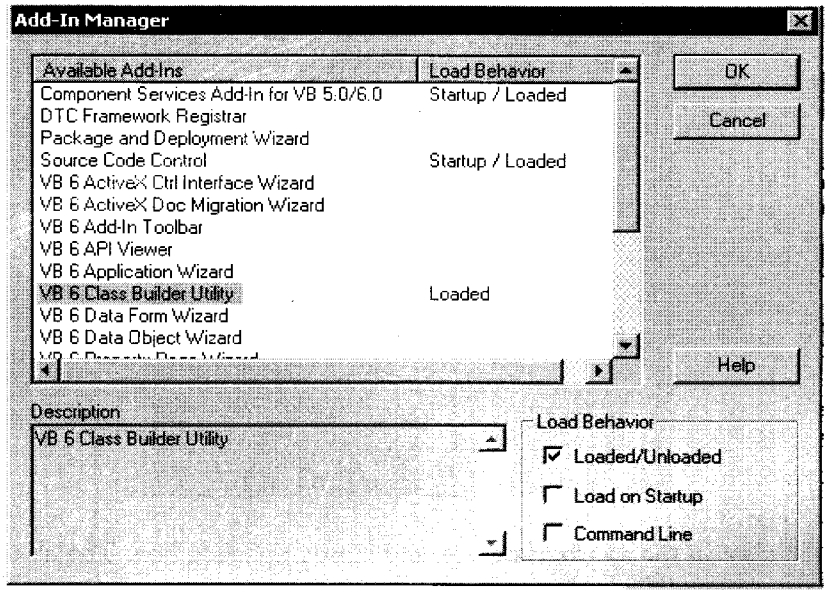


8.6 CREATING CLASSES USING CLASS BUILDER UTILITY

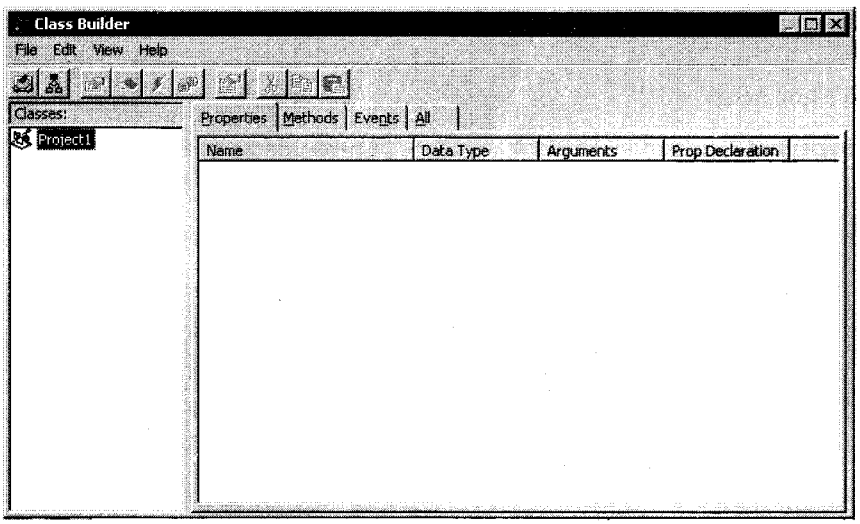
Apart from the manual method of creating a class, VB offers a useful tool for class creation in an easy manner- the Class Builder. The Class Builder automates the process of class building. It generates most of the code for us, helping us interactively in creating the members of the class.

8.6.1 To Access the Class Builder Utility

1. Click on the 'Add-Ins' menu.
2. From the sub-menus, choose 'Add-In Manager...'. An 'Add-In Manager' dialog box appears, showing the different wizards and add-ins available in VB.
3. Select the 'VB 6 Class Builder Utility'.
4. In the 'load behaviour' frame (towards the bottom-right) check the box marked 'Loaded/Unloaded'.
5. The dialog box has been reproduced here for your reference:



6. This time, select the 'Class Builder Utility' option from the sub-menus. The class builder utility appears as shown here:



On this screen, we see four tabs in the right pane: Properties, Methods, Events and All. As expected, these are for working with the properties, methods, events and all the data members, respectively.

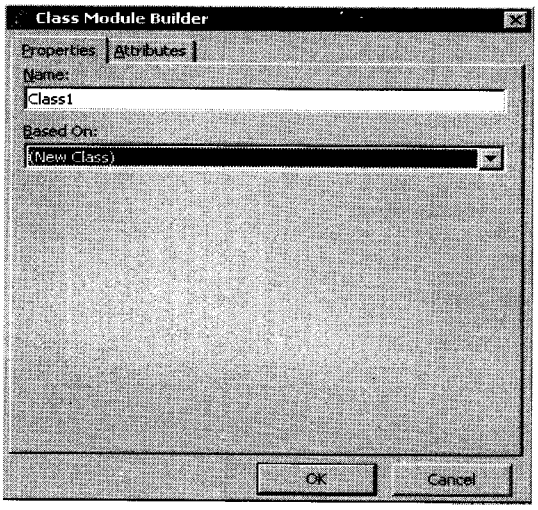
From this screen, we can choose one of the two options for the next step:

- On the 'File' menu, select 'New' 'Class' from the sub-menus.

  Or

- Right-click on Project Name in left window and select 'New' 'Class' from the pop-up list.

Whichever method we choose, we get the following screen to work, which helps add a new class to our project:

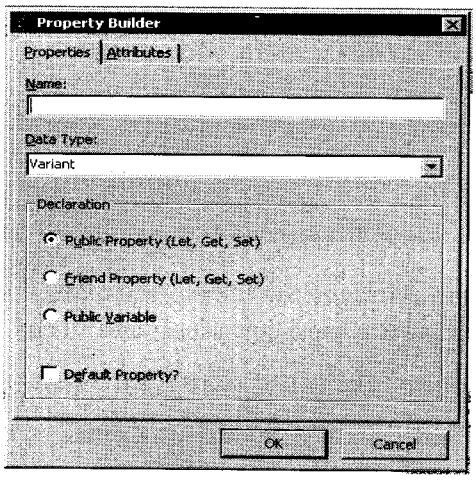


Type a name for the class in the appropriate box and close the dialog box. The class name will appear in the left pane of the class builder utility.

1. Now close the dialog box.
2. Once again, go to the 'Add-Ins' menu.

8.6.2 Adding a Property to the Class

1. Come to the right pane.
2. Select the Properties tab, and right-click on it. A pop-up menu with an option of 'New Property' appears.
3. Click on this option, and the following 'Property Builder' screen appears:



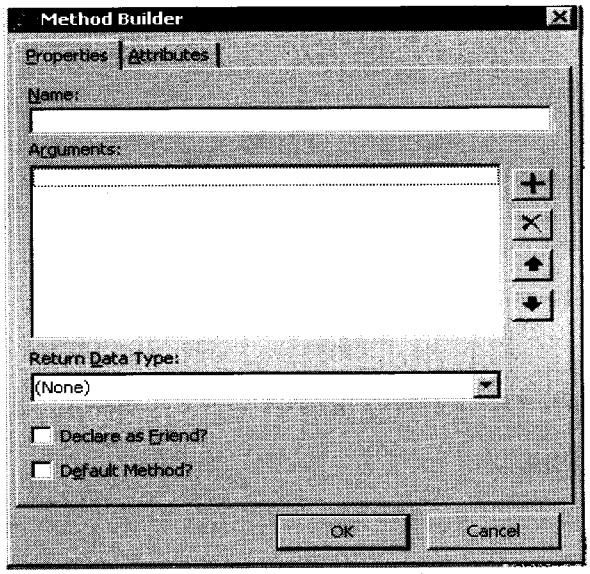
As we can see, this dialog box will help add new properties to our class. On the Properties tab, we:

- Set the name and datatype of the property.
- Specify whether the property will be a public property/friend property/public variable.

- Specify if the current property is the default property for the class by checking on the box at the bottom.
- On the Attributes tab, we specify a short description for each of the properties being created by us.

8.6.3 Adding a Method to the Class

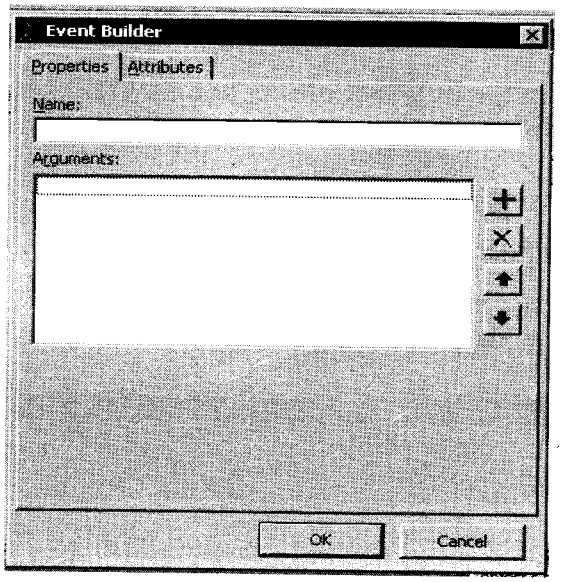
1. Come to the right pane of the class builder.
2. Select the Methods tab, and right-click in the right pane. The following 'Method Builder' dialog box pops up, for adding methods to the class:



3. Through this dialog box we:
   - ❖ set the name of the method.
   - ❖ give its arguments.
   - ❖ give its return type.
   - ❖ alter sequence of the parameters (from the arrows at the right side).
   - ❖ specify if a method is a friend.
   - ❖ specify if the method is the default method for a class.
   - ❖ on the attributes tab, specify a short description for the method.

8.6.4 Adding an Event to the Class

1. Once again, come to the right pane of class builder,
2. Select the Events tab, and right-click on the right pane. The event builder dialog box for adding events pops up, as shown here:



3. On the Properties tab, we:
   - ❖ Set the name of the event.
   - ❖ Set its arguments.
   - ❖ On the Attributes tab, specify a short description for the event.

8.6.5 Exiting the Class Builder

Once all the properties, methods and events have been added, we click on 'File' 'Exit' on the menu bar. VB asks if we want to update our project with the changes. Once we click 'Yes', the Class Builder will build the code for the class, based on the way we have configured it.

Check Your Progress

1. Define class and object.
2. What is Abstraction?

8.7 LET US SUM UP

In OOPS, a basic thinking is to segregate, i.e., isolate the various units of a system. This way, we have the benefit of flexibility in the system- we can change one sub-system with either little or no changes at all in the associated sub-systems. In abstraction, instead of looking at the finer details of an application, we look at a big system as being composed of many small sub-systems. 'Data-hiding' is another word for encapsulation. Encapsulation refers to the fact that the 'interface' and the 'implementation' are totally separated from each other. In inheritence, a class can 'inherit' another class. The big advantage of this approach is that we get a certain amount of functionality readymade. The basic concept behind polymorphism is 'One interface for multiple methods'.

8.8 KEYWORDS

OOPS: In OOPS, a basic thinking is to segregate, i.e., isolate the various units of a system.

Class: It defines the structure and behavior of the contained data.

Object: It is defined as an instance of the class.

Abstraction: It basically means how we, as human beings, deal with complexity.

Encapsulation: The data and the code is kept 'concealed', i.e., 'hidden' from the outer world.

Inheritence: Aclass can 'inherit' another class.

Polymorphism: this term means 'many forms'.

Interface Class: The interface class is like an abstract class- a class which provides behavior, but no implementation.

8.9 QUESTIONS FOR DISCUSSION

1. What is object oriented system? How is it different from other programming techniques?
2. What is encapsulation? Discuss different access specifiers.
3. What is inheritance? Discuss its advantage.
4. Discus the steps for adding class modules to project.
5. Discuss the major drawback of visual basic.

Check Your Progress: Modal Answers

1. A class is simply a container for code and data. It defines the structure and behaviour of the contained data. An object, on the other hand, is an instance of the class. The object, thus, puts the structure and behavior of the class to use.
2. Abstraction basically means how we, as human beings, deal with complexity. It means the same thing in OOPS, too. Instead of looking at the finer details of an application, we look at a big system as being composed of many small sub-systems.

8.10 SUGGESTED READINGS

Sanjeev Sharma, *Visual Basic 6*, Excel Books, New Delhi

A. Chakraborti et al., *Microsoft.Net Framework*, PHI, 2002

M. Reynolds et al., *.Net Enterprise*, Wrox/SPD, 2002

Vikas Gupta, *.Net Programming*, Dreamtech Publication

Mackenzie Sharkey, *Teach yourself Visual Basic.Net in 21 days*, SAMS Techmedia

Bible, Bill Evjen, Jason Beres et al., *Visual Basic.Net Programming*, Wiley India

LESSON

9

GRAPHICS AND FILE HANDLING, OTHER CONTROLS

CONTENTS

9.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Discuss graphics which include two controls picture box and image box
- Discuss file handling
- Understand validation controls and web controls
- Discuss html controls

9.1 INTRODUCTION

Graphics controls include picture box and image box which are used to display pictures. These controls have been discussed in this lesson.

Working with VB, we have access to full-fledged data libraries like DAO and ADO. However, there might be situations where we need a very basic data access tool. The heavy system requirements of DAO or ADO might not really be required for such a situation. In this situation, we can utilize VB power to create very simple files for holding data, maybe even text files.

We will also discuss validation controls, some web controls and html controls.

9.2 GRAPHICS

9.2.1 Graphics Controls

VB offers 2 important controls for graphics:

- PictureBox
- ImageBox

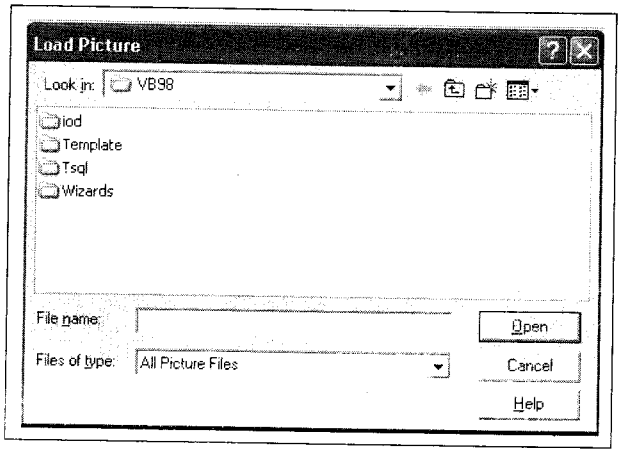
Though both the controls are meant to display pictures, here are some important differences between them:

1. Similar to the frame, the PictureBox can also act as container control. The ImageBox cannot act as a container control.
2. The PictureBox has a Boolean property, autosize. If set to true, the PictureBox will change in size to fit the image perfectly.
3. The ImageBox, has a Boolean property stretch. If set to true, it will force the image to be 'stretched' as per the size of the box.

Thus, in the PictureBox the image will stay in the original dimension. In the ImageBox, the image can be stretched either horizontally or vertically or both, thus distorting the original picture. The PictureBox doesn't have the stretch property. The ImageBox doesn't have the autosize property.

To assign an image to these controls, we set picture property. For this,

1. We select the control.
2. Select the Picture property.
3. Click on the ellipsis at the right, the "Load Picture" (Windows File Open) dialog box pops up.

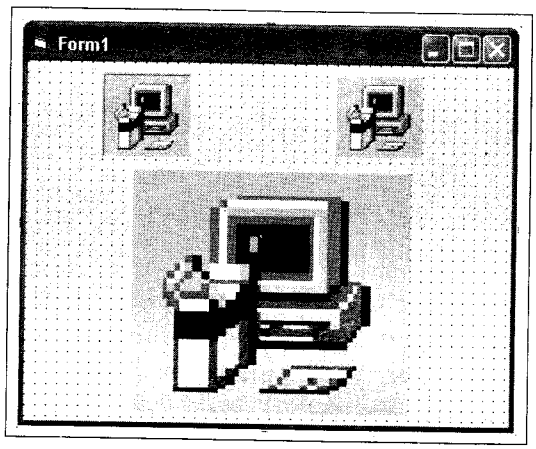


4. Once we select the picture and click on "Open", the image is displayed in the control. To set the picture at runtime, we use the LOADPICTURE method, passing the filename as a parameter:

   PictureBox1.PICTURE = LOADPICTURE(complete path of the image file)

Calling on loadpicture without any argument will remove the current picture from the control.

Here is a picture showing 1 PictureBox and 2 ImageBoxes. In the ImageBox at the bottom, the stretch is true. As we can observe, the picture in it is much bigger than the other controls. Other controls show the picture in original dimension.



When an image is assigned to one of these controls at design-time, it becomes a part of the application and is stored in a .frx file. This way, the size of the application increases considerably.

9.2.2 Graphics Methods

Print

This method draws text on a PictureBox or on a form, starting from the current point.

Picture1.PRINT "This is some text"

This line will print the message 'This is some text' on PictureBox control Picture1. The Print command includes a 'newline character'. Thus, the next Print command will print the new text line under the previous text.

Drawing Lines

The line method is used for drawing lines.

LINE (X1,Y1) - (X2,Y2)

where,

x1 and y1 are the coordinates of the line's starting points.

x2 and y2 are the coordinates of the line's ending points.

Here are some examples of using the line method:

```
LINE (20, 20) - (140, 140)  'draws a line in the default forecolor
LINE (20, 20) - (140, 140), RGB (0, 0, 255)     'draws a blue line
```

Drawing Boxes

In the Line method, VB offers the 'B' option (Box) to draw a box.

```
LINE (20, 20) - (140, 140), B
'draws a box based on line coordinates
'colour has been omitted after coordinates
```

Drawing Circles

The general syntax for drawing a circle is:

CIRCLE [STEP], (X, Y), RADIUS, [COLOUR], [START], [END]

Based on this, the simplest form of drawing a circle is:

CIRCLE (X, Y), R

The following code draws a circle in the centre of the form form1:

CIRCLE (form1.SCALEWIDTH/2.form1.SCALEHEIGHT/2, form1.SCALEHEIGHT/3

ScaleHeight/ScaleWidth refers to a control's inner dimensions in terms of the unit assigned to the coordinates system (twips, inches, etc). For example, a control with height and width of 4320 twips will have a height and width of 3 inches. If we change the coordinate system to inches, the size will remain the same, but ScaleHeight and ScaleWidth will become 3 (inches).

Setting Colours

Pre-defined Colours

VB offers a wide range of pre-defined colours in an easy-to-remember manner: just prefix the letters 'vb' to the name of colour desired, such as:

- VBRED
- VBGREEN
- VBCYAN, etc

User-defined Colours

VB offers the flexibility of specifying our own colours, based on a mix of the three basic colours. The range of each of the three colours is from 0 (minimum) to 255 (maximum). The order of the colours is always in RGB format (Red Green Blue), such as:

- RGB (225,0,0)- Red
- RGB (0,225,0)- Green
- RGB (0,0,225)- Blue

Any combination of RGB can be tried until we arrive at the right combination according to our liking.

9.3 WORKING WITH FILES

Here are some commonly used formats for working with files.

9.3.1 Sequential Files

This type of file organization involves data storage with no reference to the order in which it is stored. The data can be read only in the manner in which it was entered. For accessing a particular record, say, the 5th, we have to first skip over the four records lying in front of it. Thus, this is a data reading method which might be very slow and cumbersome. Imagine reading the 996th record of a 999-record file!! This means that a sequential file is similar to an audio cassette-to listen to the 4th song, we need to skip over the first three songs. However, this is a very good solution if we plan for a single pass through all records, processing them one-by-one. In applications where the order of records is not important and each record needs to be processed just once, we opt for this method. Basically, any text file with variable-length strings can be treated as a sequential file.

9.3.2 Random-Access Files

True to their name, these are organized on the basis of quick access to the contents. In this mode, we can directly access any record on the basis of some 'key'. In this respect, the random-access files are like CDs- we need not read the intervening data to go to a particular record in the file. This is more efficient than sequential access for heavy search-intensive operations. Here, the file is organized into records, usually of the same length. This length will be helpful in 'jumping directly' to a specific record in the file.

The Open Statement

Sequential and random-access files share some common features. We use Open for both the types. The kind of file access we get depends on the arguments we use in Open. Open reserves a file handle, or a channel, for reading/writing to a file. The Open statement links a number to the handle. When this file handle is open, the data stream can flow. The mode in which we open the file number - read, write, or both - dictates how the data stream can flow between the file and the computer.

A file handle is a unique path to a file associated with a number from the Open statement. Once Open associates a file number with the file handle, the rest of the program uses the file number to access the file. Here's the Open statement's format:

Open strFileName [For Mode] [AccessRestriction] [LockType]_As

```
[#]intFileNum [Len = intRecordLength]
```

When we run a program and assign a data file to a number, the program will use that file number to access the file. Our program never again has to use the filename to get to the file. All Open statements must include filename and file number arguments, but other arguments are optional. A simple call to Open with only the required parameters might be like this:

```
OPEN "afile.txt" AS #1
```

This statement opens a file named afile.txt as file number '1'. To refer to this file for input or output, we will refer to the file number. This file has been opened in random-access mode, because Random is the default value if we don't mention the For Mode in open statement.

Learning the File Modes

Mode is a special keyword that indicates file's access mode. The following table explains the values we can supply for the Mode argument.

Mode	Description
Append	Opens file for sequential output, starting at the end of the file. If the file doesn't exist, the file is created. Append doesn't overwrite the existing data.
Binary	Opens a file for binary data access. Here, we access a file at the byte level, meaning that we can write and read individual bytes to and from the file.
Input	Opens a file for sequential input, starting at the beginning of the file. Data is read in the same order in which it was sent to the file.
Output	Opens a file for sequential output, starting at the beginning of the file. If the file doesn't exist, VB creates the file; otherwise, VB overwrites it.
Random	Opens a file number to a file for random read and write access. Allows data to be read from and written to a file at any specific record.

Thus, the mode for opening a file determines the operation that can be performed on it. The following statements demonstrate how to use various modes for opening a file:

OPEN "filinpdt.txt" FOR INPUT AS #1

OPEN "append.txt" FOR APPEND AS #1

OPEN "output.txt" FOR OUTPUT AS #1

OPEN "random.txt" FOR RANDOM AS #1

The last statement means the same as the following:

OPEN "random.txt" AS #1

Error-handling is very important for these operations. Anytime a file is opened/accessed, an error can be raised. Good error-handling will help our application exit gracefully instead of confusing our users with nasty error messages.

Restricting Access

The optional Access Restriction argument in an Open statement helps restrict the access to Read, Write, or Read Write. This restriction is used when writing programs that run across a network. The values are as follows:

Value	Description
Read	File contents can be seen but not modified
Write	File contents can be modified
Read/Write	Both tasks can be done

Locking the File

The lock type argument specifies operations allowed on the open file by other processes. This is important in network applications. We restrict access to a file so that only one user at a time has read/write access to the file. This helps avoid a situation where two users are making changes at the same time. Otherwise, changes made by one of the users are bound to be lost.

The valid options for Lock Type are as follows:

LockType	Meaning
Shared	all users access the file simultaneously
Lock Read	only the person with file open for reading can access the file
Lock Write	only the person with file open for write access can write to the file
Lock Read Write	locks file from all users except the one who has file open for read and write access

Managing the Record Length

In random-access files, the length specified by the Len = intRecordLength option is used as the size of records that will be passed from Visual Basic to the file. This size is necessary when accessing records. The first record begins at location 1, and all subsequent records are written at locations in increments of 1. The actual location in the file of a given record is N * intRecordLength, where N is the record number.

A record is one logical line from a file that holds a complete set of data. For example, if our file holds inventory data, one record would be one item's inventory information, such as the description, price and quantity.

Accessing records operates similar to the way we access arrays. Whereas the first element in an array is stored in Array(0), the first element in a file is stored at record number 1. To make index coordination between arrays and files easy, we use Option Base 1 in the General/Declarations section.

Locating a Free File Number

Using VB, we can open multiple files at the same time, each file being assigned a different file number. Thus, it is important to keep track of the next available number when opening a file. VB has the FreeFile() function to help find the next available file number. We can be sure that the number given by this function has not been used so far. Here's the syntax:

FREEFILE [(intRangeNumber)]

The optional intRangeNumber parameter let's specify that we want the returned file number to be in a specific range: 1-255 or 256-511. The default range, if no parameter is given, is 1-255. Mostly, the default option is alright because we very rarely open more than 256 files.

The following lines use FreeFile() to obtain a file number and then open a file with that number:

```
intFileNumber = FreeFile
OPEN "ACCPAY.DAT" FOR OUTPUT AS INTFILENUMBER
```

We use FreeFile() to make sure that a file number we're about to use isn't already used up. Even in small programs, FreeFile() is a good tool to make sure we don't use the same file number more than once at a time.

We should avoid the shortcut of using FreeFile() with the Open statement:

```
OPEN strfilename FOR OUTPUT AS FREEFILE( )
```

Although this code will work, we cannot clearly know the file number for future operations on the file. It is far better to declare a numeric variable and store the value for further use.

Specifying the Close Statement

All the files opened with Open statement need to be closed at some stage. The statement used for closing a file is, as expected, Close. Close needs just one parameter - the open file number. Here's the syntax:

```
CLOSE # INTFILENUMBER[, INTFILENUMBER2][,...INTFILENUMBERX]
```

Any number of files can be closed in a single Close statement. If we don't give a file number, all open files are closed. This can be useful for stopping our applications in a single step.

This code opens two sequential files - one for reading and one for writing - using the next available file numbers. It then closes both files when done.

Code Listing file.1

```
1.  DIM intReadFile AS INTEGER, intWriteFile AS INTEGER ' Handle input file
2.  intReadFile = FreeFile                'Get first file #
3.  OPEN "AccPay.Dat" FOR INPUT AS intReadFile 'Handle output file
4.  intWriteFile = FreeFile( )             'Get next file #
5.  OPEN "AccPayOut.Dat" FOR OUTPUT AS intWriteFile
    'Some code goes here
    'to send the contents of
    'the input file to the output file
6.  CLOSE intReadFile
7.  CLOSE intWriteFile
```

We haven't used an actual file number here, since FreeFile() in lines 2 and 4 returns the available file numbers and these numbers are stored as named integers (variables).

If we don't close all open files, the file might incur some damage. Generally, if power goes out when a file is open, the file's contents might be corrupted or lost. Therefore, close a file as soon as it is not needed. If we don't close a file, it is closed when our application terminates.

Following our discussion on file types, we now go through some sample code for Creating /Opening/Reading a sequential file. In this code, we use the 'Open' statement for opening a file. If the file does not already exist, it is created for us.

Code Listing file.2

```
PRIVATE SUB command1_CLICK( )
   1. DIM rt AS STRING
   2. OPEN "d:\sanju\te.txt" FOR APPEND AS #1
'opened in append mode, adds to end of existing text
   3. WRITE #1, TEXT1
   4. CLOSE #1
   5. OPEN "d:\sanju\te.txt" FOR INPUT AS #2
'file opened for reading
   6. DO UNTIL EOF(2)
   7. LINE INPUT #2, rt
'read one line at a time
   8. MSGBOX rt
   9. LOOP
   10. CLOSE #2
END SUB
```

We can try the following variations in the code given above:

To open the file in write mode, where the existing text will be erased, change line no. 2 to:

OPEN "d:\sanju\te.txt" FOR OUTPUT AS #1

To read 8 characters from the file identified by #2, change line no. 9 to:

text2.TEXT = INPUT$ (8, #2)

9.4 VALIDATION CONTROLS

The key to validation in the .NET world is a set of controls called validation controls.

There are 5 types of individual validation controls. They are:

- Required Field Validator
- Compare Validator
- Range Validator
- Regular Expression Validator
- Custom Validator

The Required Field Validator makes sure the user enters something. It doesn't have to be anything in particular, but they can't leave the field blank. The Compare Validator and the Range Validator both

compare things using equality comparisons (the is x>=y type of thing). They only differ in that the Compare Validator is one-sided while the Range Validator allows you to specify both a lower and an upper bound. The Regular Expression Validator validates input against a regular expression. And if nothing else works for us, we can write our own criteria and encapsulate it in a Custom Validator.

In addition to these individual validation controls, there's one more kind: the Validation Summary control. This control is a great little touch by the folks from Redmond and allows us to easily check if every validator on a page is satisfied or not instead of having to check them all individually.

The validation Controls available in .Net Framework are given below:

Control	Description
RequiredFieldValidator	Ensures that the user enters data in the associated data-entry control
CompareValidator	It compares user-entered data to a constant value or the value in another data-entry control using Uses comparison operators
RangeValidator	Ensures that the user-entered data is in a range between given lower and upper bounds
RegularExpressionValidator	Ensures that the user entered data matches a regular expression pattern
CustomValidator	Ensures that the user-entered data passes validation criteria that you set yourself

9.5 WEB CONTROLS

9.5.1 Calendar

The Calendar control allows user to display a month calendar that is used to select dates and move to the next and previous months.

You can mention whether the user can select a single day, a week, or a month, or you can disable date selection entirely by setting the Selection Mode property.

The appearance of the Calendar control can be managed by setting the style properties for the various parts of the control.

9.5.2 AdRotator

The AdRotator control is used to present ad images whch moves to a new Web location, when clicked. An ad is randomly selected from a predefined list .The following sample illustrates using the AdRotator control.

VB AdRotator1.aspx

The Advertisement File property of the AdRotator control specifies the path to this file.

For creating the advertisement file, opening and closing <Advertisements> tags shows the beginning and the end of the file, respectively. Opening and closing <Ad> tags delimit each advertisement. All advertisements are displayed between the opening and closing <Advertisements> tags.

9.6 HTML CONTROLS

We have given below the list of the HTML controls:

- *HtmlAnchor:* It is used for giving hyperlinks, besides other things.
- *HtmlImage:* It is used for inserting an image.

- ***HtmlInputHidden:*** It allows you to store the information in a non viewable control on the form.
- ***HtmlSelect:*** It is used to select data.
- ***HtmlButton:*** It is a server-side control that maps to the <button> HTML element and allows you create push buttons.
- ***HtmlInputButton:*** The HtmlInputButton control (<Input type=button>) is similar in function to the <button> tag, except that it can target any browser.
- ***HtmlInputImage:*** An HtmlInputImage control is used to create a graphical button. It is functionally similar to an HtmlButton control. This control enables programming of the HTML <input type=image> element. This control is typically used together with other user input controls.
- ***HtmlTable:*** It is used to show content in a tabular format.
- ***HtmlForm:*** It specifies an HTML form for accepting user data.
- ***HtmlInputCheckBox:*** The HtmlInputCheckBox control enables programming of the HTML <input type=checkbox> element. The control accepts boolean (*true/false*) input. Its Checked property returns *true*, when checked.
- ***HtmlInputRadioButton:*** The HtmlInputRadioButton control is used to control an <input type="radio"> element. This element is used to create a radiobutton.
- ***HtmlTextArea:*** HTML server controls that are added from the Toolbox to a page in Visual Studio are simply HTML elements with certain attributes already set.
- ***HtmlGeneric:*** It is a server-side control that maps to an HTML element not represented by a specific .NET Framework class, such as <body> and .
- ***HtmlInputFile:*** It enables programming of the HTML <input type=file> element. With this control, you can allow the upload of binary or text files from a client browser to the server.
- ***HtmlInputText:*** It is a server-side control that maps to the <input type=text> and <input type=password>.

Check Your Progress

1. What does LockType argument specify in file handling?
2. What are random access files?

9.7 LET US SUM UP

VB offers 2 important controls for graphics: PictureBox and ImageBox. The Picture Box can act as container control where as the ImageBox cannot act as a container control. In the PictureBox the image will stay in the original dimension. In the ImageBox, the image can be stretched either horizontally or vertically or both, thus distorting the original picture. Print method draws text on a PictureBox or on a form

Sequential file organization involves data storage with no reference to the order in which it is stored. The data can be read only in the manner in which it was entered. In random access files, we can

directly access any record on the basis of some 'key'. In this respect, the random-access files are like CDs- we need not read the intervening data to go to a particular record in the file.

9.8 KEYWORDS

Graphics: Graphics are meant to display pictures.

RequiredFieldValidator: Ensures that the user enters data in the control.

CompareValidator: Uses comparison operators to compare user-entered data to a constant value.

RangeValidator: Ensures that the user-entered data is in a range.

CustomValidator: Ensures that the user-entered data passes validation criteria.

AdRotator: It is used to present ad images.

9.9 QUESTIONS FOR DISCUSSION

1. Discuss the controls for graphics. Differentiate between them.
2. What are the different graphics methods used? Discuss.
3. Discuss the process of working with files.
4. What are different validation controls in .Net framework?
5. List various HTML controls available in .Net.

Check Your Progress: Modal Answers

1. The LockType argument specifies operations allowed on the open file by other processes. This is important in network applications. We restrict access to a file so that only one user at a time has read/write access to the file.
2. These are organized on the basis of quick access to the contents. In this mode, we can directly access any record on the basis of some 'key'. In this respect, the random-access files are like CDs- we need not read the intervening data to go to a particular record in the file.

9.10 SUGGESTED READINGS

Sanjeev Sharma, *Visual Basic 6*, Excel Books

A. Chakraborti et al., *Microsoft.Net Framework*, PHI, 2002

M. Reynolds et al., *.Net Enterprise*, Wrox/SPD, 2002

Vikas Gupta, *.Net Programming*, Dreamtech Publication

Mackenzie Sharkey, *Teach yourself Visual Basic.Net in 21 days*, SAMS Techmedia

Bible, Bill Evjen, Jason Beres et al., *Visual Basic.Net Programming*, Wiley India

UNIT V

LESSON

10

DATA ACCESS

CONTENTS

10.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Discuss data access with ado.net
- Discuss simple binding and complex binding for binding controls to databases
- Discuss database access in web application

10.1 INTRODUCTION

Data access is needed by most of the applications at one point of time making it an important component when working with applications. Data access is basically defined as the interaction of application with a database, where all the data is stored. Every application has its own needs for

database access. VB.NET uses ADO .NET (Active X Data Object) as it's data access and manipulation protocol which also enables us to work with data on the Internet.

10.2 THE ADO.NET DATA ARCHITECTURE: DATA ACCESS

Data Access in ADO.NET is based on two components: DataSet and Data Provider.

10.2.1 DataSet

DataSet is defined as the collection of database tables. The Data Set is essentially an in-memory database. The dataset is a disconnected representation of data which acts as a local copy of the relevant portions of the database. The DataSet is located in memory and the data in it can be manipulated and updated independent of the database. The data from the database is filled into the dataset by using sqlDataAdapter. The data in DataSet can be loaded from data source like SQL server database, an Oracle database or from a Access database. DataSet is finished, changes can be made back to the central database for updating.

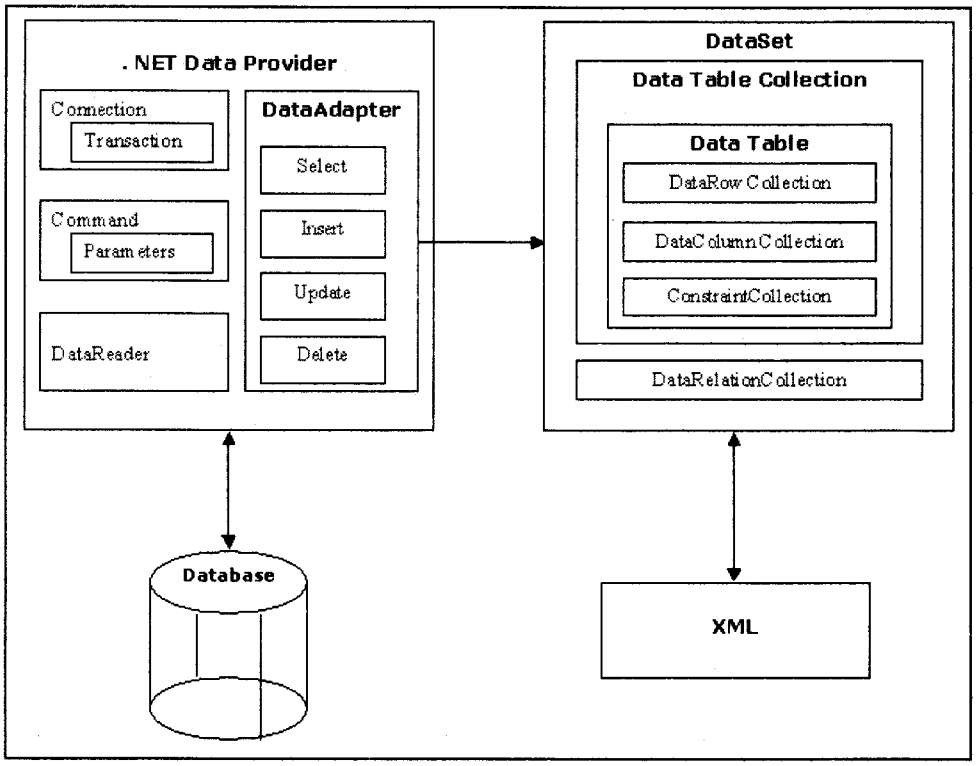
10.2.2 Data Provider

A Data Provider is a set of related components that work together to provide data in an efficient manner. It results in a performance driven manner. The Data Provider basically provides and maintains the connection to the database. The .NET Framework contains two Data Providers: the SQL Data Provider and the OleDb Data Provider. SQL Data Provider is available only to work with Microsoft's SQL Server 7.0 or later and the OleDb Data Provider permits to connect with other types of databases like Access and Oracle. Each Data Provider includes the component classes which are defined below.

- *The Connection object:* It provides a connection to the database
- *The Command object:* It executes a command
- *The Data Reader object:* It provides a forward-only, read only, connected recordset
- *The Data Adapter object:* It provides a disconnected DataSet with data and it performs updation.

Data access with ADO.NET can be briefed as follows:

The command object provides direct execution of the command to the database. A connection object provides the connection to the database for the application. If more than a single value is returned by the command, the command object returns a Data Reader to provide the data. So, The Dataset object can be filled by the Data Adapter. The database can be updated by using the command object or the DataAdapter.



Source: http://www.startvbdotnet.com/ado/default.aspx

Figure 10.1: ADO.NET Data Architecture

10.2.3 Component Classes that make up the Data Providers

The Connection Object

The Connection object provides the connection to the database. The Connection object contains all of the information which is required to open the connection to the database. Microsoft Visual Studio .NET includes two types of Connection classes: the SqlConnection object, which involves connection to Microsoft SQL Server 7.0 or later, and the OleDbConnection object, which provide connections to a wide range of database types like Microsoft Access and Oracle.

The Command Object

The Command object is includes two corresponding classes. They are SqlCommand and OleDbCommand classes. Command objects basically execute commands to a database across a data connection. The Command objects can be used to execute stored procedures on the database, SQL commands, or return complete tables directly. Command objects provide three methods for the purpose of executing commands on the database:

- *ExecuteNonQuery:* It is used to execute commands with no return values like INSERT, UPDATE or DELETE.
- *ExecuteScalar:* It is used to return a single value from a database query .
- *ExecuteReader:* It is used to return a result set by way of a DataReader object.

The DataReader Object

The DataReader object represents a forward-only and read-only result set. DataReader objects cannot be directly instantiated, unlike other components of the Data Provider. As a result of the Command object's ExecuteReader method, the DataReader is returned. The SqlCommand.ExecuteReader method returns a SqlDataReader object, and the OleDbCommand.ExecuteReader method returns an OleDbDataReader object. The DataReader can provide rows of data directly to application logic when you do not need to keep the data cached in the memory.

The important methods supported by the Data Reader Object are:

(a) Read()

(b) GetString()

(c) GetFloat(), etc.

The DataAdapter Object

The Data Adapter object is used to fill the dataset. The DataAdapter is defined as the core of ADO.NET's disconnected data access. DataAdapter serves as a bridge between sql server database and dataset. It acts as a middleman fproviding all communication between the database and a DataSet. The DataAdapter is used either to fill a DataTable or DataSet with data from the database. This is done by using Fill method. For performing any updations, DataAdapter can do the changes to the database by calling the Update method. DataAdapter interacts with database for dataset. The DataAdapter is provided with the following properties that represent database commands:

(a) SelectCommand

(b) InsertCommand

(c) DeleteCommand

(d) UpdateCommand

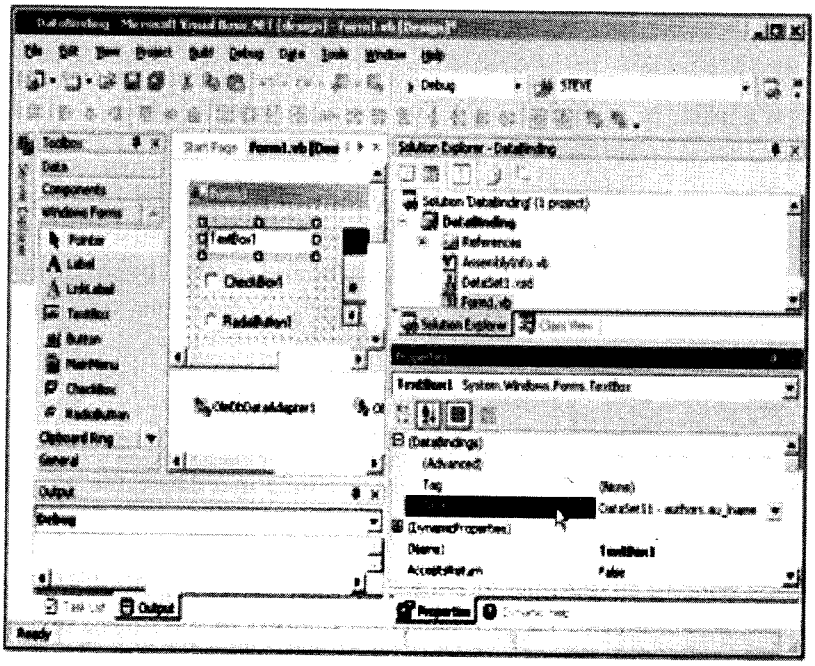
10.3 BINDING CONTROLS TO DATABASES

Binding controls to database involves simple binding and complex binding to databases.

10.3.1 Simple Binding

In Simple binding you can show only *one* data element, such as a field's value from a data table, in a control. In visual basic.net, you can bind any property of a control to a data value.

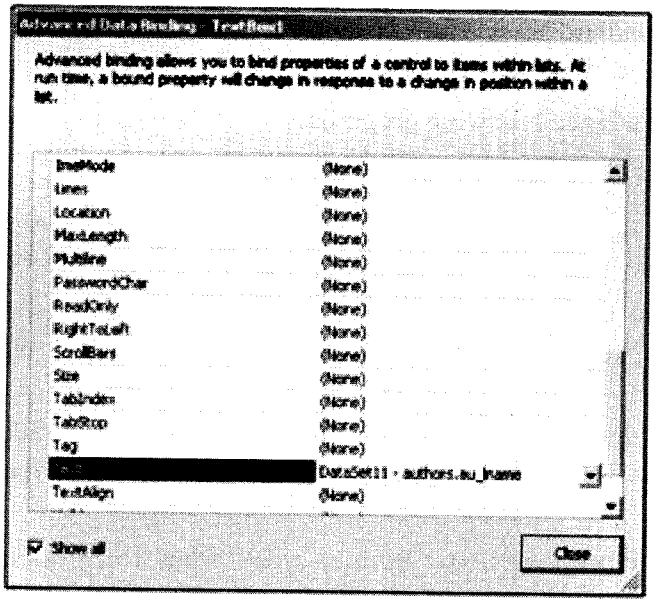
Let us show an example where we bind a Text property of the textbox to the name field in the authors table from the pubs database in a dataset, DataSet11, where you select the text box and expand its (Data Bindings) property in the Properties window. You can select a field in a dataset which is to bind just by clicking a property and selecting a table from the drop-down list that appears.



Source: http://www.yaldex.com/vb-net-tutorial-2/library.books24x7.com/book/id_5526/viewer.asp@ bookid = 5526&chunkid=0925001398.htm

Figure 10.2: Binding a Text Box to a Data Table

As mentioned, you can bind to any property of a control. To do that, click the ellipsis button that appears when you click the (entry in the (Data Bindings) property, opening the Advanced Data Binding dialog you see in Figure 10.3.



Source: http://www.yaldex.com/vb-net-tutorial2/library.books24x7.com/book/id_5526/viewer.asp@bookid=5526&chunkid=0925001398.htm

Figure 10.3: The Advanced Data Binding Dialog

By using the Advanced Data Binding dialog, you can bind any property of a control to a data source.

Note that because simple-bound controls show only one data element at a time (such as the current author's last name), it's usual to include navigation controls in a Windows form with simple-bound controls.

You can also perform simple binding in code, using a control's DataBindings property which holds a collection of Binding objects corresponding to the bindings for the control. For example, we could bind the text box to the same au_lname field that we just bound it to at design time, in code. Using the collection's Add method, you pass this method the property to bind, the data source to use, and the specific field you want to bind:

TextBox1.DataBindings.Add ("Text", DataSet11, "authors.au_lname")

The Add method is overloaded so that you can pass it a Binding object directly, as in this example, where I'm binding a date-time picker's Value property to a field in a data table:

DateTimePicker1.DataBindings.Add _(New Binding("Value", DataSet11, "customers.DeliveryDate"))

You can even bind one control to another in code this way. Here, I'm binding the Text property of one text box to another, which means that if you change the text in the source text box, the text in the bound text box will change immediately to match:

TextBox2.DataBindings.Add ("Text", TextBox1, "Text")

10.3.2 Complex Binding

Simple data binding binds to one data item at a time, such as a name displayed in a text box, but complex data binding allows a control to bind to more than one data element, unlike simple binding which binds to one data item at a time. Binding more than one record in a database, at the same time is an example of complex binding. Most controls support only simple data binding but some controls support complex data binding like data grids and list boxes.

Complex data binding include the following properties:

- *Data Source:* It is typically a dataset such as DataSet11.
- *Data Member:* The data member is basically a table in a dataset such as the **authors** table in the pubs database. Data grids use this property to determine which table to display.
- *Display Member:* The field you want a control to display, such as the author's last name, au_lname. List boxes use the Display Member and Value Member properties instead of a Data Member property.
- *Value Member:* The field you want the control to return in properties like Selected Value, such as au_id. List boxes use the Display Member and Value Member properties instead of a Data Member property.

For setting the above four properties, you have to assign them a new value. It is easy to perform settings at run time. For example, we show the binding of dataset DataSet11, to a data grid in code, showing the authors table:

DataGrid1.DataSource = DataSet11

DataGrid1.Data Member = "authors"

The built-in data grid method named Set Data Binding can also be used for this (data grids are the only controls that have this method):

```
DataGrid1.SetDataBinding(dsDataSet, "authors")
```

Here we show how to bind the dsDataSet dataset to a list box using the Display Member property, using the au_lname field in the authors table; note the syntax used to specify a field in a table: authors.au_lname:

```
ListBox1.DataSource = dsDataSet
ListBox1.DisplayMember = "authors.au_lname"
```

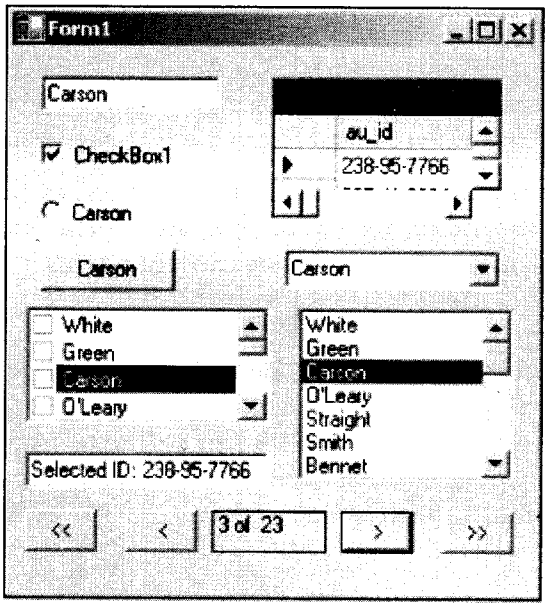
And that's all the overview we need—now it's time to get to some real code as we work with the Data Binding example.

Binding List Boxes

List boxes are complex data-binding controls. Here are the properties you use to bind this control to a data source:

- Data Source
- Display Member
- Value Member

We have shown the example in the figure shown below where we have bound a list box to the authors.au_lname field in the pubs database.



Source: http://www.yaldex.com/vb-net-tutorial2/library.books24x7.com /book/id_5526/viewer.asp@bookid=5526&chunkid=0541887519.htm

10.4 DATABASE ACCESS FROM WEB-BASED APPLICATIONS

In today's world, every web-based application needs to access some sort of database. This discussion examines the different tools available for developing web based applications and the database access options a developer has with each.

10.4.1 Different Types of Databases

There are quite a few different types of databases. Hierarchical and relational databases are probably the only ones relevant to a developer today.

Hierarchical Databases

Hierarchical databases are made like a tree of nodes. Each node must be referenced by the path to that node from some other node. Examples of hierarchical databases are LDAP databases and file systems. . Individual nodes can contain a variety of information, but the tree structure is what defines the database type. A file system is actually a database; a hierarchical one.

Relational databases are defined as a collection of tables. Each table consists of rows and columns. The intersection of a row and column is called a field. Columns have a data type such as character, numeric or boolean and usually a length. A field in one table can correspond to a field in another table, relating the rows in the two tables.

Relational Databases

There are actually different types of relational databases.

Structured Query Language databases are the most popular. The RDBMS collects the data needed in the query and returns it to the client in a result set. Clients access the data store by submitting queries to the RDBMS in an SQL database. Examples of SQL databases are the open source MySQL database and commercial products like Oracle, Sybase and Microsoft SQL Server databases.

Flat file databases are one of the relational database. Flat file databases can be opened and edited by a text editor. They are most often used to store small amounts of data. Columns can be of fixed length and white space padded. Rows are usually separated by carriage returns.

Transactional Databases

There are two varieties: transactional and non-transactional. A transactional RDBMS is used to maintain discrete sessions and provides commit and rollback methods. Changes made by one session are invisible to other sessions. They will be visible when a commit is executed. Changes can also be rolled back before they are committed. During a rollback, the state of the database prior to the changes is restored. Most expensive commercial RDBMS's are transactional while lower end commercial and free databases are not.

10.4.2 Accessing Databases

Database access in web application involves getting data from the database onto the web.

Solution for this is determined by some factors like operating system, database and development tools. The way the database information needs to be displayed is the another factor.

If the application doesn't display much information from an infrequently updated database, then the effective solution could be generating web-presentable reports. There are some commercial reporting tools such as Crystal Reports and Oracle Reports which can print straight to HTML. These tools only run on Microsoft Windows though there may be similar tools for other platforms. At a lower level, scriptable tools are there for most of the databases for executing queries. Presentations can be generated periodically with one of these tools and processed for the web with some template html files and crafty set scripting. For databases which do not include scriptable query tools, then the programs can be written in a variety of languages to complete the same end result using API calls.

For building dynamic, database-driven applications, the simplest solution is to use a web application server. Database objects can be dragged and dropped onto the web pages. Web application servers are usually constrained in OS support, functionality and scalability and in the databases that they can connect to.

Relational database objects can be dragged and dropped into web pages in InterDev, but the pages are Active Server Pages (ASP's) and the database objects are (ultimately) accessed from them through ODBC (Open Database Connectivity). Microsoft Visual InterDev is more flexible. It's not a web application server, but a web-based application development system ASP's are very extensible and ODBC can connect to a lot of different relational databases, but in the end the application will most likely have to run through Microsoft Internet Information System (IIS) on a Microsoft Operating System.

Oracle provides a unique solution. In Oracle 8i and greater the RDBMS attaches to an external web server like Apache, Netscape or Microsoft IIS and serves applications from within. Applications reside and execute inside the database as PL/SQL procedures.

Most non-drag-and-drop solutions require some programming and an API. Popular programming languages for writing web-based applications include PHP, Perl, Python, C/C++ and Java.

Database API's are almost always C libraries and may be used directly by any C or C++ program or serverlet. Alternatively, the ODBC API may be used in place of a specific database API on many platforms. xBase databases can be accessed using Sequiter CodeBASE or a number of freeware libraries. The BerkeleyDB database libraries come with Unix. Various freeware LDAP API's are available as C libraries.

For connecting to Oracle databases from C or C++ programs, Pro C is another option. Pro C is a preprocessor that translates embedded SQL into Oracle API calls.

Accessing Databases without an API

Sometimes it's necessary to write a web-based application using a language without database API. Most of the languages can either execute DLL or COM (Common Object Model) object methods or make command line calls. Shared object libraries can be written in C and command line programs can be written in some language listed above. Apart from some extra work that is involved, there is almost always a way to access a database from a language without an API.

Another limiting issue is the platform issue. Database API's only exist for a limited set of platforms. Some databases only have API's for Microsoft operating systems on Intel-based hardware. Others support both Microsoft and Unix, but only some Unixes. What about MacOS? What about PowerPC Linux? What about Microsoft NT on Alpha? If the database API doesn't exist for a particular platform

then the language-specific interface to it doesn't exist for that platform either. A program which makes non-existent interface calls won't run or compile.

RPC (Remote Procedure Call), Corba, DCOM (Distributed COM) and Java RMI (Remote Method Invocation) allow a program to invoke methods on other computers and get the results back locally. For accessing Microsoft SQL Server from Unix. SQL Server and Sybase both, use the TDS (Tabular Data Stream) protocol, so if you're using ODBC on Unix the Sybase driver may work. Distributed processing is a generic solution. These are useful if you have another computer somewhere to farm out the database work to. C and C++ support RPC and Corba on almost any platform and DCOM on some platforms (including some Unixes, but it's really expensive). Java supports RMI. Perl and Python support some of these methods outright, and may be extended with modules to support more.

There is another solution which is to write command line programs on a remote machine which execute database API calls and return the results to standard output or to a file on a shared volume.

Check Your Progress

1. What is data adapter?
2. Define the component classes of data provider.

10.5 LET US SUM UP

Data access is basically defined as the interaction of application with a database, where all the data is stored. DataSet is defined as the collection of database tables. A Data Provider is a set of related components that work together to provide data in an efficient manner. The Connection object provides the connection to the database. The Command object is includes two corresponding classes. They are SqlCommand and OleDbCommand classes. The DataReader object represents a forward-only and read-only result set. The DataAdapter is defined as the core of ADO.NET's disconnected data access. Simple binding you can show only *one* data element. Complex data binding allows a control to bind to more than one data element, unlike simple binding which binds to one data item at a time.

10.6 KEYWORDS

DataSet: It is defined as the collection of database tables.

DataProvider: It is a set of related components that work together to provide data.

DataReader object: It represents a forward-only and read-only result set.

DataAdapter: It serves as a bridge between sql server database and dataset.

10.7 QUESTIONS FOR DISCUSSION

1. Discuss the process of data access with ado.net.
2. Discuss binding controls to databases through complex binding with example.
3. How the database is accessed in web application?

Check Your Progress: Modal Answers

1. The Data Adapter object is used to fill the dataset. It is defined as the core of ADO.NET's disconnected data access. It serves as a bridge between sql server database and dataset.
2. Each Data Provider includes the component classes which are defined below.
   (i) ***The Connection object:*** It provides a connection to the database
   (ii) ***The Command object:*** It executes a command
   (iii) ***The Data Reader object:*** It provides a forward-only, read only, connected recordset
   (iv) ***The Data Adapter object:*** It provides a disconnected DataSet with data and it performs updation.

10.8 SUGGESTED READINGS

Sanjeev Sharma, *Visual Basic 6*, Excel Books

A. Chakraborti et al., *Microsoft.Net Framework*, PHI, 2002

M. Reynolds et al., *.Net Enterprise*, Wrox/SPD, 2002

Vikas Gupta, *.Net Programming*, Dreamtech Publication

Mackenzie Sharkey, *Teach yourself Visual Basic.Net in 21 days*, SAMS Techmedia

Bible, Bill Evjen, Jason Beres et al., *Visual Basic.Net Programming*, Wiley India