UNIT 5 VB SCRIPT

Structure **Page No.**

5.0 INTRODUCTION

After learning JavaScript to make WebPages that require some logical processing, we will discuss another very common scripting language, called VB Script. VBScript is Microsoft's scripting language. It enables us to write programs that enhance the power of Web pages by allowing us to control their behaviour. Although HTML enables us to develop Web pages, we cannot incorporate into them conditions or business rules without using another tool like a scripting language.

In this unit we will discuss the programming constructs that are supported by VBScript. The constructs will enable you to implement all kinds of logic for your Web pages.

We will also discuss about Objects in VBScript and talk about the Dictionary object in detail as an example. Error handling is an important part of any kind of programming. In this unit we discuss error handling in VBScript. The Err object is used for the purpose by VBScript

5.1 OBJECTIVES

The objective of this unit is to explain the basics of VBScript. After completing this unit, you will be able to write code for Web pages using VBScript. You will be able to understand the following components of VBScript:

- Variables
- Loops
- Conditional statements
- Procedures
- Properties and Methods of Dictionary Object
- Properties and Methods of the ERR Object.

5.2 WHAT IS VBSCRIPT?

VBScript is a member of Microsoft's Visual Basic family of development products. Other members include Visual Basic (Professional and Standard Editions) and Visual Basic for Applications, which is the scripting language for Microsoft Excel. VBScript is a scripting language for HTML pages on the World Wide Web and corporate intranets. VBScript is powerful and has almost all the features of Visual Basic. One of the things you should be concerned about is the safety and security of client machines that access your Web site. Microsoft took this consideration into account when creating VBScript. Potentially dangerous operations that can be done in Visual Basic have been removed from VBScript, including the capability to access dynamic link libraries directly and to access the file system on the client machine.

5.3 ADDING VBSCRIPT CODE TO AN HTML PAGE

You can use the SCRIPT element to add VBScript code to an HTML page.

The <SCRIPT> Tag
VBScript code is written within paired <SCRIPT> tags. For example, a procedure to test a delivery date might appear as follows:

```
<SCRIPT LANGUAGE="VBScript">
 <!--
  Function CanDeliver(Dt)
    CanDeliver = (CDate(Dt) - Now()) > 2
  End Function
 -->
</SCRIPT>
```

Beginning and ending <SCRIPT> tags surround the code. The LANGUAGE attribute indicates the scripting language. You must specify the language because browsers can use other scripting languages, such as JavaScript. Notice that the CanDeliver function is embedded in comment tags (<!-- and -->). This prevents browsers that do not support the <SCRIPT> tag from displaying the code on the page.

Since the example is a general function - it is not tied to any particular form control - you can include it in the HEAD section of the page as shown below:

```
<HTML>
<HEAD>
<TITLE>Place Your Order</TITLE>
<SCRIPT LANGUAGE="VBScript">
 <!--
  Function CanDeliver(Dt)
    CanDeliver = (CDate(Dt) - Now()) > 2
  End Function
 -->
</SCRIPT>
</HEAD>

<BODY>
...
```

You can use SCRIPT blocks anywhere in an HTML page. You can put them in both the BODY and HEAD sections. However, you will probably want to put all general-purpose scripting code in the HEAD section in order to keep all the code together. Keeping your code in the HEAD section ensures that all the code is read and decoded before it is needed by any calls from within the BODY section.

One notable exception to this rule is that you may want to provide inline scripting code within forms to respond to the events of objects in your form. For example, you can embed scripting code to respond to a button click in a form as shown below.

```
<HTML>
<HEAD>
<TITLE>Test Button Events</TITLE>
</HEAD>
  <BODY>
    <FORM NAME="Form1">
      <INPUT TYPE="Button" NAME="Button1"
    VALUE="Click">
      <SCRIPT FOR="Button1" EVENT="onClick"
    LANGUAGE="VBScript">
        MsgBox "Button Pressed!"
      </SCRIPT>
    </FORM>
  </BODY>
</HTML>
```

This example will display a button on the Web Page. When you click on the button it would display a message box stating "Button Pressed".

Most of your code will appear in either **Sub** or **Function** procedures and will be called only when the code you have written causes that function to execute. However, you can write VBScript code outside procedures, but still within a SCRIPT block. This code is executed only once, when the HTML page loads. This allows you to initialize data or dynamically change the look of your Web page when it loads.

5.4 VB SCRIPT BASICS

This section will discuss the basics of VBScript and describe concepts like datatypes, loops and others.

5.4.1 VBScript Data Types

VBScript has only one data type called a **Variant**. A **Variant** is a special kind of data type that can contain different kinds of information, depending on how it is used. Because **Variant** is the only data type in VBScript, it is also the data type returned by all functions in VBScript.

At its simplest, a **Variant** can contain either numeric or string information. A **Variant** behaves as a number when you use it in a numeric context and as a string when you use it in a string context. That is, if you are working with data that looks like numbers, VBScript assumes that it is a number and does whatever is most appropriate for numbers. Similarly, if you are working with data that can only be string data, VBScript treats it as string data. You can always make numbers behave as strings by enclosing them in quotation marks (" ").

Variant Subtypes

Beyond the simple numeric or string classifications, a **Variant** can make further distinctions about the specific nature of numeric information. For example, you can have numeric information that represents a date or a time. When used with other date or time data, the result is expressed as a date or a time. You can also have a rich variety of numeric information ranging from Boolean values to huge floating-point numbers. These different categories of information that can be contained in a **Variant** are called subtypes. Most of the time, you can just put the kind of data you want in a **Variant**, and the **Variant** behaves in a way that is most appropriate for the data it contains.

The following table shows the subtypes of data that a **Variant** can contain.

Subtype	Description
EmptyVariant	Uninitialized. Value is 0 for numeric variables or a zero-length string ("") for string variables.
NullVariant	Iintentionally contains no valid data.
Boolean	Contains either True or False.
Byte	Contains integer in the range 0 to 255.
Integer	Contains integer in the range -32,768 to 32,767.
Currency	-922,337,203,685,477.5808 to 922,337,203,685,477.5807.
Long	Contains integer in the range -2,147,483,648 to 2,147,483,647.
Single	Contains a single-precision, floating-point number in the range -3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.
Double	Contains a double-precision, floating-point number in the range -1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values.
Date	Contains a number that represents a date between January 1, 100 to December 31, 9999.
Time	Represents a time between 0:00:00 and 23:59:59
String	Contains a variable-length string that can be up to approximately 2 billion characters in length.
Object	Contains an object.
Error	Contains an error number.

You can use conversion functions to convert data from one subtype to another. In addition, the VarType function returns information about how your data is stored within a **Variant**.

5.4.2 VBScript Variables

A variable is a convenient placeholder that refers to a computer memory location where you can store program information that may change while your script is running. For example, you might create a variable called ClickCount to store the number of times a user clicks an object on a particular Web page. Where the variable is stored in computer memory is unimportant. What is important is that you only have to refer to a variable by name to see its value or to change its value. In VBScript, variables are always of one fundamental data type, Variant.

Declaring Variables

You declare variables explicitly in your script using the Dim statement, the Public statement, and the Private statement. For example:

```
Dim DegreesFahrenheit
```

You declare multiple variables by separating each variable name with a comma. For example:

```
Dim Top, Bottom, Left, Right
```

You can also declare a variable implicitly by simply using its name in your script. That is not generally a good practice because you could misspell the variable name in one or more places, causing unexpected results when your script is running. For that reason, the Option Explicit statement is available to require explicit declaration of all variables.

Naming Restrictions

Variable names follow the standard rules for naming anything in VBScript. A variable name:

- Must begin with an alphabetic character.
- Cannot contain an embedded period.
- Must not exceed 255 characters.
- Must be unique in the scope in which it is declared.

Scope and Lifetime of Variables

The scope of a variable is determined by where you declare it. When you declare a variable within a procedure, only code within that procedure can access or change the value of that variable. It has local scope and is called a procedure-level variable. If you declare a variable outside a procedure, you make it visible to all the procedures in your script. This is a script-level variable, and it has script-level scope.

How long a variable exists defines its lifetime. The lifetime of a script-level variable extends from the time it is declared until the time the script is finished running. At procedure level, a variable exists only as long as you are in the procedure. When the procedure exits, the variable is destroyed. Local variables are ideal as temporary storage space when a procedure is executing. You can have local variables of the same name in several different procedures because each is recognized only by the procedure in which it is declared.

Assigning Values to Variables

Values are assigned to variables creating an expression as follows: the variable is on the left side of the expression and the value you want to assign to the variable is on the right, with the '=' sign being the assignment operator. For example:

```
B = 200
```

Scalar Variables and Array Variables

Most of the time, you just want to assign a single value to a variable you have declared. A variable containing a single value is a scalar variable. At other times, it is convenient to assign more than one related value to a single variable. Then you can create a variable that can contain a series of values. This is called an array variable. Array variables and scalar variables are declared in the same way, except that the declaration of an array variable uses parentheses ( ) following the variable name. In the following example, a single-dimension array containing 11 elements is declared:

```
Dim A(10)
```

Although the number shown in the parentheses is 10, all arrays in VBScript are counted from base 0, so that this array actually contains 11 elements. In such an array, the number of array elements is always the number shown in parentheses plus one. This kind of array is called a fixed-size array.

You assign data to each of the elements of the array using an index into the array. Beginning at zero and ending at 10, data can be assigned to the elements of an array as follows:

```
A(0) = 256
A(1) = 324
A(2) = 100
. . .
A(10) = 55
```

Similarly, the data can be retrieved from any element using an index into the particular array element you want. For example:

```
. . .
SomeVariable = A(8)
. . .
```

Arrays are not limited to a single dimension. You can have as many as 60 dimensions, although most people cannot comprehend more than three or four dimensions. Multiple dimensions are declared by separating an array's size numbers in the parentheses with commas. In the following example, the MyTable variable is a two-dimensional array consisting of 6 rows and 11 columns:

```
Dim MyTable(5, 10)
```

In a two-dimensional array, the first number is always the number of rows; the second number is the number of columns.

You can also declare an array whose size changes while your script is running. This is called a dynamic array. The array is initially declared within a procedure using either the **Dim** statement or using the ReDim statement. However, for a dynamic array, no size or number of dimensions is placed inside the parentheses.

For example:

```
Dim MyArray()
ReDim AnotherArray()
```

To use a dynamic array, you must subsequently use **ReDim** to determine the number of dimensions and the size of each dimension. In the following example, **ReDim** sets the initial size of the dynamic array to 25. A subsequent **ReDim** statement resizes the array to 30, but uses the **Preserve** keyword to preserve the contents of the array as the resizing takes place.

```
ReDim MyArray(25)
. . .
ReDim Preserve MyArray(30)
```

There is no limit to the number of times you can resize a dynamic array, but you should know that if you make an array smaller than it was, you lose the data in the eliminated elements.

5.4.3 VBScript Constants

A constant is a meaningful name that takes the place of a number or string and never changes. VBScript defines a number of intrinsic constants. You can get detailed information about these intrinsic constants from the VBScript Language Reference.

Creating Constants

You create user-defined constants in VBScript using the Const statement. This lets you create string or numeric constants with meaningful names and allows you to assign them literal values. For example:

```
Const MyString = "This is my string."
Const MyAge = 49
```

Note that the string literal is enclosed in quotation marks (" "). Quotation marks are the most obvious way to differentiate string values from numeric values. Date literals and time literals are represented by enclosing them in number signs (#). For example:

```
Const CutoffDate = #6-1-97#
```

You may want to adopt a naming scheme to differentiate constants from variables. This will save you from trying to reassign constant values while your script is running. For example, you might want to use a "vb" or "con" prefix on your constant names, or you might name your constants in all capital letters. Differentiating constants from variables eliminates confusion as you develop more complex scripts.

5.4.4 VBScript Operators

VBScript has a full range of operators, including arithmetic operators, comparison operators, concatenation operators, and logical operators.

Operator Precedence

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence. You can use parentheses to override the order of precedence and force some parts of an expression to be evaluated before others. Operations within parentheses are always performed before those outside. Within parentheses, however, standard operator precedence is maintained.

When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Comparison operators all have equal precedence; that is, they are evaluated in the left-to-right order in which they appear. Arithmetic and logical operators are evaluated in the following order of precedence.

1. Arithmetic operators		2. Comparison operators		3. Logical operators	
Description	**Symbol**	**Description**	**Symbol**	**Description**	**Symbol**
Exponentiation	^	Equality	=	Logical negation	Not
Unary negation	-	Inequality	<>	Logical conjunction	And
Multiplication	*	Less than	<	Logical disjunction	Or
Division	/	Greater than	>	Logical exclusion	Xor
Integer division	\	Less than or equal to	<=	Logical equivalence	Eqv
Modulus arithmetic	Mod	Greater than or equal to	>=	Logical implication	Imp
Addition	+	Object equivalence	Is		
Subtraction	-				
String concatenation	&				

The associativity of the operators is left to right. When multiplication and division occur together in an expression, each operation is evaluated as it occurs from left to right. Likewise, when addition and subtraction occur together in an expression, each operation is evaluated in order of appearance from left to right.

The string concatenation (&) operator is not an arithmetic operator, but in precedence it falls after all arithmetic operators and before all comparison operators. The **Is** operator is an object reference comparison operator. It does not compare objects or their values; it checks only to determine if two object references refer to the same object.

5.5 USING CONDITIONAL STATEMENTS

You can control the flow of your script with conditional statements and looping statements. Using conditional statements, you can write VBScript code that makes decisions and repeats actions. The following conditional statements are available in VBScript:

If...Then...Else statement
Select Case statement

Making Decisions Using If...Then...Else

The **If...Then...Else** statement is used to evaluate whether a condition is **True** or **False** and, depending on the result, to specify one or more statements to execute. Usually the condition is an expression that uses a comparison operator to compare one value or variable with another. For information about comparison operators, see Comparison Operators. **If...Then...Else** statements can be nested to as many levels as you need.

Running Statements if a Condition is True

To run only one statement when a condition is **True**, use the single-line syntax for the **If...Then...Else** statement. The following example shows the single-line syntax. Notice that this example omits the **Else** keyword.

```
Sub FixDate()
  Dim myDate
  myDate = #2/13/95#
  If myDate < Now Then myDate = Now
End Sub
```

To run more than one line of code, you must use the multiple-line (or block) syntax. This syntax includes the **End If** statement, as shown in the following example:

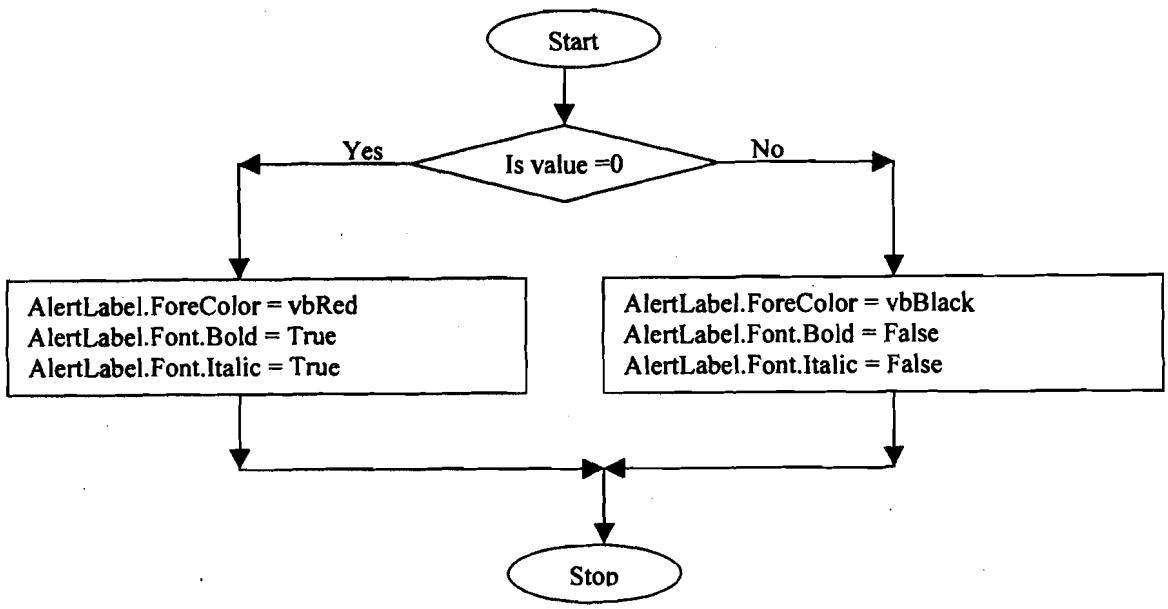
```
Sub AlertUser(value)
    If value = 0 Then
      AlertLabel.ForeColor = vbRed
      AlertLabel.Font.Bold = True
      AlertLabel.Font.Italic = True
    End If
End Sub
```

Running Certain Statements if a Condition is True and Running Others if a Condition is False

You can use an **If...Then...Else** statement to define two blocks of executable statements: one block to run if the condition is **True**, the other block to run if the condition is **False**.

```
Sub AlertUser(value)
  If value = 0 Then
    AlertLabel.ForeColor = vbRed
    AlertLabel.Font.Bold = True
    AlertLabel.Font.Italic = True
  Else
    AlertLabel.Forecolor = vbBlack
    AlertLabel.Font.Bold = False
    AlertLabel.Font.Italic = False
  End If
End Sub
```

The following flow chart explains the flow of the above example.



Deciding Between Several Alternatives

A variation on the **If...Then...Else** statement allows you to choose from several alternatives. Adding **ElseIf** clauses expands the functionality of the **If...Then...Else** statement so that you can control program flow based on different possibilities. For example:

```
Sub ReportValue(value)
  If value = 0 Then
    MsgBox value
  ElseIf value = 1 Then
    MsgBox value
```

```
ElseIf value = 2 then
  Msgbox value
Else
  Msgbox "Value out of range!"
End If
```

You can add as many **ElseIf** clauses as you need to provide alternative choices. Extensive use of the **ElseIf** clauses often becomes cumbersome. A better way to choose between several alternatives is the **Select Case** statement.

Making Decisions with Select Case

The **Select Case** structure provides an alternative to **If...Then...ElseIf** for selectively executing one block of statements from among multiple blocks of statements. A **Select Case** statement provides capability similar to the **If...Then...Else statement**, but it makes code more efficient and readable.

A **Select Case** structure works with a single test expression that is evaluated once, at the top of the structure. The result of the expression is then compared with the values for each **Case** in the structure. If there is a match, the block of statements associated with that **Case** is executed:

```
Select Case Document.Form1.CardType.Options(SelectedIndex).Text
  Case "MasterCard"
    DisplayMCLogo
    ValidateMCAccount
  Case "Visa"
    DisplayVisaLogo
    ValidateVisaAccount
  Case "American Express"
    DisplayAMEXCOLogo
    ValidateAMEXCOAccount
  Case Else
    DisplayUnknownImage
    PromptAgain
End Select
```

Notice that the **Select Case** structure evaluates an expression once at the top of the structure. In contrast, the **If...Then...ElseIf** structure can evaluate a different expression for each **ElseIf** statement. You can replace an **If...Then...ElseIf** structure with a **Select Case** structure only if each **ElseIf** statement evaluates the same expression.

5.6 LOOPING THROUGH CODE

Using Loops to Repeat Code

Looping allows you to run a group of statements repeatedly. Some loops repeat statements until a condition is **False**; others repeat statements until a condition is **True**. There are also loops that repeat statements a specific number of times. The following looping statements are available in VBScript:

Do...Loop: Loops while or until a condition is **True**.
While...Wend: Loops while a condition is **True**.
For...Next: Uses a counter to run statements a specified number of times.

For Each...Next: Repeats a group of statements for each item in a collection or each element of an array.

Using Do Loops

You can use **Do...Loop** statements to run a block of statements an indefinite number of times. The statements are repeated either while a condition is **True** or until a condition becomes **True**.

Repeating Statements While a Condition is True

Use the **While** keyword to check a condition in a **Do...Loop** statement. You can check the condition before you enter the loop (as shown in the following ChkFirstWhile example), or you can check it after the loop has run at least once (as shown in the ChkLastWhile example). In the ChkFirstWhile procedure, if myNum is set to 9 instead of 20, the statements inside the loop will never run. In the ChkLastWhile procedure, the statements inside the loop run only once because the condition is already **False**.

```
Sub ChkFirstWhile()
   Dim counter, myNum
   counter = 0
   myNum = 20
   Do While myNum > 10
      myNum = myNum - 1
      counter = counter + 1
   Loop
   MsgBox "The loop made " & counter & " repetitions."
End Sub

Sub ChkLastWhile()
   Dim counter, myNum
   counter = 0
   myNum = 9
   Do
      myNum = myNum - 1
      counter = counter + 1
   Loop While myNum > 10
   MsgBox "The loop made " & counter & " repetitions."
End Sub
```

The above figure shows two example code fragments illustrating the do... loop and do while ... loop statements. In the first loop the condition is checked before executing the code but in the second loop the statements are first executed once and then the condition is checked. In the first loop the instructions given in the loop's body will never be executed if the condition fails initially but in the second loop these instructions would be executed at least once. The following flow chart shows the execution plan of the second procedure i.e. CheckLastWhile:

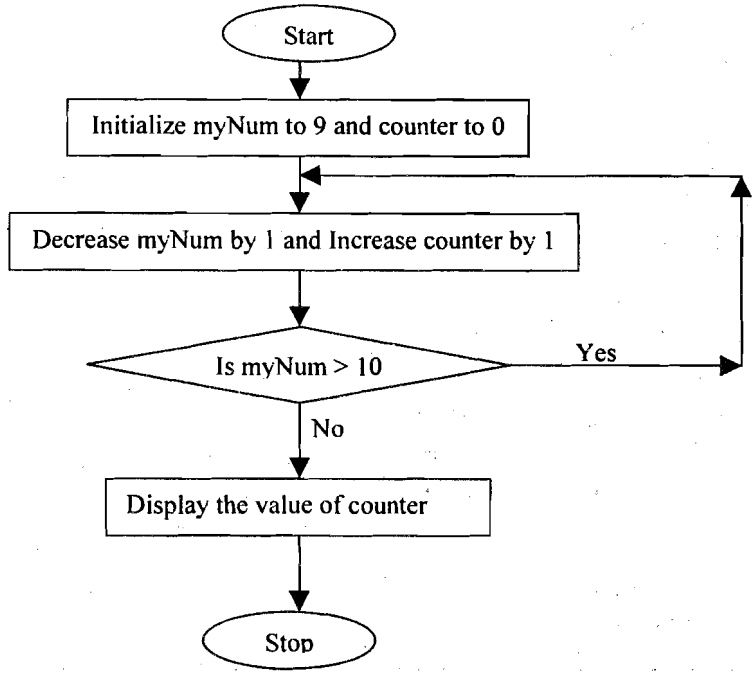


Figure 5.1: Execution Plan of the Second Loop in Figure 5.9

Repeating a Statement Until a Condition Becomes True

You can use the **Until** keyword in two ways to check a condition in a **Do...Loop** statement. You can check the condition before you enter the loop (as shown in the following ChkFirstUntil example), or you can check it after the loop has run at least once (as shown in the ChkLastUntil example). As long as the condition is **False**, the loop continues.

```
Sub ChkFirstUntil()
   Dim counter, myNum
   counter = 0
   myNum = 20
   Do Until myNum = 10
      myNum = myNum - 1
      counter = counter + 1
   Loop
   MsgBox "The loop made " & counter & " repetitions."
End Sub

Sub ChkLastUntil()
   Dim counter, myNum
   counter = 0
   myNum = 1
   Do
      myNum = myNum + 1
      counter = counter + 1
   Loop Until myNum = 10
   MsgBox "The loop made " & counter & " repetitions."
End Sub
```

Exiting a Do...Loop Statement from Inside the Loop

You can exit a **Do...Loop** by using the **Exit Do** statement. Because you usually want to exit only in certain situations, such as to avoid an endless loop, you

should use the **Exit Do** statement in the **True** statement block of an **If...Then...Else** statement. If the condition is **False**, the loop runs as usual. Otherwise, you probably did not need a loop in the first place.

In the following example, myNum is assigned a value that creates an endless loop. The **If...Then...Else** statement checks for this condition, preventing the endless repetition.

```
Sub ExitExample()
   Dim counter, myNum
   counter = 0
   myNum = 9
   Do Until myNum = 10
      myNum = myNum - 1
      counter = counter + 1
      If myNum < 10 Then Exit Do
   Loop
   MsgBox "The loop made " & counter & " repetitions."
End Sub
```

Using While...Wend

The **While...Wend** statement is provided in VBScript for those who are familiar with its usage. However, because of the lack of flexibility in **While...Wend**, it is recommended that you use **Do...Loop** instead.

Using For...Next

You can use **For...Next** statements to run a block of statements a specific number of times. For loops, use a counter variable whose value is increased or decreased with each repetition of the loop.

For example, the following procedure causes a procedure called MyProc to execute 50 times. The **For** statement specifies the counter variable x and its start and end values. The **Next** statement increments the counter variable by 1.

```
Sub DoMyProc50Times()
   Dim x
   For x = 1 To 50
      MyProc
   Next
End Sub
```

Using the **Step** keyword, you can increase or decrease the counter variable by the value you specify. In the following example, the counter variable j is incremented by 2 each time the loop repeats. When the loop is finished, total is the sum of 2, 4, 6, 8, and 10.

```
Sub TwosTotal()
   Dim j, total
   For j = 2 To 10 Step 2
      total = total + j
   Next
   MsgBox "The total is " & total
```

```
End Sub
```

To decrease the counter variable, you use a negative **Step** value. You must specify an end value that is less than the start value. In the following example, the counter variable myNum is decreased by 2 each time the loop repeats. When the loop is finished, total is the sum of 16, 14, 12, 10, 8, 6, 4, and 2.

```
Sub NewTotal()
  Dim myNum, total
  For myNum = 16 To 2 Step -2
    total = total + myNum
  Next
  MsgBox "The total is " & total
End Sub
```

You can exit any **For...Next** statement before the counter reaches its end value by using the **Exit For** statement. Because you usually want to exit only in certain situations, such as when an error occurs, you should use the **Exit For** statement in the **True** statement block of an **If...Then...Else** statement. If the condition is **False**, the loop runs as usual. Otherwise you probably did not need to use the loop in the first place.

Using For Each...Next

A **For Each...Next** loop is similar to a **For...Next** loop. Instead of repeating the statements a specified number of times, a **For Each...Next** loop repeats a group of statements for each item in a collection of objects or for each element of an array. This is especially helpful if you do not know how many elements are present in a collection.

In the following HTML code example, the contents of a **Dictionary** object are used to place text in several text boxes:

```
<HTML>
<HEAD><TITLE>Forms and Elements</TITLE></HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Sub cmdChange_OnClick
  Dim d                'Create a variable
  Set d = CreateObject("Scripting.Dictionary")
  d.Add "0", "Athens"    'Add some keys and items
  d.Add "1", "Belgrade"
  d.Add "2", "Cairo"

 a = d.items
  For i = 0 To d.Count
    document.frmForm.Elements(i).Value = a(i)
  Next
End Sub
-->
</SCRIPT>
<BODY>
```

```
<CENTER>
<FORM NAME="frmForm"> <p>
   <Input Type = "Text"><p>
   <Input Type = "Text"><p>
   <Input Type = "Text"><p>
   <Input Type = "Button" NAME="cmdChange" VALUE="Click
Here"><p>
</FORM>
</CENTER>
</BODY>
</HTML>
```

Check Your Progress 1

Write code to check whether a number is even and prime or odd and prime.

Write code in VBScript to generate the Fibonnacci series. The series 1, 1, 2, 3, 5, ... and so on. From the third number onward, the nth number is the sum of the (n - 1)th and the (n - 2)th numbers.

Write code to find the factorial of any given number. This is the product of all numbers from 2 to that number.

5.7 VBSCRIPT PROCEDURES

In VBScript there are two kinds of procedures; the Sub procedure and the Function procedure.

Sub Procedures

A **Sub** procedure is a series of VBScript statements, enclosed by **Sub** and **End Sub** statements, that perform actions but don't return a value. A **Sub** procedure can take arguments (constants, variables, or expressions that are passed by a calling procedure). If a **Sub** procedure has no arguments, its **Sub** statement must include an empty set of parentheses ().

The following code shows a **Sub** procedure uses two intrinsic, or built-in, VBScript functions, MsgBox and InputBox, to prompt a user for some information. It then displays the results of a calculation based on that information. The calculation is performed in a **Function** procedure created using VBScript. The **Function** procedure is shown after the following discussion.

```
Sub ConvertTemp()
  temp = InputBox("Please enter the temperature in degrees F.", 1)
  MsgBox "The temperature is " & Celsius(temp) & " degrees C."
End Sub
```

Function Procedures

A **Function** procedure is a series of VBScript statements enclosed by the **Function** and **End Function** statements. A **Function** procedure is similar to a **Sub** procedure, but can also return a value. A **Function** procedure can take arguments (constants, variables, or expressions that are passed to it by a calling procedure). If a **Function** procedure has no arguments, its **Function** statement must include an empty set of parentheses (). A **Function** returns a value by

assigning a value to its name in one or more statements of the procedure. The return type of a **Function** is always a **Variant**.

In the following example, the Celsius function calculates degrees Celsius from degrees Fahrenheit. When the function is called from the ConvertTemp **Sub** procedure, a variable containing the argument value is passed to the function. The result of the calculation is returned to the calling procedure and displayed in a message box.

```
Sub ConvertTemp()
   temp = InputBox("Please enter the temperature in degrees F.", 1)
   MsgBox "The temperature is " & Celsius(temp) & " degrees C."
End Sub

Function Celsius(fDegrees)
   Celsius = (fDegrees - 32) * 5 / 9
End Function
```

Getting Data into and out of Procedures

Each piece of data is passed into your procedures using an argument. Arguments serve as placeholders for the data you want to pass into your procedure. You can name your arguments with any valid variable name. When you create a procedure using either the **Sub** statement or the **Function** statement, parentheses must be included after the name of the procedure. Any arguments are placed inside these parentheses, separated by commas. For example, in the following example, fDegrees is a placeholder for the value being passed into the Celsius function for conversion:

```
Function Celsius(fDegrees)
   Celsius = (fDegrees - 32) * 5 / 9
End Function
```

To get data out of a procedure, you must use a **Function**. Remember, a **Function** procedure can return a value; a **Sub** procedure cannot.

Using Sub and Function Procedures in Code

A **Function** in your code must always be used on the right side of a variable assignment or in an expression. For example:

```
Temp = Celsius(fDegrees)
```

or

```
MsgBox "The Celsius temperature is " & Celsius(fDegrees) & " degrees."
```

To call a **Sub** procedure from another procedure, you can just type the name of the procedure along with values for any required arguments, each separated by a comma. The Call statement is not required, but if you do use it, you must enclose any arguments in parentheses.

The following example shows two calls to the MyProc procedure. One uses the **Call** statement in the code; the other does not. Both do exactly the same thing.

```
Call MyProc(firstarg, secondarg)
MyProc firstarg, secondarg
```

Notice that the parentheses are omitted in the call when the **Call** statement isn't used.

5.8 VB SCRIPT CODING CONVENTIONS

Coding conventions are suggestions that may help you write code using Microsoft Visual Basic Scripting Edition. Coding conventions can include the following:

Naming conventions for objects, variables, and procedures
Commenting conventions
Text formatting and indenting guidelines

The main reason for using a consistent set of coding conventions is to standardize the structure and coding style of a script or set of scripts so that you and others can easily read and understand the code. Using good coding conventions results in precise, readable, and unambiguous source code that is consistent with other language conventions and is as intuitive as possible.

Constant Naming Conventions

Earlier versions of VBScript had no mechanism for creating user-defined constants. Constants, if used, were implemented as variables and distinguished from other variables using all uppercase characters. Multiple words were separated using the underscore (_) character. For example:

USER_LIST_MAX
NEW_LINE

While this is still an acceptable way to identify your constants, you may want to use an alternative naming scheme, now that you can create true constants using the Const statement. This convention uses a mixed-case format in which constant names have a "con" prefix. For example:

ConYourOwnConstant

Variable Naming Conventions

For purposes of readability and consistency, use the following prefixes with descriptive names for variables in your VBScript code.

Sub type	Prefix	Example
Boolean	Bln	BlnFound
Byte	Byt	BytQuantity
Date	Dtm	DtmStart
Double	Dbl	DblPrice
Error	Err	ErrOrderNum
Integer	Int	IntPrice
Long	Lng	LngDistance
Object	Obj	ObjCurrent
Single	Sng	SngAverage
String	Str	StrName

Variable Scope

Variables should always be defined with the smallest scope possible. VBScript variables can have the following scope.

Scope	Where Variable is Declared	Visibility
Procedure-level	Event, Function, or Sub procedure	Visible in the procedure in which it is declared
Script-level	HEAD section of an HTML page, outside any procedure	Visible in every procedure in the script

Variable Scope Prefixes

As script size grows, so does the value of being able to quickly differentiate the scope of variables. A one-letter scope prefix preceding the type prefix provides this, without unduly increasing the size of variable names.

Scope	Prefix	Example
Procedure-level	None	DblPrice
Script-level	S	SDblPrice

Descriptive Variable and Procedure Names

The body of a variable or procedure name should use mixed case and should be as complete as necessary to describe its purpose. In addition, procedure names should begin with a verb, such as InitNameArray or CloseDialog.

For frequently used or long terms, standard abbreviations are recommended to help keep name length reasonable. In general, variable names greater than 32 characters can be difficult to read. When using abbreviations, make sure they are consistent throughout the entire script. For example, randomly switching between Cnt and Count within a script or set of scripts may lead to confusion.

Object Naming Conventions

The following table lists recommended conventions for objects you may encounter while programming VBScript.

Code Commenting Conventions

All procedures should begin with a brief comment describing what they do. This description should not describe the implementation details (how it does it) because these often change over time, resulting in unnecessary comment maintenance work, or worse, erroneous comments. The code itself and any necessary inline comments describe the implementation.

Arguments passed to a procedure should be described when their purpose is not obvious and when the procedure expects the arguments to be in a specific range. Return values for functions and variables that are changed by a procedure, especially through reference arguments, should also be described at the beginning of each procedure.

Procedure header comments should include the following section headings. For examples, see the following table that explains the section heading and the comment contents to be inserted while developing code.

Section Heading	Comment Contents
Purpose	What the procedure does (not how).
Assumptions	List of any external variable, control, or other element whose state affects this procedure.
Effects	List of the procedure's effect on each external variable, control, or other element.
Inputs	Explanation of each argument that is not obvious. Each argument should be on a separate line with inline comments.
Return Values	Explanation of the value returned.

Remember the Following Points

Every important variable declaration should include an inline comment describing the use of the variable being declared.
Variables, controls, and procedures should be named clearly enough that inline comments are only needed for complex implementation details.

At the beginning of your script, you should include an overview that describes the script, enumerating objects, procedures, algorithms, dialog boxes, and other system dependencies. Sometimes a piece of pseudocode describing the algorithm can be helpful.

Formatting Your Code

Screen space should be conserved as much as possible, while still allowing code formatting to reflect logic structure and nesting. Here are a few pointers:

- Standard nested blocks should be indented four spaces.
- The overview comments of a procedure should be indented one space.
- The highest level statements that follow the overview comments should be indented four spaces, with each nested block indented an additional four spaces. For example, see the code below:

```
'*************************************************************
' Purpose:  Locates the first occurrence of a specified user
'         in the UserList array.
' Inputs:   strUserList():  the list of users to be searched.
'         strTargetUser:  the name of the user to search for.
' Returns:  The index of the first occurrence of the strTargetUser
'         in the strUserList array.
'         If the target user is not found, return -1.
'*************************************************************

Function intFindUser (strUserList(), strTargetUser)
  Dim i                ' Loop counter.
  Dim blnFound                  ' Target found flag
  intFindUser = -1
  i = 0                ' Initialize loop counter
  Do While i <= Ubound(strUserList) and Not blnFound
If strUserList(i) = strTargetUser Then
        blnFound = True   ' Set flag to True
        intFindUser = i   ' Set return value to loop count
    End If
    i = i + 1           ' Increment loop counter
  Loop
End Function
```

Check Your Progress 2

1. Write a menu - driven program in VBScript to check whether a number is even, odd and whether it is prime.

2. Write a procedure to check whether a string is a palindrome.

5.9 DICTIONARY OBJECT IN VBSCRIPT

The Dictionary object stores data as key, item pairs. A Dictionary object is the equivalent of a PERL associative array. Items, which can be any form of data, are stored in the array. Each item is associated with a unique key. The key is used to retrieve an individual item and is usually an integer or a string, but can be anything except an array.
The following code illustrates how to create and use a Dictionary object:

```
<HTML>
<HEAD><TITLE>Forms and Elements</TITLE></HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
  Dim d              'Create a Script Level variable
  Set d = CreateObject("Scripting.Dictionary")

Sub cmdAdd_OnClick
  d.Add "0", "Amar"    'Add some keys and items
  d.Add "1", "Bunty"
  d.Add "2", "Chaman"

  a = d.items
  For i = 0 To d.Count
    document.frmForm.Elements(i).Value = a(i)
  Next
End Sub

Sub cmdExist_onClick
  d.Add "a", "Amar"    'Add some  keys and items.
  d.Add "b", "Bunty"
  d.Add "c", "Chaman"
  If d.Exists("c") Then
       msgbox "C key exists."
  Else
    msgbox "C key does not exist."
  End If
End Sub

Sub cmdKey_onClick
  d.Add "a", "Amar"    'Add some keys and items.
  d.Add "b", "Bunty"
  d.Add "c", "Chaman"
  d.Key("c") = "d"       'Set key for "c" to "d".
End Function


End Sub
Sub cmdKeys_onClick
  d.Add "a", "Amar"    'Add some  keys and items.
  d.Add "b", "Bunty"
  d.Add "c", "Chaman"
  a = d.keys
  For i = 0 To d.Count
    document.frmForm.Elements(i).Value = a(i)
  Next
End Sub

Sub cmdRemove_onClick
  d.Add "a", "Amar"    'Add some  keys and items.
  d.Add "b", "Bunty"
  d.Add "c", "Chaman"

  d.Remove("b")
  a = d.keys
  For i = 0 To d.Count
    document.frmForm.Elements(i).Value = a(i)
  Next
```

```
End Sub

Sub cmdCompareMode_onClick
  d.CompareMode = vbTextCompare
  d.Add "a", "Amar"     'Add some keys and items.
  d.Add "b", "Bunty"
  d.Add "c", "Chaman"
  d.Add "B", "Baltimore"   'Add method fails on this line because the
                           'letter b already exists in the Dictionary.
End Sub
-->

</SCRIPT>
<BODY>
<CENTER>
<FORM NAME="frmForm"><p>
        <Input Type = "Text"><p>
        <Input Type = "Text"><p>
        <Input Type = "Text"><p>
        <Input Type = "Button" NAME="cmdAdd" VALUE="Add">
        <Input Type = "Button" NAME="cmdExist" VALUE="Exist">
        <Input Type = "Button" NAME="cmdKeys" VALUE="Keys">
        <Input Type = "Button" NAME="cmdRemove" VALUE="Remove"><p>

        <Input Type = "Button" NAME="cmdCompareMode" VALUE="Compare
Mode">
        <Input Type = "Button" NAME="cmdKey" VALUE="Key">

</FORM>
</CENTER>
</BODY>
</HTML>
```

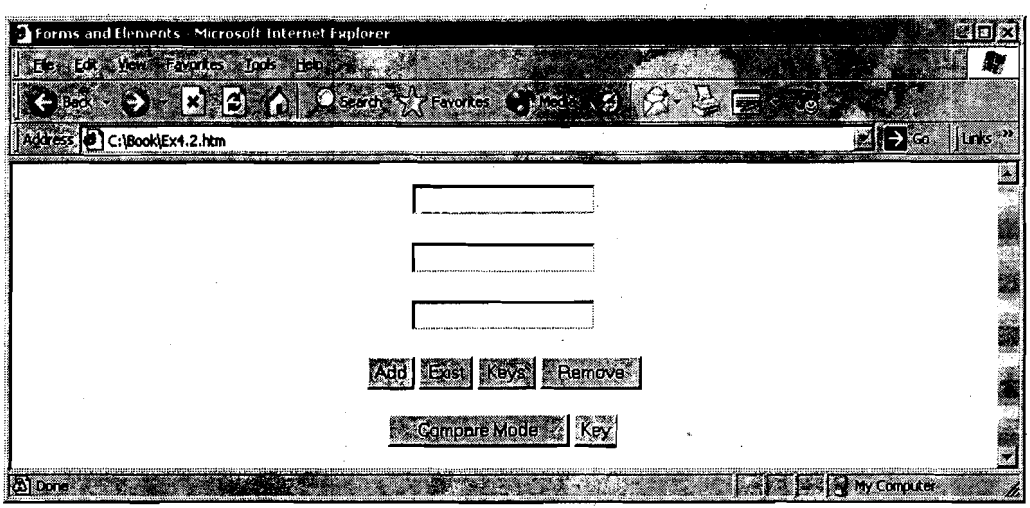


Figure 5.1: Code Using the Dictionary Object

5.9.1 Methods: VBScript Dictionary Object

Method	Description
Add Method	Adds a key, item pair.
Exists Method	Indicates if a specified key exists.
Items Method	Returns an array containing all items in a Dictionary object.
Keys Method	Returns an array containing all keys in a Dictionary object.
Remove Method	Removes a key, item pair.Properties: VBScript Dictionary Object

Add Method

Syntax	object.Add key, item
Object	The name of a Dictionary object. Required
Key	The key associated with the item being added. Required.
Item	The item associated with the key being added. Required.
Remarks	An error occurs if the key already exists.

Refer to Figure 5.20 for the example. In the cmdAdd_onClick procedure this method is used to add the Keys and Items to the Dictionary object.

Exists Method

Returns True if a specified key exists in the Dictionary object, False if it does not.

Syntax	object.Exists(key)
Object	The name of a Dictionary object. Required.
Key	The key value being searched for in the Dictionary object. Required.

Refer to Figure 5.20 for the example. In the cmdExists_onClick procedure this method is used to check the existence of any Key value in the Dictionary object.

Items Method

Returns an array containing all the items in a Dictionary object.

Syntax	object.Items
Object	The name of a Dictionary object. Required.

Refer to Figure 5.20 for the example code. In the cmdAdd_onClick procedure this method is used to retrieve the items in the array variable.

Keys Method

Returns an array containing all existing keys in a Dictionary object.

Syntax	object.Keys
Object	The name of a Dictionary object. Required.

Refer to Figure 5.20 for example code. In the cmdKeys_onClick procedure this method is used to retrieve the Keys in the array variable.

Remove Method

Removes a key, item pair from a Dictionary object. An error occurs if the specified key, item pair does not exist.

Syntax	object.Remove(key)
Object	The name of a Dictionary object. Required
Key	The Key associated with the key-item pair you want to remove from the Dictionary object. Required

Refer to Figure 5.20 for example code. In the cmdRemove_onClick procedure this method is used to remove the item from the dictionary object.

5.9.2 VBScript Dictionary Object Properties

Just like normal objects the Dictionary object also has certain properties. These properties can be set to any valid value and can be retrieved as and when required.

Property	Description
CompareMode Property	The comparison mode for string keys.
Count Property	The number of items in a Dictionary object.
Item Property	An item for a key.
Key Property	A key Syntax: VBScript Dictionary Object Scripting.Dictionary

CompareMode Property

Sets and returns the comparison mode for comparing string keys in a Dictionary object.

Syntax: object.CompareMode[= *compare*]

The CompareMode property has the following parts:

Part	Description
Object	Required. Always the name of a Dictionary object.
Compare	Optional. If provided, *compare* is a value representing the comparison mode used by functions such as StrComp.

The *compare* argument has the following settings:

Constant	Value	Description
VbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.

Remarks:

Values greater than 2 can be used to refer to comparisons using specific Locale IDs (LCID). An error occurs if you try to change the comparison mode of a Dictionary object that already contains data.

The CompareMode property uses the same values as the *compare* argument for the StrComp function.

Refer to Figure 5.20 for example code. In the cmdCompareMode_onClick procedure this property is used to set the compare mode to Text, so that no two text keys that are the same can be added.

Count Property

Returns the number of items in a collection or Dictionary object. This property can only be read and cannot be set directly.

Syntax	object.Count
Object	The name of a Dictionary object. Required.

Refer to Figure 5.20 for example code. In the cmdAdd_onClick procedure this property is used to retrieve the number of items stored in the Dictionary object.

Item Property

Sets or returns an *item* for a specified *key* in a Dictionary object. For collections, returns an *item* based on the specified *key*. This property can be retrieved or set.

Syntax: object.Item(key) [= newitem]

The Item property has the following parts:

Part	Description
object	Required. Always the name of a collection or Dictionary object.
key	Required. *Key* associated with the *item* being retrieved or added.
newitem	Optional. Used for Dictionary object only; no application for collections. If provided, *newitem* is the new value associated with the specified *key*.

Remarks:

If *key* is not found when changing an *item*, a new *key* is created with the specified *newitem*. If *key* is not found when attempting to return an existing item, a new *key* is created and its corresponding item is left empty.

Key Property

Sets a key in a Dictionary object. If key is not found when changing a key, an error will occur.

Syntax: *object*.Key(*key*) = *newkey*

The **Key** property has the following parts:

Part	Description
object	Required. Always the name of a **Dictionary** object.
Key	Required. *Key* value being changed.
newkey	Required. New value that replaces the specified *key*.

Refer to Figure 5.20 for example code. In the cmdKey_onClick procedure this property is used to change the key value from "C" to "D".

5.10 ERR OBJECT

The VBScript Err Object contains information about run-time errors. The properties of the Err object are set by the generator of an error - Visual Basic, an Automation object, or the VBScript programmer.

The default property of the Err object is Number. Err.Number contains an integer and can be used by an Automation object to return an SCODE.

When a run-time error occurs, the properties of the Err object are filled with information that uniquely identifies the error and information that can be used to handle it. To generate a run-time error in your code, use the VBScript Err Object Raise Method. The Err object's properties are reset to zero or zero-length strings ("") after an On Error Resume Next statement. The VBScript Err Object Clear Method can be used to explicitly reset Err object. The Err object is an intrinsic object with global scope - there is no need to create an instance of it in your code.

Consider the following example, which displays some of the possible errors that can be raised explicitly to generate run time errors.

```
<HTML>
<HEAD>

<TITLE>IGNOU</TITLE>

<SCRIPT LANGUAGE="VBScript">

<!--

Sub cmdSubmit_OnClick
```

```
On Error resume Next
' Check to see if the user entered anything.
 If (Len(document.form1.txtAge.value) = 0) Then
  MsgBox "You must enter your age before submitting."
  Exit Sub
 End If
' Check to see if the user entered a number.
 If (Not(IsNumeric(document.form1.txtAge.value ))) Then
  MsgBox "You must enter a number for your age."
  Exit Sub
 End If
' Check to see if the age entered is valid.
 If (document.form1.txtAge.value <= 0) Or  (document.form1.txtAge.value > 100)
Then
  MsgBox "The age you entered is invalid."
  Exit Sub
 End If
' Data looks okay so submit it.
 MsgBox "Thanks for providing your age."
 document.form1.submit
End Sub
Sub cmdtype_OnClick
On Error Resume Next
        for i = 1 to 35
        Err.clear
        Err.Raise i
          MsgBox Err.description & " - Error number - " &i
        Next
End Sub
-->
</SCRIPT>
</HEAD>
<BODY  bgColor="#ffffcc" Text="#000099" >
<H1>Error Handling and validations</H1>
<B><P> This example demonstrates validation techniques and Error handling in
VBScript. </P><?B>
<B>Please enter the age between 1 and 100. Otherwise, an error message would be
```

```
flashed on clicking the submit button.</B>
<FORM NAME="form1">
<TABLE>
<TR>
<TD>Enter your age:</TD>
<TD><INPUT TYPE="Text" NAME="txtAge" SIZE="2"></TD>
</TR>
<TR>
<TD><INPUT TYPE="Button" NAME="cmdSubmit" VALUE="Submit"></TD>
<TD></TD>
<TD><INPUT TYPE="Button" NAME="cmdtype" VALUE="     Type Of Errors
"></TD>
    <TD></TD>
</TR>
 </TABLE>
</FORM>
</BODY>
</HTML>
```

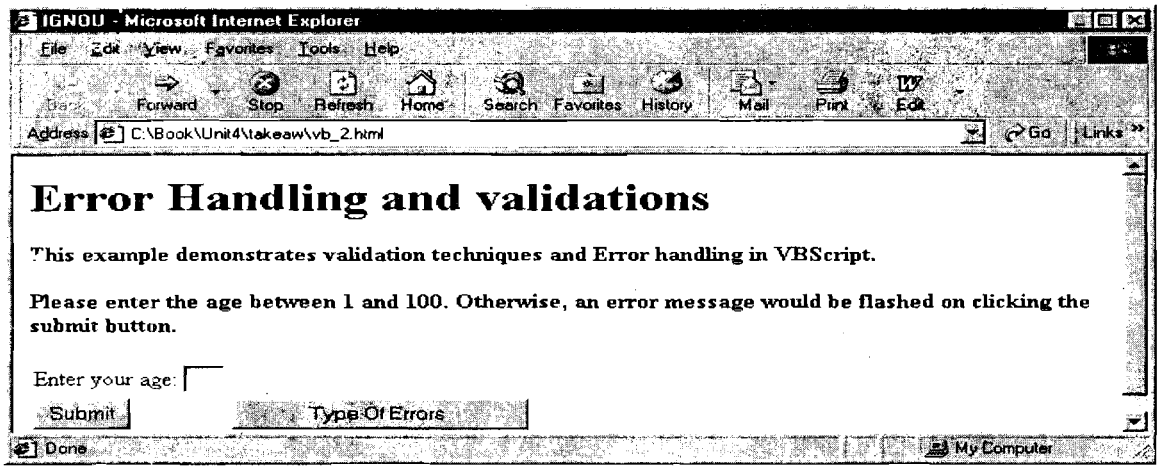


Figure 5.2: Error Handling in VBScript

5.10.1 Methods: VBScript Err Object

Clear Method	Clears all property settings.
Raise Method	Generate a run-time error.

- **VBScript Err Object Clear Method**

  Syntax : Err.Clear
  Clears all property settings of the Err object. Use Clear to explicitly clear the Err object after an error has been handled. This is necessary, for example, when you use deferred error handling with On Error Resume Next. VBScript calls the Clear method automatically whenever any of the following statements are executed:

  * On Error Resume Next
  * Exit Sub
  * Exit Function

- **VBScript Err Object Raise Method**

  Syntax: Err.Raise(number, source, description, helpfile, helpcontext)
  The On Error Resume Next statement, also called an error handler, is a procedure-level statement. It only remains in effect within the procedure that contains the on error declaration. The VBScript interpreter, like many

languages, raises an error to higher calling levels until it finds a procedure that handles the error. If none is found, the script halts and the user is presented with the error results by the interpreter. The Raise method is used for generating run-time errors. All the arguments of this method are optional except Number. However, if you use Raise, without specifying some arguments, and the property settings of the Err object contain values that have not been cleared, those values become the values for your error. When setting the number property to your own error code in an Automation object, you add your error code number to the constant vbObjectError. For example, to generate the error number 1050, assign vbObjectError + 1050 to the number property.

Arguments	Description
Number	A Long integer subtype that identifies the nature of the error. VBScript errors (both VBScript-defined and user-defined errors) are in the range 0-65535.
Source	A string expression naming the object or application that originally generated the error. When setting this property for an Automation object, use the form project.class. If nothing is specified, the programmatic ID of the current VBScript project is used.
Description	A string expression describing the error. If unspecified, the value in number is examined. If it can be mapped to a VBScript run-time error code, a string provided by VBScript is used as the description. If there is no VBScript error corresponding to number, a generic error message is used.
Helpfile	The fully qualified path to the Help file in which help on this error can be found. If unspecified, VBScript uses the fully qualified drive, path, and file name of the VBScript Help file.
Helpcontext	The context ID identifying a topic within helpfile that provides help for the error. If omitted, the VBScript Help file context ID for the error corresponding to the number property is used, if it exists

5.10.2 Properties: VBScript Err Object

Properties	Description
Description Property	The descriptive string associated with an error.
HelpContext Property	A context ID for a topic in a Windows help file.
HelpFile Property	A fully qualified path to a Windows help file.
Number Property	A numeric value identifying an error.
Source Property	The name of the object or application that originally generated the error.

- **Description Property**

  Returns or sets a descriptive string associated with an error. The Description property consists of a short description of the error. Use this property to alert the user to an error that you cannot or do not want to handle. When generating a user-defined error, assign a short description of your error to this property. If Description is not filled in, and the value of theVBScript Err Object Number Property corresponds to a VBScript run-time error, the descriptive string associated with the error is returned.

Syntax	Err.Description [= *stringexpression*]
Argument : Stringexpression	A string expression containing a description of the error.

- **HelpContext Property**

  Sets or returns a context ID for a topic in a Help File. If a Help file is specified

in the VBScript Err Object HelpFile Property, the HelpContext property is used to automatically display the Help topic identified. If both HelpFile and HelpContext are empty, the value of the VBScript Err Object Number Property is checked. If it corresponds to a VBScript run-time error value, then the VBScript Help context ID for the error is used. If the Number property does not correspond to a VBScript error, the contents screen for the VBScript Help file is displayed.

Syntax	Err.HelpContext [= *contextID*]
Argument : contextID	A valid identifier for a Help topic within the Help file. Optional.

- **HelpFile Property**

  Sets or returns a fully qualified path to a Help File. If a Help file is specified in HelpFile, it is automatically called when the user clicks the Help button (or presses the F1 key) in the error message dialog box. If the VBScript Err Object HelpContext Property contains a valid context ID for the specified file, that topic is automatically displayed. If no HelpFile is specified, the VBScript Help file is displayed.

Syntax	Err.HelpFile [= *contextID*]
Argument : contextID	The fully qualified path to the Help file. Optional.

- **Number Property**

  Returns or sets a numeric value specifying an error. Number is the Err object's default property. When returning a user-defined error from an Automation object, set Err.Number by adding the number you selected as an error code to the constant vbObjectError. For example, you use the following code to return the number 1051 as an error code:

```
Err.Raise Number:= vbObjectError + 1051, Source:= "SomeClass"
```

Syntax	Err.Number [= *errornumber*]
Argument : errornumber	An integer representing a VBScript error number or an SCODE error value.

- **Source Property**

  Returns or sets the name of the object or application that originally generated the error. The Source property specifies a string expression that is usually the class name or programmatic ID of the object that caused the error. Use Source to provide your users with information when your code is unable to handle an error generated in an accessed object. For example, if you access Microsoft Excel and it generates a Division by zero error, Microsoft Excel sets the VBScript Err Object Number Property to its error code for that error and sets Source to Excel application. Note that if the error is generated in another object called by Microsoft Excel, Excel intercepts the error and sets Err.Number to its own code for Division by zero. However, it leaves the other Err object (including Source) as set by the object that generated the error.

  Source always contains the name of the object that originally generated the error - your code can try to handle the error according to the error documentation of the object you accessed. If your error handler fails, you can use the Err object information to describe the error to your user, using Source and the other Err to inform the user which object originally caused the error, a description of the error, and so forth.

Syntax	Err.Source [= *stringexpression*]
Argument : stringexpression	A string expression representing the application that generated the error.

Case Study: Design a web page, which is used for accepting information regarding stocks. Make sure the form performs the possible validations. The simple example code below merely shows the form and accepts input but does not store the data or perform any action.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--


Sub cmd_submit_OnClick
' To continue the flow if any error occurs
on Error Resume Next

If document.form1.TxtName.Value = "" or  document.form1.TxtEMail.Value = ""
Then
Dim MyMessage
MyMessage = "Please enter your name and e-mail address."
MsgBox MyMessage, 0, "Incomplete Form Error"
   document.form1.TxtName.focus()
End If

' Check to see if the user entered anything.

 If (Len(document.form1.txtAge.value) = 0) Then

   MsgBox "You must enter your age before submitting."
   Exit Sub
   document.form1.txtAge.focus()


 End If
 If (Not(IsNumeric(document.form1.txtAge.value))) Then

   MsgBox "You must enter a number for your age."

   Exit sub

 End If

' Check to see if the age entered is valid.

 If (document.form1.txtAge.value < 0) Or  (document.form1.txtAge.value > 100)
Then

'   MsgBox "The age you entered is invalid."
        Err.clear
        Err.Raise 6
        Err.description = "Error - The age you entered is invalid."
        msgbox Err.description
  Exit Sub

 End If
```

```
' Data looks okay so submit it.

  MsgBox "Thanks for sharing your views."

  document.form1.submit

End Sub
-->

</SCRIPT>
</HEAD>
<BODY  bgColor="#ffffcc" Text="#000099" Link="#ff0000" VLink="#ff0000"
ALink="#009900">

<!--ENDOFADD -->
<FORM name = "form1" >
<CENTER>
<TABLE BORDER=8 CELLSPACING=1 CELLPADDING=0
WIDTH="600"><TR><TD>
<TABLE BORDER=5 CELLSPACING=1 CELLPADDING=0
WIDTH=100%><TR><TD>
<FONT SIZE="" FACE="Arial"><B><CENTER><H2>Sample Stock
Survey</H2></CENTER></B></FONT></TD>
<A NAME="ItemAnchor1"></A>
</TR></TABLE>
<HR>
<TABLE BORDER=5 CELLSPACING=1 CELLPADDING=0 WIDTH=100%>
<TR>
<TD ALIGN=RIGHT>*Name:</TD>
<TD><INPUT TYPE="TEXT" NAME="TxtName">
</TD>
<TD ALIGN=RIGHT>*E-Mail:</TD>
<TD><INPUT TYPE="TEXT" NAME="TxtEMail"></TD>
</TR>
<TR><TD ALIGN=RIGHT>Address:</TD>
<TD><INPUT TYPE="TEXT" NAME="TxtAddress"></TD>
<TD ALIGN=RIGHT>Age:</TD>
<TD><INPUT TYPE="TEXT" NAME="txtAge"></TD>
<A NAME="ItemAnchor1"></A>
</TR></TABLE>

<TABLE BORDER=0 CELLSPACING=1 CELLPADDING=0
WIDTH=100%><TR><TD>
<CENTER>
<FONT SIZE="" FACE="Arial">
<CENTER>
<TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0"
WIDTH="75%">
<BR></CENTER>

</FONT></CENTER></TD>
<TD>
<FONT SIZE="" FACE="Arial">
</TR>
</TABLE>

<TABLE BORDER=0 CELLSPACING=1 CELLPADDING=0>
<TR>
```

```
<TD>
<TABLE BORDER=0><TR><TD VALIGN="TOP">
<FONT SIZE="2" FACE="Arial" >
<B>Describe your investment experience</B>
</FONT>
</TD>
</TR>
</TABLE>


<TABLE BORDER=0>
<TR>
<TD VALIGN="TOP">
<FONT SIZE="2" FACE="Arial" >
<INPUT TYPE="RADIO" NAME="RESULT_RadioButton-3" VALUE="Radio-
0">beginner
</FONT>
</TD>
<TD VALIGN="TOP">
<FONT SIZE="2" FACE="Arial" >
<INPUT TYPE="RADIO" NAME="RESULT_RadioButton-3" VALUE="Radio-
1">intermediate
</FONT>
</TD>
<TD VALIGN="TOP">
<FONT SIZE="2" FACE="Arial" >
<INPUT TYPE="RADIO" NAME="RESULT_RadioButton-3" VALUE="Radio-
2">expert
</FONT>
</TD>
<TD VALIGN="TOP">
<FONT SIZE="2" FACE="Arial" >
</FONT>
</TD>
</TR>
</TABLE>
</TD>
<A NAME="ItemAnchor4"></A>
</TR>
</TABLE>
<TABLE BORDER=0 CELLSPACING=1 CELLPADDING=0>
<TR>
<TD>
<TABLE BORDER=0>
<TR>
<TD>
<FONT SIZE="2" FACE="Arial" >
<B>Types of Investments you make</B>
</FONT>
</TD>
</TR>
</TABLE>
<TABLE BORDER=0><TR><TD VALIGN="TOP">
<FONT SIZE="2" FACE="Arial" >
<INPUT TYPE="CHECKBOX" NAME="RESULT_CheckBox-4"
VALUE="CheckBox-0">Individual Stocks<BR>
<INPUT TYPE="CHECKBOX" NAME="RESULT_CheckBox-4"
VALUE="CheckBox-1">Options<BR>
```

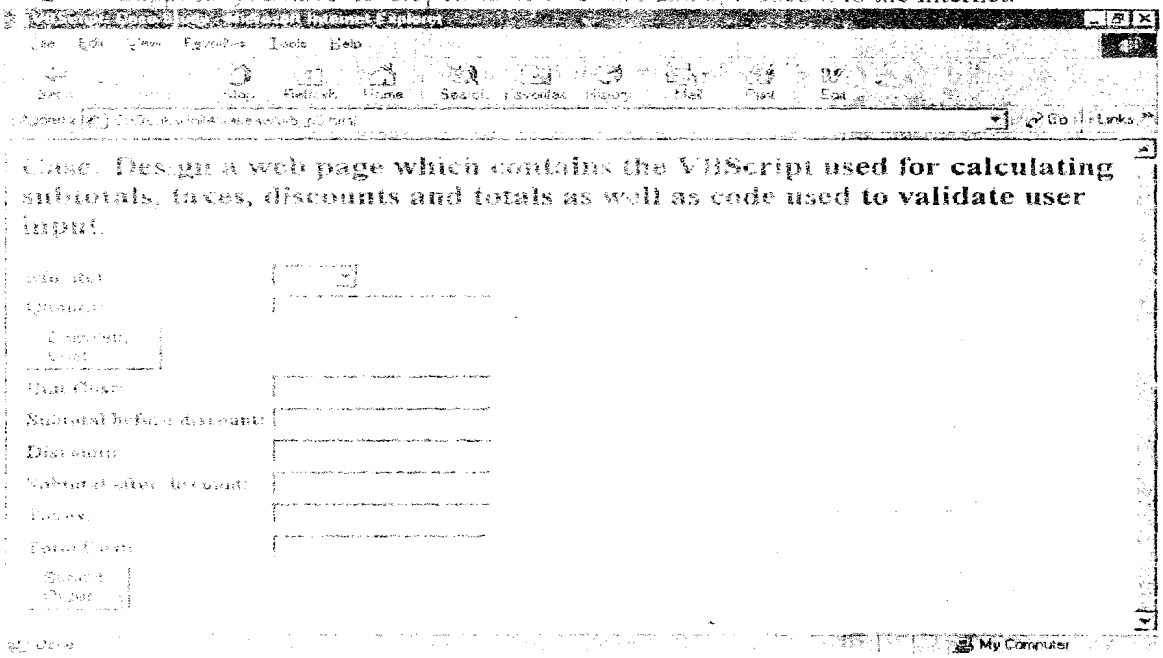
```
<INPUT TYPE="CHECKBOX" NAME="RESULT_CheckBox-4"
VALUE="CheckBox-2">Mutual Funds<BR>
<INPUT TYPE="CHECKBOX" NAME="RESULT_CheckBox-4"
VALUE="CheckBox-3">Real Estate
</FONT>
</TD>
</TR>
</TABLE>
</TD>
<A NAME="ItemAnchor5"></A>
</TR>
</TABLE>
<TABLE BORDER=0 CELLSPACING=1 CELLPADDING=0>
<TR>
<TD>
<TABLE BORDER=0>
<TR>
<TD  VALIGN="TOP">
<FONT  SIZE="2"  FACE="Arial" >
<B>How do you buy your stocks?</B>
</FONT>
</TD>
</TR>
</TABLE>
<TABLE BORDER=0><TR><TD VALIGN="TOP">
<FONT  SIZE="2"  FACE="Arial" >
<!--DROP_DOWN_TYPE -->
<SELECT NAME="RESULT_RadioButton-5">
<OPTION>
</OPTION>
<!--DROP_DOWN_TYPE NAME="RESULT_RadioButton-5" VALUE="Radio-0" -
-><OPTION>1) On-Line</OPTION>
<!--DROP_DOWN_TYPE NAME="RESULT_RadioButton-5" VALUE="Radio-1" -
-><OPTION>2) Touch Tone Trading
</OPTION>
<!--DROP_DOWN_TYPE NAME="RESULT_RadioButton-5" VALUE="Radio-2" -
-><OPTION>3) Broker Assisted
</OPTION>
<!--DROP_DOWN_TYPE NAME="RESULT_RadioButton-5" VALUE="Radio-3" -
-><OPTION>4) Other</OPTION>
</SELECT>
</FONT>
</TD>
</TR>
</TABLE>
</TD>
<A NAME="ItemAnchor6"></A>
</TR>
</TABLE>
<TABLE BORDER=0 CELLSPACING=1 CELLPADDING=0>
<TR><TD>
<TABLE BORDER=0 CELLSPACING=1 CELLPADDING=0>
<TR><TD>
<FONT  SIZE="2"  FACE="Arial" >
<B>What is your hot stock pick for this year?</B>
</FONT></TD>
</TR><TR>
<TD>
```

```
<FONT SIZE="2" FACE="Arial" >
<INPUT TYPE="TEXT" NAME="RESULT_TextField-6" SIZE="30"
MAXLENGTH="30">
</FONT>
</TD></TR>
</TABLE>
</TD>
<A NAME="ItemAnchor7"></A>
<TD>
<TABLE BORDER=0 CELLSPACING=1 CELLPADDING=0><TR><TD>
<FONT SIZE="2" FACE="Arial" >
<B>Stock Symbol if you know it</B>
</FONT>
</TD>
</TR><TR>
<TD>
<FONT SIZE="2" FACE="Arial" >
<INPUT TYPE="TEXT" NAME="RESULT_TextField-7" SIZE="4"
MAXLENGTH="4">
</FONT>
</TD></TR>
</TABLE>
</TD>
<A NAME="ItemAnchor8"></A>
</TR>
</TABLE>
<TABLE BORDER=0 CELLSPACING=1 CELLPADDING=0>
<TR><TD>
<TABLE BORDER=0 CELLSPACING=1 CELLPADDING=0>
<TR><TD VALIGN="BOTTOM">
<FONT SIZE="2" FACE="Arial" >
<B>Any Investment Advice for others?</B>
</FONT>
</TD>
</TR><TR>
<TD>
<FONT SIZE="2" FACE="Arial" >
<TEXTAREA NAME="RESULT_TextArea-8" ROWS="7" COLS="35"
WRAP="SOFT"></TEXTAREA>
</FONT></TD></TR>
</TABLE>
</TD>
<A NAME="ItemAnchor9"></A>
</TD></TR><TR><TD>


</CENTER>
</TD></TR>
</TABLE>
</TD></TR>
</TABLE>
</CENTER>
<BR>
<CENTER>
<INPUT TYPE="SUBMIT" NAME="cmd_submit" VALUE="Submit">
</CENTER>
</FORM>
</BODY>
```

Check Your Progress 3

1. Design the following web page with all Validations and Runtime error handling.
2. Suppose you have developed some freeware and uploaded it to the Internet.



Before downloading the software, the user has to fill up and submit an introductory form. Design the form as shown with validation of the inputs **(Case Study).**

5.11 SUMMARY

In this unit you have learned how to create Web pages that use VBScript. You have learned to add logic to your Web pages as with any other programming application. You have also learned Object support in VBScript. One of the very common and often used objects, the Dictionary Object has been discussed in detail. Validations form an important part of forms, which have been shown in many of the examples in this unit. You have also learned how to use the Err object for handling Runtime errors. You will now be able to develop Web pages like Enquiry forms, admission forms and so on. You have learned about the different operators and datatypes supported by VBScript and how to use them. You have learnt about functions and procedures, and how to make your programs modular using them. Coding conventions should be followed in order to make your programs readable and easily understandable by others. These include indentation of the code, naming the variables and constants and other formatting conventions such as those for comments.

5.12 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. The following code will do the job:

```
<SCRIPT LANGUAGE=vbscript>
<!--
```

```
str=trim(str)
Dim bln,ctr
bln=true
ctr=1

if(cint(str)<1)then
IsPrime=false
exit function
end if

if(cint(str)=1 or cint(str)=2 or cint(str)=3)then
IsPrime=true
exit function
end if

for ctr=cint(str)-1 to 2 step -1
if(cint(str)mod(ctr)=0)then
bln=false
end if
next

IsPrime=bln

End Function

Function IsEven(str)
str=trim(str)
if(str="0")then
IsEven=true
exit function
end if
if(cint(str)mod(2)=0)then
IsEven=true
else
IsEven=false
end if
End Function

Sub main()
str=text1.value
Dim blnEven,blnPrime

if not(isnumeric(str))then
msgbox "Not numeric"
exit sub
end if

if(IsEven(str))then
msgbox "Even"
else
msgbox "Odd"
end if

if(IsPrime(str))then
msgbox "prime"
else
msgbox "not prime"
end if
```

```
End Sub
}

//-->
</SCRIPT>
```

2. Use the following function to generate the fibonnacci series.

```
FUNCTION fib (n)
if (N=0) or (N=1) then
Fib = 1
  ElseFib = Fib (N- 1) + Fib (N-2)
END If
END FUNCTION
```

3. Use the following function to find the factorial

```
Function Factorial(n)
 Dim i
 If n < 0 Then
  Factorial = 0
  Exit Function
 End If
 Factorial = 1
 For i = 2 To n
  Factorial = Factorial * i
 Next
End Function
```

Check Your Progress 2

1.
```
<HTML>
   <HEAD>
   <META NAME="GENERATOR" Content="Microsoft Visual Studio
   6.0">
   <TITLE></TITLE>
   <SCRIPT LANGUAGE=vbscript>
   <!--

   Function IsPrime(str)
   str=trim(str)
   Dim bln,ctr
   bln=true
   ctr=1

   if(cint(str)<1)then
   IsPrime=false
   exit function
   end if

   if(cint(str)=1 or cint(str)=2 or cint(str)=3)then
   IsPrime=true
   exit function
   end if

   for ctr=cint(str)-1 to 2 step -1
   if(cint(str)mod(ctr)=0)then
   bln=false
```

```
end if
next

IsPrime=bln

End Function

Function IsEven(str)
str=trim(str)
if(str="0")then
IsEven=true
exit function
end if
if(cint(str)mod(2)=0)then
IsEven=true
else
IsEven=false
end if
End Function

Sub mysub()
str=text1.value

if not(isnumeric(str))then
msgbox "Not numeric"
exit sub
end if

if(rad(0).checked)then
if(IsEven(str))then
msgbox "Yes it is even"
else
msgbox "No it is not even"
end if
Exit sub
end if

if(rad(1).checked)then
if(IsEven(str))then
msgbox "No it is not odd"
else
msgbox "Yes it is odd"
end if
Exit sub
end if

if(rad(2).checked)then
if(IsPrime(str))then
msgbox "Yes it is a prime number"
else
msgbox "No it is not a prime number"
end if
Exit sub
end if

End Sub
```

```
//-->
</SCRIPT>
</HEAD>
<BODY>
<P>

Even

<INPUT type=radio name="rad" checked value="even">
Odd
<INPUT type=radio name="rad" value="odd">
Prime
<INPUT type=radio name="rad" value="prime">

</P>
<P><INPUT id=text1 name=text1></P>
<P><INPUT id=button1 type=button value=Button name=button1
LANGUAGE=javascript onclick="mysub()"></P>

</BODY>
</HTML>
```

2. Following is the code:

```
Function IsPalindrome(str)
Dim iStart,iEnd,ctr,blnPalin
str=trim(str)
blnPalin=true
iEnd=len(str)
iCnt=round(iEnd / 2)
iStart=1
for ctr=1 to cint(iCnt)
if(mid(str,iEnd,1)<>mid(str,iStart,1))then
IsPalindrome=false
Exit Function
end if
iStart=cint(iStart)+1
iEnd=cint(iEnd)-1
next
if(blnPalin=true)then
IsPalindrome=true
else
IsPalindrome=false
end if
End Function

Sub blankStr()
str=text1.value
if(trim(str)="")then
msgbox "Pls. enter a number"
exit sub
end if

if(IsPalindrome(str))then
msgbox "Yes"
else
msgbox "No"
end if
```

```
End Sub
```

Check Your Progress 3

1.

```
<HTML>

<H2> Case: Design a web page which contains the VBScript used for
calculating subtotals,
 taxes, discounts and totals as well as code used to validate user
input.</H2>

<HEAD>
<TITLE>VBScript: Case Study</TITLE>

<SCRIPT LANGUAGE="VBScript">
<!-- Add this to instruct non-IE browsers to skip over VBScript modules.
Option Explicit

Sub cmdCalculate_OnClick
 Dim AmountofDiscount
 Dim AmountofTax
 Dim DISCOUNT_LIMIT
 Dim DISCOUNT_RATE
 Dim SubtotalBefore
 Dim SubtotalAfter
 Dim TAX_RATE
 Dim TotalCost

 If (Len(Document.frmCaseStudy.txtQuantity.Value) = 0) Then
  MsgBox "You must enter a quantity."
  Exit Sub
 End If

 If (Not IsNumeric(Document.frmCaseStudy.txtQuantity.Value)) Then
  MsgBox "Quantity must be a numeric value."
  Exit Sub
 End If

 If (Len(Document.frmCaseStudy.cmbProducts.Value) = 0) Then
  MsgBox "You must select a product."
  Exit Sub
 End If


 DISCOUNT_LIMIT = 1000
 DISCOUNT_RATE = .10
 TAX_RATE = 0.06
' Calculate the subtotal for the order.
 SubtotalBefore = Document.frmCaseStudy.txtQuantity.Value *
Document.frmCaseStudy.lblUnitCost.value


 If (SubtotalBefore > DISCOUNT_LIMIT) Then
  AmountofDiscount = SubtotalBefore * DISCOUNT_RATE
 Else
  AmountofDiscount = 0
 End If
```

```
  SubtotalAfter = SubtotalBefore - AmountofDiscount

' Calculate taxes and total cost.
  AmountofTax = SubtotalAfter * TAX_RATE
  TotalCost = SubtotalAfter + AmountofTax

' Display the results.
  Document.frmCaseStudy.lblSubtotalBefore.value = SubtotalBefore
  Document.frmCaseStudy.lblDiscount.value = AmountofDiscount
  Document.frmCaseStudy.lblSubtotalAfter.value = SubtotalAfter
  Document.frmCaseStudy.lblTaxes.value = AmountofTax
  Document.frmCaseStudy.lblTotalCost.value = TotalCost

End Sub

Sub cmdSubmit_onClick
' Submit this order for processing.
  MsgBox "Your order has been submitted."
  Document.frmCaseStudy.Submit
End Sub

Sub cmbProducts_onchange()

   Select Case Document.frmCaseStudy.cmbProducts.SelectedIndex
    Case 1
      Document.frmCaseStudy.lblUnitCost.value = 1590
    Case 2
      Document.frmCaseStudy.lblUnitCost.value = 880
    Case 3
      Document.frmCaseStudy.lblUnitCost.value = 1940
    Case Else
      Document.frmCaseStudy.lblUnitCost.value = 0
  End Select
End Sub

-->
</SCRIPT>

</HEAD>

<BODY>
<FORM NAME="frmCaseStudy">
 <TABLE>
  <TR>
   <TD><B>Monitor:</B></TD>
   <TD>
  <SELECT NAME = "cmbProducts">
          <OPTION VALUE ="0" ></OPTION>
          <OPTION VALUE ="1" >Item 1</OPTION>
          <OPTION VALUE ="2">Item 2</OPTION>
          <OPTION VALUE ="3">Item 3</OPTION>
  </SELECT>
   </TD>
  </TR>
  <TR>
   <TD><B>Quantity:</B></TD>
   <TD>
  <INPUT TYPE = "TEXT" NAME ="txtQuantity" >
```

```
    </TD>
   </TR>
   <TR>
    <TD><INPUT TYPE="Button" NAME="cmdCalculate"
VALUE="Calculate
Cost"></TD>
    <TD></TD>
   </TR>
   <TR>
    <TD><B>Unit Cost:</B></TD>
    <TD>
  <INPUT TYPE = "TEXT" NAME ="lblUnitCost" >
    </TD>
   </TR>
   <TR>
    <TD><B>Subtotal before discount:</B></TD>
    <TD>
  <INPUT TYPE = "TEXT" NAME ="lblSubtotalBefore" >
    </TD>
   </TR>
   <TR>
    <TD><B>Discount:</B></TD>
    <TD>
  <INPUT TYPE = "TEXT" NAME = "lblDiscount" >
    </TD>
   </TR>
   <TR>
    <TD><B>Subtotal after discount:</B></TD>
    <TD>
  <INPUT TYPE = "TEXT" NAME ="lblSubtotalAfter" >
    </TD>
   </TR>
   <TR>
    <TD><B>Taxes:</B></TD>
    <TD>
  <INPUT TYPE = "TEXT" NAME ="lblTaxes" >
    </TD>
   </TR>
   <TR>
    <TD><B>Total Cost:</B></TD>
    <TD>
  <INPUT TYPE = "TEXT" NAME ="lblTotalCost" >
    </TD>
   </TR>
   <TR>
    <TD><INPUT TYPE="Button" NAME="cmdSubmit"
VALUE="Submit
Order"></TD>
    <TD></TD>
   </TR>
  </TABLE>
 </FORM>

</BODY>

</HTML>
```

2.
```
<HTML>
<HEAD>
<TITLE>ABC Software's Registration Page</TITLE>
</HEAD>
<BODY BGCOLOR=WHITE> <CENTER>
<FONT SIZE=7 FACE="Arial" COLOR=BLUE>ABC</FONT>
<FONT SIZE=5 COLOR=BLACK>Registration</FONT>
</CENTER>
<HR COLOR="BLUE">
<FONT FACE="COMIC Sans MS"> Thank you for taking the time to
download and test our new product. In order to serve you well, we ask that you
submit the following information before downloading
<FONT FACE=ARIAL COLOR=BLUE>ABC</FONT>software.
<HR COLOR=RED WIDTH=75%> <CENTER>
<TABLE BORDER BORDERCOLOR="BLUE">
<TR>
<TD ALIGN=RIGHT>*Name:</TD>
<TD><INPUT TYPE="TEXT" NAME="TxtName">
</TD>
</TR>
 <TR>
<TD ALIGN=RIGHT>*E-Mail:</TD>
<TD><INPUT TYPE="TEXT" NAME="TxtEMail"></TD>
</TR>
<TR><TD ALIGN=RIGHT>Address:</TD>
<TD><INPUT TYPE="TEXT" NAME="TxtAddress"></TD>
</TR>
<TR>
<TD ALIGN=RIGHT>City:</TD>
<TD><INPUT TYPE="TEXT" NAME="TxtCity"></TD>
</TR>
 <TR>
<TD ALIGN=RIGHT>State:</TD>
<TD><INPUT TYPE="TEXT" NAME="TxtState"></TD>
</TR>
<TR>
<TD ALIGN=RIGHT>Zip:</TD>
<TD><INPUT TYPE="TEXT" NAME="TxtZip"></TD>
</TR>
 <TR>
<TD ALIGN=RIGHT>Country:</TD>
<TD><INPUT TYPE="TEXT" NAME="TxtCountry"></TD>
</TR>
<TR>
<TD COLSPAN=2 ALIGN=CENTER><INPUT TYPE="SUBMIT"
NAME="Btn1" VALUE="Send It!"><BR> <FONT SIZE=2>* Required
Field</FONT></TD>
</TR>
</TABLE>
<HR COLOR=RED WIDTH=75%>
<FONT SIZE=2>Thanks!</FONT>
<SCRIPT LANGUAGE="VBScript">
<!--
Sub Btn1_OnClick
If TxtName.Value="" Or TxtEMail.Value="" Then
Dim MyMessage
MyMessage = "Please enter your name and e-mail address."
MsgBox MyMessage, 0, "Incomplete Form Error"
```

```
End If

End Sub -->
</SCRIPT>
</BODY>
</HTML>
```

5.13 FURTHER READINGS

1. *Learning VBScript* by Paul Lomax. Publisher: O'Reilly. Year: 2001
2. *Instant VBScript* by Alex Homer, Darren Gill. Publisher: Wrox Press Inc Pub. Year: 2003
3. *VBscript for the World Wide Web* by Paul Thurrott, Nolan Hester. Publisher: Peachpit Press. Year: 1999
4. *Teach Yourself Vbscript in 21 Days* by Keith Brophy, Timothy Koets. Publisher: Sams Publishing. year: 1996